

# **Plug-In Software Development Kit**

Version 6

September 18, 2003

## **Orientation**

[Introduction](#)

[How to Use the Documentation](#)

[What You Need to Get Started](#)

[Recommended Reading](#)

[Plug-In Types Overview](#)

## Guidance

[Introduction](#)

[Fundamental Concepts of the 3ds max SDK](#)

[Plug-In Architecture Overview](#)

[Overview of the Principal Classes](#)

[Writing Plug-In Applications](#)

[Building the Sample Files](#)

[Must Read Sections for All Developers](#)

[Must Read Sections by Plug-In Type](#)

[What's New in the 3ds max 4.0 SDK](#)

[Required Changes to 3ds max 3.x Plug-Ins for 3ds max 4.0](#)

[General Terminology](#)

[Common Problems and Solutions](#)

[Advanced Topics](#)

## Reference

[Introduction](#)

[Typeface Conventions](#)

[Data Types](#)

[Generate a Class\\_ID](#)

[Class Interface](#)

[Class List by Category](#)

## **Development Support**

[Sparks Developer Program](#)

[Sparks Developer Knowledgebase](#)

[Sparks Addin for MS VC++ 6 IDE](#)

[Sparks Addin for MS VC++ 7 IDE](#)

[Sparks Message Archives](#)

[Sparks Documentation Updates](#)

[Sparks Webboard](#)

[Sparks Downloads](#)

[Sparks Incident Reporting](#)

## **Introduction**

The 3ds max Software Development Kit (SDK) is an object-oriented programming library for creating plug-in applications for 3ds max. The SDK provides a comprehensive set of classes that developers can combine and extend to create seamlessly integrated plug-in applications.

Using the SDK one can create a great variety of plug-ins. In fact, much of 3ds max itself is written as plug-in applications.

## How to Use the Documentation

The documentation for the SDK is an on-line help file. This format provides easy hypertext jumping between related topics. The on-line format also allows developers to copy source code from the documentation to paste into plug-ins. This is especially handy in the Reference sections.

### **The on-line help is broken into three modules: Orientation, Guidance, and Reference.**

The **Orientation** module gives you an overview of how to get started with the SDK. It lists what hardware and software is needed to develop 3ds max plug-in applications. This module discusses what technical expertise is required. There is a section on recommended reading in C++, CG and Windows programming.

This module also provides an introduction to the types of plug-ins developers may create using the SDK. Most of the descriptions in this section allow hyperlink jumps to the main classes these plug-ins are derived from. Additionally, relevant sample code available in the SDK is listed here.

The **Guidance** module is intended to enable you to learn to accomplish a specific task. It begins with a description of the 3ds max plug-in architecture that provides a diagram of the principal classes. There is a section describing information common to all plug-in development.

There is a section for experienced 3D Studio IPAS programmers making the transition to MAX. This section discusses the similarities and differences in the IPAS and 3ds max programming architectures. The changes in development environment, plug-in/system control flow, user interface, and graphics programming support are reviewed. This section also discusses the six types of IPAS routines possible under 3D Studio DOS and how analogous plug-ins may be implemented under the 3ds max architecture.

The General Terminology section defines terms used throughout the SDK documentation. These terms also appear as popup help in other parts of the documentation.

The Advanced Topics section presents detailed information about key concepts in using the APIs. The topics presented in this section cross class boundaries, and are instead based on specific implementation issues developers need to understand when designing plug-ins for MAX. Developers getting started with the SDK should take a look through the various Advanced Topics sections. This will help in becoming familiar with what information is available.

The **Reference** module delivers comprehensive reference information about each class used in the API. At the beginning of each class description is an overview of the purpose of the class. All the methods and operators used by each class are documented.

This section also has a class list organized by category. This provides an organized way to jump to the many classes in the SDK. These categories are: Main Plug-In Classes, Interface Classes, Geometry Classes, User Interface Classes, and Miscellaneous Utility Classes.

## What You Need to Get Started

See Also: [Recommended Reading](#).

## **Technical Expertise**

In order to use the SDK you need to know how to program in C++. You should also be familiar with object-oriented programming concepts. This document can get you started, but it does not teach C++ or object-oriented programming. It also assumes a basic understanding of the use of 3ds max and the terminology of its user interface. This is required to understand the way the plug-in applications relate to the entire 3ds max user environment. The documents also refers to specific 3ds max plug-ins present in the core program. Familiarity with these plug-ins from a users perspective will help you understand the example source code.

It is also assumed you are familiar with the Microsoft Visual C++ compiler and development tools.

## **Software**

Microsoft Windows NT version 4.0 or greater, including Windows 2000, XP.

Microsoft Visual C++ version 6.0 (3ds max versions 5.1 and lower).

Microsoft Visual C++ version 7.0 (3ds max 6).

## **Hardware**

The same as required for 3ds max itself.

## **Recommended Reading**

This section lists several C++, Computer Graphics and Windows books you may find helpful.

## **C++ and Object Oriented Programming:**

Bjarne Stroustrup, *The C++ Programming Language* (Second edition, Reading, Massachusetts: Addison-Wesley Publishing Company, 1994), ISBN: 0-201-53992-6.

Bruce Eckel, *Thinking in C++* (Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1995), ISBN: 0-13-917709-4.

Al Stevens, *Teach Yourself... C++* (Fourth edition, New York, New York: MIS:Press). ISBN 1-55828-406-0.

## **Computer Graphics Programming:**

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, Richard L. Phillips, Introduction to Computer Graphics (Reading, Massachusetts: Addison-Wesley Publishing Company, 1993). ISBN: 0-201-60921-5.

John Vince, 3-D Computer Animation (Reading, Massachusetts: Addison-Wesley Publishing Company, 1993). ISBN: 0-201-62756-6.

Alan Watt and Mark Watt, Advanced Animation and Rendering Techniques (Reading, Massachusetts: Addison-Wesley, 1992). ISBN: 0-201-54412-1

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, Computer Graphics Principles and Practice (Second edition in C, Reading, Massachusetts: Addison-Wesley Publishing Company, 1995). ISBN: 0-201-84840-6.

Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, Texturing and Modeling A Procedural Approach (Cambridge, MA: Academic Press, Inc., 1994). ISBN: 0-12-228760-6. This book is very applicable to 3ds max texture plug-in creation.

Les Piegl, Wayne Tiller, The NURBS Book (Berlin Heidelberg New York: Springer-Verlag, 1995). ISBN: 3-540-55069-0. This book provides a great deal of information on B-spline curves and surfaces (NURBS).

Gerald Farin, Curves and Surfaces for Computer Aided Geometric Design : A Practical Guide (San Diego, CA, 1993). ISBN: 0-12-249052-5. This book provides information applicable to the creation of spline and patch plug-ins.

James D. Murray and William vanRyper, Encyclopedia of Graphics File Formats (Second Edition, O'Reilly & Associates, Inc., 1996, 1994). ISBN: 1-56592-161-5. This book provides information about various graphic file formats that are applicable to the creation of image loader / saver plug-ins.

## **Windows Programming:**

Charles Petzold, Programming Windows 95 (Redmond, Washington: Microsoft Press, 1996). ISBN: 1-55615-676-6.

Jeffrey Richter, Advanced Windows The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95 (Microsoft Press, 1995). ISBN: 1-55615-677-4.

## Plug-In Types Overview

See Also: [Plug-In Architecture Overview](#).

The following is a brief description of the categories of plug-ins one may create using the 3ds max Software Development Kit.

Sample code in the SDK is available to demonstrate techniques for programming many of these plug-in types. Additional sample code is available to [Sparks Developers](#). Also see the section on [Debugging](#).

The 3ds max 4.0 SDK comes provided with an SDK appwizard that can be used with Microsoft Visual C++. This appwizard allows you to quickly generate skeleton source code for a variety of different plugin types and essentially replaced the previous R3\_SKELETON projects included in the previous release of the software.

The generated code from the appwizard provides a quick way to start building plugins. Each project follows a standard structure with regard to headers and code files. Developers should find it easy to take this generated code and can start creating plugins without manually setting up the entire project from scratch.

## **Procedural Objects**

Procedural Objects are the general class of developer-defined objects that can be used in 3ds max.

## Geometric Objects

These are the only procedural objects that actually get rendered.

### Primitives

Primitive objects such as boxes, spheres, cones, cylinders, and tubes are implemented as procedural geometric objects. These are derived from class [GeomObject](#) or [SimpleObject](#). Example code may be found in **\MAXSDK\SAMPLES\OBJECTS\BOX.CPP, SPHERE.CPP, CONE.CPP**, etc.

### Particles

Developers may create procedural object particle system plug-ins. Some examples are particles that depend upon procedural motion like fireworks, explosions, and water, or particles that track the surface of objects like electrical fields or flame. Applications can be derived from [ParticleObject](#) or [SimpleParticle](#). Example code is available in **\MAXSDK\SAMPLES\OBJECTS\RAIN.CPP** and **\MAXSDK\SAMPLES\MODIFIERS\GRAVITY.CPP**.

### Loft Objects

The 3ds max Loftter is implemented as a procedural object plug-in. Developers may define other modeling modules that fit this form.

### Compound Objects

Compound objects take several objects and combine them together to produce a new object. Examples are Booleans (which produce a new object using operations Union, Intersection and Difference) and Morph objects. Sample code for the boolean object can be found in **\MAXSDK\SAMPLES\OBJECTS\BOOLOBJ.CPP**.

### Patches

Developers can create patch modeling systems that work inside MAX. The TriPatch and Patch Grid are examples of patch objects. These plug-ins are derived from [PatchObject](#). See **\MAXSDK\SAMPLES\OBJECTS\PATCHGRD.CPP**, and **TRIPATCH.CPP** for sample code.

### NURBS

The NURBS API provides an interface into the NURBS objects used by

MAX. Using the API developer can create new NURBS objects or modify existing ones. See the Advanced Topics section [Working with NURBS](#) for more information.

## Helper Objects

Helper objects are items such as dummy objects, grids, tape measurers and point objects. These objects may be derived from classes **HelperObject** or **ConstObject**. Sample code may be found in **\MAXSDK\SAMPLES\OBJECTS\HELPERS\GRIDHELP.CPP**, and **PHELP.CPP**. [See Class HelperObject](#) and [Class ConstObject](#).

## Shape Objects

These are shapes such as Circles, Arcs, Rectangles, Donuts, etc. New splines may be subclassed off [SimpleSpline](#). Sample code may be found in **\MAXSDK\SAMPLES\OBJECTS\CIRCLE.CPP, ELLIPSE.CPP, ARC.CPP**, etc.

## Procedural Shapes

These are shapes that are defined procedurally. An example procedural shape, Helix, may be found in **\MAXSDK\SAMPLES\OBJECTS\HELIX.CPP**. When an edit spline modifier is applied to a procedural shape it is converted to splines with segments that provide vertices in a linear approximation of the shape. This allows the procedural shape to be edited. Procedural shapes may be derived from class [SimpleShape](#). Other examples include Procedural lines, and Text.

Any of these objects can be edited with an edit spline modifier or extruded or surfrev'd.

## Lights

Developers may create custom plug-in lights. There are several classes from which light plug-ins may be derived. These are [LightObject](#), and [GenLight](#).

Example code may be found in

**\MAXSDK\SAMPLES\OBJECTS\LIGHT.CPP.**

## Cameras

Developers may create custom cameras. The two classes from which cameras may be derived are [CameraObject](#) and [GenCamera](#). An example of a plug-in camera may be found in

**`\MAXSDK\SAMPLES\OBJECTS\CAMERA.CPP.`**

## Object Modifiers

Object modifiers are applied to objects in their own local transform space to modify them in some way. Deformations like Bend, Taper, and Twist are examples of Object Modifier plug-ins. Example code may be found in **\MAXSDK\SAMPLES\MODIFIERS\BEND.CPP, TAPER.CPP**, etc. Extrude and Surfrev are also object modifier plug-ins. Sample code for Extrude can be found in **\MAXSDK\SAMPLES\MODIFIERS\EXTRUDE.CPP**. Developers may also create surface modifier plug-ins to alter smoothing groups, texture coordinates, and material assignments. See [Class Modifier](#) or [Class SimpleMod](#).

## **Edit Modifiers**

These plug-ins allow specific object types to be edited. For example, an Edit Mesh modifier allows objects that can convert themselves into triangle meshes to be edited, while an Edit Patch modifier allows objects that can convert themselves into patches to be edited. Edit modifiers typically allow the user to select sub-object elements of the object (vertices faces and edges in the case of the Edit Mesh modifier) and perform at least the standard move/rotate/scale transformations to them. They may also support additional operations (such as the extrude option of the Edit Mesh modifier). Example code may be found in **\MAXSDK\SAMPLES\MODIFIERS\EDITMESH.CPP**.

## Space Warps

Space Warps are basically object modifiers that affect objects in world space instead of in the object's local space (they were originally called 'world space modifiers'). Space Warps are non-rendering objects that affect other objects in the scene based on the position and orientation of the other objects that are bound to the Space Warp object. For example, the Ripple Space Warp applies a sine wave deformation to objects bound to it. Other examples of Space Warps include things like explosions, wind fields, and gravity. Sample code may be found in **\MAXSDK\SAMPLES\MODIFIERS\SINWAVE.CPP**.

Space warps are created in the Creation branch of the command panel, which makes them slightly different from regular modifiers (because they are combinations of space warp objects and space warp modifiers).

Space warps may also affect particle systems. For example, a force field can be applied to a particle system by a space warp. The force field provides a function of position in space, velocity and time that gives a force. The force is then used to compute an acceleration on a particle which modifies its velocity. For details see **\MAXSDK\INCLUDE\OBJECT.H**,

**\MAXSDK\SAMPLES\MODIFIERS\GRAVITY.CPP**, and  
**\MAXSDK\SAMPLES\OBJECTS\RAIN.CPP**.

A collision object can also be applied to a particle system by a space warp. The collision object checks a particle's position and velocity and determines if the particle will collide with it in the next period of time. If so, it modifies the position and velocity.

## Controllers

Controller plug-ins are the objects in 3ds max that control animation. Controllers come in different types based on the data they control. The most common controllers are interpolating or keyframe controllers. Other controller types are position/rotate/scale, mathematical expressions and fractal noise. Example controller code may be found in

**\MAXSDK\SAMPLES\HOWTO\PCONTROL\PCONTROL.CPP**,  
and **NOIZCTRL.CPP** etc. Controllers may be derived from [Class Control](#) or [Class StdControl](#).

## **Systems**

Systems are basically combinations of more than one type of procedural object, along with optional controllers, or modifiers, or space warps all working together. These plug-ins can provide high-order parametric control over very complex systems. An example system is Biped which uses procedural objects and master/slave controllers.

## **File Import**

These plug-ins allows 3D geometry and other scene data to be imported and exported to file formats other than the 3ds max format. An example file import plug-in may be found in `\MAXSDK\SAMPLES\IMPEXP\3DSIMP.CPP`. These plug-ins are derived from [Class SceneImport](#).

## **File Export**

These plug-ins allows 3D geometry and other scene data to be exported to file formats other than the 3ds max format. Sample code may be found in **\MAXSDK\SAMPLES\IMPEXP\3DSEXP.CPP**. These plug-ins are derived from [Class SceneExport](#).

## Atmospheric Plug-Ins

These plug-ins are used for atmospheric effects. MAX's Fog, and Volume Fog are two atmospheric plug-ins. Certain particle system-ish effects can be accomplished via atmospherics more efficiently. For example, a fire effect that is not done with particles but rather as a function in 3D space (the Combustion plug-in is a good example of this). Instead of rendering particles you traverse a ray and evaluate a function. These plug-ins typically use very little memory relative to a particle system equivalent. Atmospheric plug-ins also have the ability to reference items in the scene. (For example, MAX's Volume Lights reference lights in the scene.) These plug-ins are derived from [Class Atmospheric](#).

## Plug-In Materials

These are additional developer-defined material types. Examples are Standard, Mix, and Multi/Sub-Object materials. New materials are subclassed from [Class Mtl](#). Also see the section [Working with Materials and Textures](#). The sample code for these plug-ins is in **\MAXSDK\SAMPLES\MATERIALS**.

## Plug-In Textures

Procedural Texture plug-ins define 2- or 3-dimensional functions which can be assigned as maps within the shader tree architecture of the Materials Editor. Maps may be assigned as ambient, diffuse, specular, shininess, shininess strength, self-illumination, opacity, filter (transmission) color, bump, reflection and refraction maps. These functions may vary over time to produce animated effects. There are both 2D and 3D procedural textures, compositors and color modifiers. These plug-ins are derived from [Class Texmap](#). Also see the section [Working with Materials and Textures](#). The sample code for these plug-ins is in `\MAXSDK\SAMPLES\MATERIALS`.

## **2D Procedural**

Examples of 2D texture are **BITMAP.CPP** and **CHECKER.CPP**.

## **3D Procedural**

Examples of 3D textures are **MARBLE.CPP** and **NOISE.CPP**.

## **Compositor**

Some examples of compositors are **MASK.CPP** and **MIX.CPP**.

## **Color Modifier**

An example color modifier is RGB **TINT.CPP**.

Developers that have created a 3D Studio/DOS SXP and a corresponding 3ds max texture plug-in may want to have a look at [Class Tex3D](#). It provides a way to have an instance of your 3ds max texture plug-in created automatically when the corresponding SXP is found in a 3DS file being imported.

## Image Processing Plug-Ins

### Filters

Filters may be used to alter images in the video post data stream. Filters may operate on a single image or may combine two images together to create a new composite image. These plug-ins are derived from [Class ImageFilter](#). Also see the section [Working with Bitmaps](#).

### One Pass Filter

This plug-in type allows a single image in the video post data stream to be adjusted in some manner. An example plug-in of this type is

**\MAXSDK\SAMPLES\POSTFILTERS\NEGATIVE\NEGATIVE**

### Layer Filter

This plug-in allows two images to be composited to create a single new image. An example of this type of plug-in is

**\MAXSDK\SAMPLES\POSTFILTERS\ADD\ADD.CPP** or  
**\MAXSDK\SAMPLES\POSTFILTERS\ALPHA\ALPHA.CPP.**

## **G Buffer**

A G-buffer is used to store, at every pixel, information about the geometry at that pixel. All plug-ins in video post can request various components of the G-buffer. When video post calls the renderer it takes the sum of all the requests and asks the renderer to produce the G-buffer. Developers can use this information to create visual effects that are impossible to achieve without access to a G-buffer. See [Class GBuffer](#).

## Rendering Effects

This plug-in type is available in release 3.0 and later only.

There is a new item under the Rendering menu which displays the Rendering Effects dialog. From this modeless dialog, the user can select and assign a new class of plug-in, called a "Rendering Effect," which is a post-rendering image-processing effect. This lets the user apply image processing **without** using Video Post, and has the added advantage of allowing animated parameters and references to scene objects. The base class for these plug-ins is [Class Effect](#).

Sample code is available in the directory

**\MAXSDK\SAMPLES\RENDER\RENDEREFFECT.**

## Snap Plug-Ins

This plug-in type is available in release 2.0 and later only.

This plug-in type allows custom points to be provided to the 3ds max snapping system. For example a door plug-in could provide a custom snap for the hinge center. See [Class Osnap](#) for details. For sample code see **`\MAXSDK\SAMPLES\SNAPS\SPHERE\SPHERE.CPP`**.

## **Image Loading and Saving Plug-Ins**

Image loading and saving plug-ins allow the image file formats loaded and saved by 3ds max to be extended. An example is the JPEG loader / saver. Sample code may be found in the sub-directories of **\MAXSDK\SAMPLES\IO**. These plug-in types are derived from [Class BitmapIO](#). Device drivers are also derived from this class. See the sample code in **\MAXSDK\SAMPLES\IO\WSD\WSD.CPP**.

## Utility Plug-Ins

These plug-ins are useful for implementing modal procedures such as 3D paint, dynamics, etc. These plug-ins are accessed from the Utility page of the command panel. Example code may be found in the subdirectory **\MAXSDK\SAMPLES\UTILITIES**. These plug-ins are sub-classes of [Class UtilityObj](#).

## **Global Utility Plug-Ins**

This plug-in type is available in release 3.0 and later only.

These simple utility plug-ins are loaded at boot time, after initialization, but before the message loop starts, and remain loaded. This is how the new 3ds max COM/DCOM interface is implemented. For details see [Class GUP](#).

## Track View Utility Plug-Ins

These plug-ins are launched via the 'Track View Utility' icon just to the left of the track view name field in the toolbar. Clicking on this button brings up a dialog of all the track view utilities currently installed in the system. Most utilities will probably be modeless floating dialogs, however modal utilities may be created as well. These can provide general utility functions that operate on keys, time or function curves in Track View. Sample code is available in **\MAXSDK\SAMPLES\UTILITIES\RANDKEYS.CPP**, **ORTKEYS.CPP** and **SELKEYS.CPP**. These plug-ins are sub-classes off [Class TrackViewUtility](#).

## Plug-In Renderers

Plug-In renderers are derived from the class **Renderer**. The standard 3ds max scanline renderer is itself derived from this class. In a trivial sense, there are only a few methods to implement to create a renderer: **Open()**, **Render()**, **Close()**, **ResetParams()** and **CreateParamDlg()**. See [Class Renderer](#) for more details on this plug-in type.

## Shader Plug-Ins

This plug-in type is available in release 3.0 and later only.

This plug-in type works with the new Standard material. It allows plug-in developers to add additional shading algorithms to the drop down list of available options (previously Constant, Phong, Blinn, Metal). This was only possible previously by writing an entire Material plug-in (which could be a major undertaking). See the base class for this plug-in type [Class Shader](#) for details.

## Sampler Plug-Ins

This plug-in type is available in release 3.0 and later only.

This plug-in type works with the Standard material of release 3. A Sampler is a plug-in that determines where inside a single pixel the shading and texture samples are computed. The user interface of Samplers appears in the Super Sampling rollout in the Sampler dropdown. See [Class Sampler](#) for details.

## **Anti-Aliasing Filter Plug-Ins**

This plug-in type is available in release 3.0 and later only.

This plug-in type is used for filtering and anti-aliasing the image. Documentation for the base class for these filters is in [Class FilterKernel](#). Sample Code is available in the subdirectory

**\MAXSDK\SAMPLES\RENDER\AAFILTERS.**

## Shadow Generator Plug-Ins

This plug-in type is available in release 3.0 and later only.

The generation of shadows is accessible via this plug-in type. The standard 3ds max mapped and raytraced shadows have are plug-ins of this form. See [Class ShadowType](#) and [ShadowGenerator](#) for details. There is also a handy class for creating shadow map buffers. See [Class ShadBufRenderer](#).

## **Sound Plug-In**

A sound plug-in can take control of sound/music production in MAX. These plug-ins control not only the sounds they generate but also the system clock. They can thus coordinate the timing of external sound input / output devices with the animation. Sound plug-ins can provide their user interface as part of the 3ds max Track View.

Sound plug-ins are derived from [Class SoundObj](#).

## **Color Selector Plug-In**

This plug-in type is available in release 3.0 and later only.

This plug-in type provides the user with a custom color picker that appears whenever a standard 3ds max color swatch control is clicked. These plug-ins are selected in the General tab of the Preferences dialog. The color picker chosen is saved in the 3DSMAX.INI file in the "ColorPicker" section so that the choice is maintained between sessions. If the DLL for the selected color picker is not available, it will always default back to the "Default" color picker. See [Class ColPick](#) for details.

## **Front End Controllers**

These plug-ins allow a developer to completely take over the 3ds max user interface. This includes the toolbar, pulldown menus, and command panel. See [Class FrontEndController](#) for details.

## **Motion Capture Input Devices**

Motion Capture Input Device plug-ins can now be written that plug-in to the 3ds max motion capture system. See [Class IMCInputDevice](#) for details. Sample code is available in the subdirectory **\MAXSDK\SAMPLES\MOCAP**.

## **Image Viewer Plug-In**

An image viewer is available from the 3ds max File menu under View File. A developer may replace the viewer DLL launched by this command to provide enhanced functionality for image browsing. The source code for this viewer is in **\MAXSDK\SAMPLES\VIEWFILE\VIEWFILE.CPP**. This plug-in is derived from [Class ViewFile](#).

## Notification Program

There is a program whose source code is in `\MAXSDK\SAMPLES\UTILITIES\notify\notify.cpp`. This program gets invoked by the network manager to handle network progress notifications.

A developer may write another "Notify" program in order to do any proprietary type of notifications. Note that "Notify" can be either a "\*.exe", a "\*.bat", or a "\*.cmd" executable. This allows a user to create a simple script file to do something without having to resort to writing a binary program.

The current **Notify.exe** is very simple as it is used simply as a demonstration. It plays a different wave file for each of the event types. If invoked with no command line, it will bring up a dialog box asking the user to define each of the three wave files. The dialog has "Browse" buttons next to each wave file field which puts the user right into the Windows' "Media" directory where wave files are saved. There are also "play" buttons next to each sound so they can be tested.

## Guidance Introduction

This module of the documentation is intended to enable a developer to learn to accomplish a specific task in programming 3ds max. There is a section on the 3ds max plug-in architecture that provides a diagram of the principal classes. There is a section describing information common to all plug-in development. This module provides How To sections for several of the plug-in types including procedural objects, object space modifiers, space warps, controllers and system plug-ins. These sections include commented source code and descriptions of categories of methods developers must implement to create each plug-in type. There is a section for experienced 3D Studio IPAS programmers making the transition to MAX. This section discusses the similarities and difference in the IPAS and 3ds max programming architectures. The changes in development environment, plug-in/system control flow, user interface, and graphics programming support are reviewed. This section also discusses the six types of IPAS routines possible under 3D Studio DOS and how they may be implemented under the 3ds max plug-in architecture.

The General Terminology section defines terms used throughout the SDK documentation. These terms also appear as popup help in other parts of the documentation.

The Advanced Topics section presents detailed information about key concepts in using the APIs. The topics presented in this section cross class boundaries, and are instead based on specific implementation issues developers need to understand when designing plug-ins for MAX. Developers getting started with the SDK should take a look through the various Advanced Topics sections to become familiar with what information is available.

There is also a section on commonly-encountered problems and solutions.

## Plug-In Directory Search Mechanism

See Also: [DLL Functions and Class Descriptors](#).

This section discusses the system 3ds max uses to search for plug-in DLLs. This includes path searching and the standard file name extensions required by plug-ins.

## **DLL Path Search**

In the root directory with the 3ds max executable is a file **PLUGIN.INI**. This file contains a one-line entry for each plug-in search location. When 3ds max begins execution it searches each path listed in this file looking for plug-in DLLs. When a plug-in DLL is found it is loaded. This system allows developers to store their plug-ins in whatever sub-directory structure they wish. The format for the **PLUGIN.INI** file is:

### **Descriptive Text=Search Path**

Here's what a sample **PLUGIN.INI** might look like:

**Standard MAX plug-ins=C:\3DSMAX\STDPLUGS**

**Miscellaneous free plug-ins=C:\3DSMAX\MISCPLUG**

From the 3ds max File menu is a choice Configure Paths... The dialog page for Plug Ins lists the paths defined in **PLUGIN.INI**. The user may use the interactive tools in this page to edit **PLUGIN.INI** and add or remove search paths. Manual maintenance shouldn't be required often, however, since a plug-in's install routine can automatically write a string to the **PLUGIN.INI** file that contains all the information required after creating the sub-directory and copying its files into it. Mostly, manual maintenance will be used to create directories for freebie plug-ins, and to de-install directories for plug-ins that are no longer used.

## Standard File Name Extensions

There are certain file name extensions that plug-in DLLs must use. 3ds max uses these standard extensions as part of its search algorithm to determine which files it will attempt to load. Below is a list of the extensions and the plug-in types associated with each one:

- BMI** - Bitmap Manager IO DLLs.
- BMF** - Bitmap Manager Filter Plug-Ins.
- BMS** - Bitmap Manager Storage DLLs.
- DLB** - Shader Plug-Ins.
- DLC** - Controllers.
- DLE** - Scene Export Plug-Ins.
- DLF** - Font Loaders.
- DLH** - Sampler Plug-Ins
- DLI** - Scene Import Plug-Ins.
- DLK** - Filter Kernels (Anti-aliasing filters)
- DLM** - Modifiers.
- DLO** - Procedural Objects.
- DLR** - Renderers.
- DLS** - Object Snap Plug-Ins.
- DLT** - Materials and Textures.
- DLU** - Utility Plug-Ins.
- DLV** - Rendering Effects
- FLT** - Image Filter Plug-Ins.
- GUP** - Global Utility Plug-Ins.

Note that it is acceptable to have several different plug-in types present in a single DLL. As long as the DLL uses one of the file name extensions listed above, 3ds max will load ALL the plug-ins present. For example, if a developer created a system plug-in using both a procedural object and a controller, he or she could place them both in the same DLL. 3ds max would load them all (provided the file name extension matched one of those above).

## Plug-In Help System

Plug-In developers may add on-line help for their applications and make this available to users via the 3ds max Help menu.

From the 3ds max Help menu is a choice Plug-in Help... When selected, this brings up a dialog with a list of installed help systems. The user may select one (by double-clicking or selecting and clicking on Display Help) and that help system's main menu will be activated.

Help files for plug-ins are specified in a separate **[Help]** section of the **PLUGIN.INI** file. The syntax is:

**Description of the helpfile=C:\PATH\TO\FIND\THE\HELPPFILE.HLP**

The description is the string that will appear in the list that pops up when you select Plug-in Help, and the specified .HLP file will be loaded when the user asks to see the help.

For example, the following lines added to **PLUGIN.INI** would add the SDK Help file to the list of choices.

**[Help]**

**SDK Help=C:\MAXSDK\HELP\SDK.HLP**

Note: For 3ds max 2.5 and later, developers may use help files in either .HLP (WinHelp) or .CHM (HTMLHelp) formats.

## Plug-In Configuration System

Developers often need to store configuration files for use with their plug-in applications. There is a standard location defined for these files. This allows a 3ds max user to have all their configuration files stored together and not spread around different hard-to-find locations.

The standard location is the **PlugCFG** directory off the 3ds max root directory. This location may be changed by a user using the Files / Configure Paths... menu command. The developer should retrieve the location of this directory using the **GetDir(APP\_PLUGCFG\_DIR)** method of the **BitmapManager** class or the **Interface** class.

The following samples code from `\MAXSDK\SAMPLES\IO\WSD\WSD.CPP` demonstrates one approach a developer may use to read and write configuration files:

```
#define WSDCONFIGNAME _T("wsd.ini")
#define WSDSECTION _T("Default State")
#define WSDHOSTKEY _T("Hostname")
#define WSDSYSKEY _T("System")
#define WSDDEFAULT _T("accom")
#define WSDDEFSYS _T("ntsc")
// GetCfgFilename()
void BitmapIO_WSD::GetCfgFilename( TCHAR *filename ) {
    _tcscopy(filename,TheManager->GetDir(APP_PLUGCFG_DIR));
    int len = _tcslen(filename);
    if (len) {
        if (_tcscmp(&filename[len-1],_T("\\")))
            _tcscat(filename,_T("\\"));
    }
    _tcscat(filename,WSDCONFIGNAME);
}
```

Note: The **GetPrivateProfileString** function retrieves a string from the specified section in an initialization file. This function is part of the Windows API. See the Windows API on-line help for more details.

```
// ReadCfg()
```

```

int BitmapIO_WSD::ReadCfg() {
    TCHAR filename[MAX_PATH];
    TCHAR system[64];
    GetCfgFilename(filename);
    int res =
    GetPrivateProfileString(WSDSECTION,WSDHOSTKEY,WSDDEFA
        hostname,MAX_PATH,filename);
    if (res) {

    GetPrivateProfileString(WSDSECTION,WSDSYSKEY,WSDDEFSY
        system,64,filename);
    } else {
        _tcscpy(system,WSDDEFSYS);
    }
    if (!_tcscmp(system,WSDDEFSYS)) {
        ntsc = TRUE;
        height = 486;
    } else {
        ntsc = FALSE;
        height = 576;
    }
    return (res);
}

```

Note: The **WritePrivateProfileString** function copies a string into the specified section of the specified initialization file. This function is part of the Windows API. See Windows API on-line help for more details.

```

// WriteCfg()
void BitmapIO_WSD::WriteCfg() {
    TCHAR filename[MAX_PATH];
    TCHAR system[64];
    if (ntsc)
        _tcscpy(system,WSDDEFSYS);
    else
        _tcscpy(system,_T("pal"));
}

```

```
GetCfgFilename(filename);  
WritePrivateProfileString(WSDSECTION,WSDHOSTKEY,hostnam  
WritePrivateProfileString(WSDSECTION,WSDSYSKEY,system,file  
}
```

## Plug-In Architecture Overview

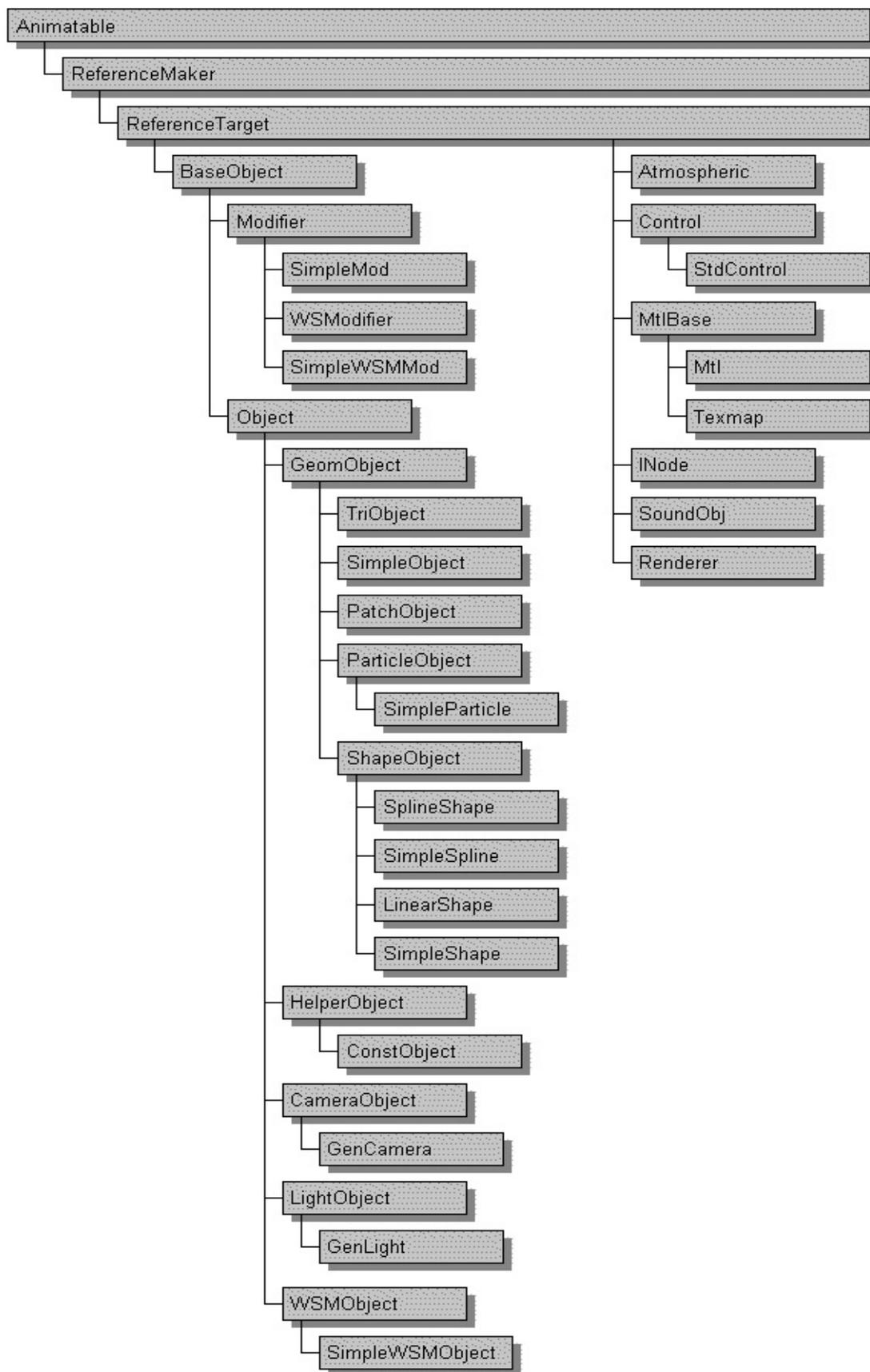
See Also: [Overview of the Principal Classes](#)

The SDK is a set of C++ classes and related routines. Writing a plug-in involves creating objects from the classes and implementing methods to allow communication between the plug-in and the system. In addition to using the provided SDK classes, you can integrate new classes seamlessly into the SDK framework.

Most of the SDK classes inherit from three [= 4\) BSPSPopupOnMouseOver\(event\);">abstract base classes](#). The root class of these three is called Animatable. It defines most of the animation and track view-related methods. Derived from Animatable is ReferenceMaker. This class allows you to make references to other objects. Derived from ReferenceMaker is ReferenceTarget. A reference is a two-way link between objects in the scene. It creates an official record of the dependencies between the ReferenceMaker and the ReferenceTarget. Its primary function is to allow a reference target to inform its dependent reference maker that it has changed in some way.

Classes that don't inherit from Animatable are primarily those that don't deal with animation. For example, the Interface class provides a mechanism for plug-ins to call functions in 3ds max itself. The Interface class is not derived from Animatable.

The following diagram shows the inheritance tree of the principal public classes in the SDK. The base classes (classes that do not have parents) are shown at the top, and the inheritance hierarchy proceeds toward the bottom and to the right.



## **Class Hierarchy**

# Writing Plug-In Applications

See Also: [Main Plug-In Classes](#).

This section presents information common to developing all 3ds max plug-in applications. It discusses the sample files, the plug-in directory search mechanism, the plug-in help system, the configuration file system, the standard DLL functions, and creating thread-safe plug-ins.

Included with the SDK are sample programs that provide many examples of how to write plug-in applications. These sample programs are found in the sub-directories of **\MAXSDK\SAMPLE**. You'll find it very instructive to examine the source code of these examples. The section on [Building the Sample Files](#) has more detailed information on the sample programs available.

3ds max plug-ins are implemented as dynamic link libraries (DLLs). DLLs are object code libraries that let multiple programs share code, data, and resources. Dynamic linking allows an executable module to include only the information needed at run time to locate the executable code for a DLL function. This type of linking differs from the more familiar static linking, which requires a copy of a library function's executable code in the executable module of each application using it.

The development tool for creating these DLLs is Microsoft Visual C++. For a description of how to create a new project file see the Advanced Topics section [Creating a New Plug-In Project](#).

Developers may store their plug-in DLLs in any location they wish. Developers need to let 3ds max know where to search for the DLLs. How this is done is discussed in the section [Plug-In Directory Search Mechanism](#).

Plug-In developers may add on-line help for their applications and make this available to users via the 3ds max Help menu. See the section [Plug-In Help System](#) for more details.

There is a standard location for developers to save any configuration files required by their plug-in application. These may be .INI files, binary configuration files or whatever is needed. See the section [Plug-In Configuration System](#) for details.

## Standard DLL Functions

There are a standard set of functions that ALL plug-in DLLs must implement. These functions are:

**DLLMain()**

**LibDescription()**

**LibNumberClasses()**

**LibClassDesc()**

**LibVersion()**

These functions allow 3ds max to access, work with, and maintain the plug-ins inside the DLL. To review the full details of these functions, see the Advanced Topics section [DLL Functions and Class Descriptors](#)

## **Reentrant and Thread Safe Plug-Ins**

3ds max plug-ins must be 'Reentrant' and 'Thread Safe'. See the Advanced Topics section [Thread Safe Plug-Ins](#) for more information.

## Building the Sample Files

See Also: [Main Plug-In Classes](#).

## Overview

The SDK provides sample code for many of the plug-in types. This section discusses the procedure to build and run the sample plug-ins included in the SDK. All the sample code is located in the sub-directories of **\MAXSDK\SAMPLES**. A MAK file is provided to build each sample.

Each MAK file has several configurations within it. There is a Release configuration, and a Hybrid configuration. In the *special* developer SDK available to registered developers ([See Debugging](#)) there is an extra Debug configuration. Developers may use any of these. Developers using the standard SDK should use the Hybrid configuration for source level debugging and the Release configuration for distribution code. The plug-ins that ship with 3ds max were created using the Release configuration.

These configurations differ based on their optimization of the code, their inclusion of debug symbols, and their use of runtime libraries.

The Release configuration creates optimized code while the other configurations do not.

Both the Hybrid and the Debug configurations contain debug symbols useful in source level debugging. The Release configuration does not include debug symbols.

The Release and Hybrid configurations both use the same runtime libraries. These are called **Multithreaded DLL** in VC++. The Debug configuration uses a different one, the **Debug Multithreaded DLL** library. The difference is the Debug configuration uses a different heap to track memory. The non debug libraries use the same heap as regular release code. The issue is that heaps cannot be mixed and matched. This is why a special Debug version of 3ds max is required to use the special Debug configuration.

Developers using the standard SDK should use the Hybrid configuration for debugging and the Release configuration for distribution code.

## Building the Samples

The MAK files assume the SDK is installed in the \MAXSDK directory off the root of the drive.

The MAK files also assume that the directory to store the 3ds max plug-ins is \MAXSDK\PLUGIN. If this is not where you wish to put them you will need to change the output file directory. Load the MAK file by choosing **File / Open Workspace...** changing the **List File of Type...** to **\*.MAK** and selecting the MAK file.

**1** From the **Build** pull down menu choose **Settings...**

**2** From the Project Settings dialog box choose the **Link** Tab.

**3** Under **Output File Name:** Enter the path to the desired plug-in directory and the output file name.

To build one of the sample projects:

**1** Load the MAK file into VC++.

**2** From the **Build** menu choose **Rebuild All**. This will compile and link the plug-ins in the MAK file. Note that this process will overwrite any plug-in DLLs with the same name in the destination directory.

## Running The Sample Programs

**Important Note:** The SDK plug-ins are the same as the standard 3ds max plug-ins. If you want to run the samples you should copy the plug-in from the `\MAXSDK\PLUGIN` directory to your `\3DSMAX\STDPLUGS` directory. This will replace the standard 3ds max plug-in with the SDK version. If you try to load them both (i.e. put them in separate directories and tell 3ds max to load plug-ins from each of these directories) you'll have a Class\_ID conflict. 3ds max will not load two plug-ins that have the same Class\_ID. It will only load the first one and put up a warning about the second one.

Another option is to change the Class\_ID used in the sample code before it is compiled. 3ds max may then load both plug-ins. For more information on Class\_IDs see the section [DLL Functions and Class Descriptors](#).

## **Ready To Build Plug-In Projects**

The 3ds max 4.0 SDK comes provided with an SDK appwizard that can be used with Microsoft Visual C++. This appwizard allows you to quickly generate skeleton source code for a variety of different plugin types and essentially replaced the previous R3\_SKELETON projects included in the previous release of the software.

The generated code from the appwizard provides a quick way to start building plugins. Each project follows a standard structure with regard to headers and code files. Developers should find it easy to take this generated code and can start creating plugins without manually setting up the entire project from scratch.

# Must Read Sections for All Developers

See Also: [Must Read Sections by Plug-In Type](#), [Advanced Topics](#).

The following sections in the help file should be read by all developers. These are fundamental issues every programmer needs to understand to write plug-ins for MAX.

## [DLL/Lib Functions and Class Descriptors](#)

All 3ds max Plug-Ins must implement these DLL and Library functions.

## [Memory Allocation](#)

This section discusses how plug-ins are allocated and deallocated in memory.

## [Intervals](#)

Intervals are an object in 3ds max that describe a range of time. They are a key concept in understanding animation in MAX.

## [Geometry Pipeline System](#)

The Geometry Pipeline is a system used by all plug-ins that deal with geometry.

## [Sub-Anims](#)

The hierarchy of animatable items that appears in Track View is referred to as the sub-anim hierarchy. This section documents how a plug-in provides access to it's sub-anims.

## [References](#)

The concept of a reference is what allows parts of 3ds max that relate to one another to communicate to manage the relationship. This section presents information on how this works.

## [Nodes](#)

Nodes are the items in the 3ds max that have a one to one correspondence with objects in the scene. This section provides information on nodes.

## [Custom User Interface Controls](#)

This section describes the custom controls developers may use in their user interface design.

## [Parameter Blocks](#)

Parameter Blocks are the objects in 3ds max that help to manage the variables that a plug-in uses to store animated values.

## [Parameter Maps](#)

Parameter Maps are used to simplify the programming effort required to manage the user interface of a plug-ins parameters.

### [Matrix Representations of 3D Transformations](#)

Matrices are used in 3ds max to represent various transformation. This section explain how they work.

### [Rotation Concepts](#)

This section provides information on some of the classes that deal with rotation.

# Must Read Sections by Plug-In Type

See Also: [Must Read Section for All Developers](#), [Plug-In Types Overview](#).

The following sections in the help file are those that should be read by developers creating a certain plug-in type. This is simply an aide to point you at the relevent sections that must be understood to create a specific kind of plug-in.

## **Atmospheric**

[Working with Bitmaps](#)

[Working with Lights](#)

[Working with Materials](#)

[References](#)

## **Controllers**

[Keyframe and Procedural Controller Data Access](#)

[Node and Object Offset Transformations](#)

## **File Import / File Export**

[Geometry Pipeline System](#)

[Working with Controllers](#)

[Keyframe and Procedural Controller Data Access](#)

[Node and Object Offset Transformations](#)

[Object Creation Methods](#)

[Computing Face and Vertex Normals](#)

[References](#)

[Nodes](#)

## **Image Loading and Saving**

[Working with Bitmaps](#)

## **Image Processing (Filters)**

[Working with Bitmaps](#)

## **Materials**

[Working with Bitmaps](#)

[Working with Materials](#)

[References](#)

[Sub-Anims](#)

## **Object Modifiers**

[Geometry Pipeline System](#)

[Object Modification](#)

[References](#)

[Sub-Anims](#)

## **Procedural Objects**

[Geometry Pipeline System](#)

[Object Modification](#)

[References](#)

[Sub-Anims](#)

## **Renderer**

[Geometry Pipeline System](#)

[Working with Bitmaps](#)

[Working with Lights](#)

[Working with Materials](#)

[References](#)

[Sub-Anims](#)

## **Space Warps**

[Space Warps Plug-Ins](#)

[Geometry Pipeline System](#)

[Object Modification](#)

[References](#)

[Sub-Anims](#)

## **System**

[Geometry Pipeline System](#)

[Working with Controllers](#)

[References](#)

[Sub-Anims](#)

## **Texture Maps**

[Working with Bitmaps](#)

[Working with Materials](#)

[References](#)

[Sub-Anims](#)

## **Track View Utilities**

[Track View](#)

# Fundamental Concepts of the MAX SDK

See Also: [Must Read Section for All Developers](#), [Advanced Topics](#).

## **Overview**

This section provides an overview of the fundamental concepts involved in using the SDK. It provides an explanation of many of the basic issues that developers deal with while creating plug-in applications.

The SDK provides tremendous control over almost all aspects of MAX. For this reason, there is a lot of information. However, broken down into smaller parts, things aren't complex. This section will discuss most of the basic issues in simple terms, and has hyperlinks to the sections in the SDK that contain more detailed information.

## The Object Oriented Philosophy of MAX

3ds max can be thought of as an operating system for 3D graphics. What a user is doing when running MAX, for the most part, is using plug-ins. Almost all areas of 3ds max are open to developers, and the 3ds max programmers use the same SDK and tools as third-party developers.

Instances of the plug-in classes developer create are called **Objects**. The way that they can be built independent of one another, yet work together to function as a system, is what makes the 3ds max plug-in architecture so powerful. The biggest impact this object oriented design has on developers is that one can develop plug-ins independently, without having to know or worry about how the other plug-ins are going about their tasks. Developers do have to understand the 3ds max Reference Architecture, however. The Reference Architecture (discussed later in this topic) is the system that allows all the plug-ins to communicate.

## **Plug-In Types**

All of the following areas of 3ds max are implemented as plug-ins: shapes and splines, patch objects, particle systems, modifiers, space warps, lights, cameras, bitmaps, texture maps, materials, image filters and compositors, animation controllers, file import/export utilities, atmospheric effects, renderers, and sound support.

For a list of all the plug-in types that can be developed with the SDK see the topic [Plug-In Types Overview](#).

## MAX Plug-Ins and the Tools for Creating Them

This section describes the type of executable used for 3ds max plug-ins and the tools used to create them.

Plug-Ins for 3ds max are developed with Microsoft Visual C++. Plug-ins are implemented as Windows Dynamic Link Libraries (DLLs). A dynamic-link library (DLL) is an executable file that acts as a shared library of functions. Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a DLL, which contains one or more functions that are compiled, linked, and stored separately from the processes that use them.

For step by step information on how to set up a new plug-in project using Visual C++ see the section [Creating a New Plug-In Project](#) (this includes information on the AppWizard that's available to quickly create projects). For information on debugging plug-ins using the Visual C++ debugger see the topic [Debugging](#). Once your plug-in is working you can optimize it for speed. See the section on [Profiling Plug-in Performance](#) for this information.

An additional tool that is helpful in searching for information on specific methods in the C++ source code is `\MAXSDK\HELP\SDKLINK.ZIP`. This tool launches the SDK online help and jumps directly to the reference information on the selected method. To use this program unzip it and review the installation and use instructions in the **README.DOC** file.

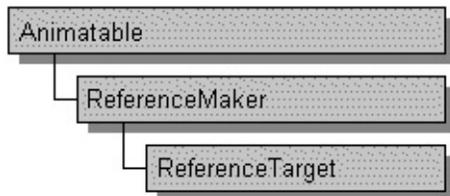
There is also a file, **USERTYPE.DAT**, which can be used to enable syntax highlighting of user defined keywords in VC++ 6.0 IDE. This can make browsing and writing code that much more enjoyable. This file lets you get all the 3ds max methods and definitions highlighted in a custom color inside of Visual C++ (like the colored C/C++ keyword you have now).

To put this tool in place, copy the file `\MAXSDK\HELP\USERTYPE.DAT` into the `\DevStudio6\Common\MSDev98\Bin` directory. After restarting VC++, you can then change the color of 3ds max keywords by modifying the color under Tools/Options/Format/Colors/User Defined Keywords. It is recommended that you choose a different color than the color assigned to Tools/Options/Format/Colors/Keywords, which are specific to C++. Having two different color patterns aids in your understanding of the code.

## The C++ Class Hierarchy

This section describes the hierarchy of C++ classes used to develop plug-ins. C++ classes are used by 3ds max because of their efficiency. When a plug-in is linked directly into the parent program, as it is with Windows DLLs, program flow can proceed directly from 3ds max into the plug-in. The plug-in and 3ds max share memory and can directly access one another's data.

The majority of 3ds max plug-ins are related to modeling, animation and rendering. These plug-ins are generally derived from a series of base classes that provide common characteristics such as the ability to appear in Track View and the ability of the plug-ins to communicate with one another as they are modified and changed.



### Three Base Classes of the MAX Class Hierarchy

At the base level of this hierarchy is the animation related class named **Animatable**. The primary methods of this class relate to appearing in Track View and dealing with memory management. Derived from **Animatable** are two classes used in the communication between different parts of MAX.

These are **ReferenceMaker** and **ReferenceTarget**. These two classes are used to facilitate communication between dependent objects in MAX. This inter-object communication is described later in this topic.

There are other plug-ins that are not derived from these classes. Generally, these are items that don't directly relate to animation. For instance, import / export plug-ins simply get data in and out of MAX. They don't appear in Track View, and function independently of other plug-ins.

For a more complete look at the class hierarchy see the section [Plug-In Architecture Overview](#). For a summary of the main methods of these classes see [Overview of the Principal Classes](#).

## How Plug-Ins Interact with MAX

This section discusses the ways 3ds max calls functions provided by the plug-in, and how plug-ins can call functions provided by MAX.

Plug-ins are derived from a base class provided by MAX. For example, if you wanted to develop a geometric object plug-in you might derive your class from class **SimpleObject** as shown below.

```
class MyGeomObject : public SimpleObject {  
// ...  
};
```

The 3ds max header files include the definition of the **SimpleObject** class. It is your responsibility as the programmer to provide implementations of some of the required methods of **SimpleObject** in your plug-in. Said another way, your plug-in implements (provides the code for) certain methods of the base class. 3ds max can then call these methods on your plug-in. For example, **SimpleObject** has a method **BuildMesh()**. This method is called by 3ds max when it needs to get a triangle mesh representation of your geometric object. In the code for your implementation of **SimpleObject::BuildMesh()** you generate the mesh you need. When 3ds max needs this mesh representation it calls your **BuildMesh()** method.

In other cases, methods are provided by the base class (by MAX). That is, the base class itself provides the implementation. These methods may be called by the plug-in. These methods are inherited from the base class from which the plug-in is derived. For example, a geometric object primitive may be derived from class **GeomObject**. This geometric primitive can call **NotifyDependents()**. This is a method of **ReferenceTarget** that the primitive inherited via the base class hierarchy (**GeomObject->Object->BaseObject->ReferenceTarget**).

In the SDK documentation, methods that the plug-in implements are labeled 'Implemented by the Plug-in'. Those that 3ds max provides the implementation for are labeled 'Implemented by the System'.

There are other ways that plug-ins can call methods provided by MAX. This is done through a pointer to an **Interface Class**. An interface class is simply a class without data members and only pure virtual methods. Classes such as these are essentially just a table of function pointers. Interface classes are used quite often

in 3ds max programming. The most general interface class is named, logically enough, **Interface**. A pointer to this class is passed into a great many methods and plug-ins use it to perform common operations provided by MAX. For example, when a modifier plug-in wants to show its user interface in the command panel it calls a method of **Interface** named **AddRollupPage()** to take care of it. Or when a plug-in wants to retrieve the current time as specified by the frame slider in MAX, the **Interface** method **GetTime()** is called. See the Advanced Topics section [Overview of Interface Classes](#) for more details.

## How Plug-Ins Interact with Each Other -- The Reference Architecture

In the object oriented paradigm of MAX, where independent plug-in objects work together to facilitate the overall functioning of the system, some form of inter-object communication is required. The Reference Architecture is the mechanism 3ds max uses to handle this communication. Consider the following example to understand why it is needed.

One of the 3ds max animation controller plug-ins is called the Look At Controller. This is a transform controller, meaning it controls the position, rotation and scale of an item in the scene. The Look At controller rotates the item it's assigned to so that it is always pointed towards (or "looking at") another object. For example, say you create a Free Camera and assign the Look At controller as the camera's transform controller. You then pick a Box in the scene as the target of the Look At controller. This will cause the Free Camera to rotate such that its direction of view is pointing right at the Box. Whenever the Box is moved, the controller will readjust itself so that the camera remains pointing towards it. Also, if the Box is deleted from the scene, the camera will recognize this and simply remain stationary.

Now consider that each of these items (the Camera, the Look At Controller, and the Box) are all plug-ins. And each is written without any specific knowledge about the other objects. How then can these objects communicate so they are aware of one another? For example, how does the Camera know that its direction of view changes when the Box moves? Or how does the Look At controller know that the Box has been deleted from the scene? The answers are that these objects communicate via what are called **References**.

A Reference is a way to associate two items. One item is considered dependent upon the other. In the example above, the Look At Controller is dependent on the Box. And the Camera is dependent on the Look At Controller. That is to say, if the Box changes in some way it needs to let other objects that might depend on it know it has changed. The controller, via the Reference, gets notified that the Box has changed. And the controller needs to do the same for those items which depend on it. When the controller has changed, it lets those items which may depend on it know. Thus the camera gets notified.

An object sets up this record of dependency by creating a Reference. When an object makes a reference, it is called a Reference Maker. The object that is referenced (that it depends on) is called a Reference Target. In the example above, the Camera is a Reference Maker. It references the Look At Controller.

The Look At Controller is the Reference Target of the Camera. The Look At Controller is also a Reference Maker -- it references the Box. The Box is the Reference Target of the controller. The Box, in fact, has its own references, for example, its transform controller.

The inter-dependencies created through References exist all over MAX. In a simple scene, dozens of references exist (many that 3ds max is using internally). In a complex scene, hundreds may exist.

For additional details on how References are used for inter-object communication see the Advanced Topics section [References](#). Also see the sub-topic in that section called **Viewing Reference Messages**. It discusses a utility plug-in called the Reference Watcher. This plug-in may be used to help understand the reference structure of a chosen item and to monitor the reference messages it sends. This provides a quick, visual way to examine the use of references in MAX.

## **The Parametric Nature of MAX Objects and the Impact on Plug-Ins**

The geometric objects provided by 3ds max are parametric objects. This means they are defined by their user interface parameters, rather than directly by the vertices and faces which make up a mesh surface. For example, a Sphere primitive is defined by its Radius, Segments count, Hemisphere setting, etc. These are what are stored in memory, and saved by 3ds max when the sphere object is written to disk. A triangle mesh representation of the Sphere is required for rendering, but it is computed 'on the fly' from the parameters. Contrast this with the older 3D Studio for DOS. Its objects were not parametric and existed only as the vertices and faces that described the surface. If a user wanted to go back and modify the sphere's segment count she would have to delete and re-create a new sphere object with the setting changed. In 3ds max one can simply change the segment count at any time and the surface is regenerated.

Geometric objects must eventually be converted to triangle meshes for rendering, however. The 3ds max renderer works with mesh objects **only**, so primitives, booleans, loft objects, particles, patches and NURBS surfaces all are converted to triangle meshes.

In addition to procedural objects, 3ds max provides an extensive collection of Modifiers. Many of these are applied to objects to provide some form of modification, deformation, manipulation or editing of the underlying object. Users can also apply Space Warp plug-ins to alter geometry based on world space positions. The implication of all this for plug-in developers is that they need to understand the conversion process from parametric object to modified triangle mesh.

The **Geometry Pipeline System** of 3ds max is used to handle this processing and conversion. The geometry pipeline is the system used to allow parametric objects to be operated on by modifiers, with the eventual generation of a triangle mesh suitable for rendering and display in the viewports. For the full details see the Advanced Topic section [Geometry Pipeline System](#). For information regarding how procedural objects and modifiers work together to facilitate the modification see the section [Object Modification](#).

Plug-in developers may find helpful to study the code for the Collapse Utility. This utility lets a user collapse the Stack of one or more selected objects into an Editable Mesh and, optionally, perform a Boolean operation on them at the same time. See the code in

**\MAXSDK\SAMPLES\UTILITIES\COLLAPSE.CPP.**

## The Scene -- Nodes

The 3ds max Scene, that is the objects, lights, cameras, space warps, particles, etc., are each associated with a **Node**. There is a one to one correspondence between each item in the scene and a node. A node is simply the item that manages the information needed to allow its associated object to exist in the scene. These are things such as the transform controller, the material used, data about parent-child hierarchies, and grouping information.

The transform controller of the node controls the position, rotation and scale of the object in the scene. The material assigned to the node is used by the renderer to give the object its surface characteristics such as color, texture, bumpiness. The hierarchy information is used to handle the linked / unlined state of objects for forward and inverse kinematics.

For more detailed information see the Advanced Topics section [Nodes](#), [Working With Materials and Textures](#), and [Working with Controllers](#).

## Time and Intervals

Developers dealing with animation need to specify certain points in time, and also certain periods of time. This section discusses each of these.

The basic unit of time in 3ds max and the SDK is equal to 1/4800th of a second. This length of time is a data type called a **TimeValue**. TimeValues are used throughout the class methods in the SDK when a specific instant in time needs to be specified. For example, if your plug-in needed to know the current system time (position of the 3ds max frame slider) it would call the methods **Interface::GetTime()**. This methods returns a **TimeValue** that indicates the current time (in 1/4800ths of a second).

In MAX, intervals are commonly used to define the time period in which an object is unchanging. To be as fast as possible 3ds max avoids re-computing objects whenever it can. If an animated plug-in can specify how long a period it is unchanged 3ds max can know it doesn't have to re-evaluate the object over that time. For example, 3ds max needs to know over what length of time the Modification done by a modifier is valid for. Consider a Bend modifier where the user has animated the bend angle over time (say un-bent from frame 0 to frame 50, and then bent 45 degrees at frame 100.) So that 3ds max doesn't need to re-evaluate the modifier every frame (after all, its not bending from frame 0 to 50) the modifier provides 3ds max with a **Validity Interval** that tells it how long the deformation is valid for. This validity interval is relative to a certain time. For instance, say 3ds max asks the modifier how long the modification is valid for at frame 0. The modification is valid from frame 0 to frame 50. If 3ds max asked how long the modification was valid for after frame 50, say at frame 75, then the answer is only for a single instant of time. That's because the modifier is animated from frame 50 to 100. At each frame it is different (bent more towards the final 45 degrees). Therefore the validity interval is only a single point in time (or **TimeValue**).

Intervals of time are specified in the SDK using an **Interval** object. This object has a start and end time and various methods to work with the interval. There are numerous methods that a plug-in implements to specify validity intervals in MAX. The modifier example above would do so in the method **Modifier::LocalValidity()**. For more information on Time in the SDK see the Advanced Topics section [Time](#). More information on intervals can be found in the topic [Intervals](#).

## Animation Controllers

All animation in 3ds max is managed by a plug-in type called an animation controller (or controller for short). Some controllers are keyframe based. This means the user sets the value of the controller at important (key) points in time and the controller provides all the values at other times by interpolating between the key values. Others are procedural, meaning the value is computed at a certain time based on a pre-programmed effect (for instance, the Noise controller uses a fractal function to compute the value).

There are six basic controller types based on the data type they control: floating point values (**float**), three float values (**Point3**), Position (**Matrix3**), Rotation (**Quat**), Scale (**ScaleValue**), and Transform (**Matrix3**). Using the SDK developers can create new controller types, or work with the properties of existing types. See the topic [Working With Controllers](#) for more information.

Many plug-ins use animated parameters, for instance the animated radius parameter of a procedural Sphere. 3ds max provides tools for developers wanting to use animated parameters that make them very easy to manage. Two of these are **Parameter Blocks** and **Parameter Maps**. Parameter Blocks provide a mechanism for storing values for a plug-ins parameters and managing the controller which handle the interpolation or generation of values. Parameter Maps are used to associated a control in the user interface of the plug-in with an animated value. See the Advanced Topics sections [Parameter Blocks and Maps in Release 3 and Later](#), [Parameter Maps](#) and [Parameter Blocks](#) for details.

## User Interface Issues

The user interacts with 3ds max mainly via the command panel rollups/dialogs, the mouse (or tablet), the keyboard, and the 3D viewports. The SDK provides developers with the ability to control each of these areas of user interaction.

Plug-Ins generally provide their user interface controls in one of three ways: In one or more of the Command Panel branches, in rollups which appear in the Materials Editor, Environment, or Render dialogs, or as modal or modeless floating dialogs. Many of the buttons, spinners, edit fields, etc. that appear in these dialogs/rollups are from the set of the 3ds max provided **Custom Controls**. These are used to make the UI of plug-ins consistent in appearance from one to another. For more information on these controls and how to work with them see [Custom User Interface Controls](#).

In addition to the custom controls, the mouse is used to interact with MAX. The way this type of interaction is handled in the SDK is through what are called **Command Modes**. For more information see the section [Command Modes and Mouse Procs](#).

Plug-ins can also make use of the keyboard to speed user interaction with their plug-in. This is done by registering what are called **Keyboard Accelerators**. These allow the user to execute some of their plug-in's functions by assigning them to keystrokes. For information on how this is done see the Advanced Topics section [Keyboard Accelerators and Dialog Messages](#).

Finally, plug-ins often need to draw in the 3D viewports to show themselves or their 'gizmos'. This is accomplished using the methods provided by the interactive renderer class GraphicsWindow. For information on the way to work with this renderer see [Interactive Renderer: Graphics Window](#).

## Functions Every Plug-In Must Provide

This section describes a set of five functions required by every plug-in DLL. It also outlines an object developers need to provide for each plug-in class.

Every plug-in developer needs to provide the code for the DllMain function and four Lib functions. Each is described briefly below:

### Function Called to Handle Initialization

**DllMain()** -- This function is the hook used by Windows to initialize the DLL. 3ds max plug-ins use it to initialize the common controls library and MAX's custom controls.

### Functions Used by MAX to Inventory and Categorize the Plug-ins in a DLL

These four functions describe the number and properties of the plug-ins provided by the DLL.

**LibNumberClasses()** -- This function returns the number of plug-in classes contained in the DLL.

**LibVersion()** -- This is a function that allows the system to deal with obsolete versions of plug-in DLLs.

**LibDescription()** -- This function returns a text string to present to the user if the DLL is unavailable.

**LibClassDesc()** -- This function returns a pointer to an object called a **Class Descriptor** for each plug-in class in the DLL. This Class Descriptor object describes the properties of each plug-in class and a way to allocate an instance of the class in memory. The next section describes these Class Descriptors.

### Creating and Classifying Instances of Plug-In Classes

The **Class Descriptor** described above is an object derived from class **ClassDesc**. It has several important purposes. The two main ones relate to classifying the type of object the plug-in is, and allocating the memory for instances of the plug-in objects. The full details of Class Descriptors are described elsewhere but three important methods are described below.

Plug-ins of all types create 3ds max system objects. These are not the 3D objects that appear in a scene, but objects in the C++ sense. There are two IDs associated with each plug-in system object. These are the Super Class ID and the Class ID. The Super Class ID specifies what super-class of 3ds max

the plug-in class is a sub-class of. The Class ID differentiates between the various plug-ins for a super-class. For example, a Sphere object has a Class ID, **unique** to it. This is simply an ID generated using a program provided by the SDK that uniquely identifies the plug-in class (it is in `\MAXSDK\HELP\GENCID.EXE` -- [Click Here to Try It](#)). Its Super Class ID, which is shared by all geometric primitives, is **GEOMOBJECT\_CLASS\_ID**. The Super Class IDs are defined by MAX, and each plug-in falls into one of these predefined categories. For instance, all Texture Map plug-ins share the same Super Class ID of **TEXMAP\_CLASS\_ID**. Each individual texture map plug-in has its own unique Class ID however. Thus, the Super Class ID defines which kind of object it is, the Class ID uniquely identifies a specific plug-in class.

The Class Descriptor method **ClassDesc::SuperClassID()** returns the Super Class ID. The method **ClassDesc::ClassID()** returns the Class ID. Another function of the class descriptor is to allocate instances of the plug-in class. For example, when a 3ds max file is loaded which contains a procedural Sphere, 3ds max needs a way to create an instance of the Sphere plug-in. It calls a method of the Sphere's class descriptor (**ClassDesc::Create()**). See the next section for more on this method.

The functions outlined above are described in greater detail in the Advanced Topics section [DLL/Lib Functions and Class Descriptors](#).

### **The Memory Allocation and Deallocation of Plug-In Classes**

As discussed above, instances of plug-in classes are created by calling Class Descriptor's Create method (**ClassDesc::Create()**). This applies to all plug-in classes in the SDK.

The way this memory is freed varies based on the class the plug-in is derived from. Usually it is done by calling a method named **DeleteThis()** on the plug-in. In the classes that are part of the **Animatable** class hierarchy this is a method of **Animatable**. So for these plug-ins, the memory is allocated by **ClassDesc::Create()** and deallocated by **Animatable::DeleteThis()**.

For other plug-ins that are not derived from **Animatable**, the plug-in class itself may provide a **DeleteThis()** method. For example, the **Bitmap** class is not derived from **Animatable**. It however provides its own **DeleteThis()** method to deallocate the memory created by **ClassDesc::Create()**.

In other cases, memory is freed automatically by 3ds max itself when it is done with the plug-in. For example, the **ImageFilter** class used to create image processing effects for use in Video Post has its memory freed by 3ds max itself (by using the **delete** operator).

See the Advanced Topics section [Memory Allocation](#) for more details.

## Summary

This topic has provided a look at some of the fundamental issues developers need to understand to create 3ds max plug-ins. The Advanced Topics sections in the SDK provide detailed information about many different aspects of the API. To review the relevant parts of the SDK that relate to a certain plug-in type see the section [Must Read Sections by Plug-In Type](#).

# Moving from IPAS to MAX Plug-In Development

See Also: [Plug-In Types Overview](#), [Plug-In Architecture Overview](#).

## **Overview**

This topic is provided as an aide to IPAS programmers making the transition to 3ds max plug-in development. This section discusses the similarities and differences in the IPAS and 3ds max programming architectures. The changes in development environment, plug-in/system control flow, user interface, and graphics programming support are reviewed. This section also discusses the six types of IPAS routines possible under 3D Studio DOS and how they may be implemented under the 3ds max plug-in architecture.

## **Development Environment**

IPAS routines were developed in 3D Studio DOS using the C programming language with the MetaWare or Watcom compilers. In MAX, development is done using the C++ programming language with Microsoft Visual C++ 5.0 under Windows NT. The 3ds max architecture is an object-oriented system that provides a comprehensive set of classes that developers can combine and extend to create seamlessly integrated plug-in applications. See the section [Writing Plug-In Applications](#) for more information on the basics of 3ds max development.

## **Control Flow**

IPAS plug-ins would completely take over control of 3D Studio DOS. When the IPAS routine started it had exclusive control and maintained this control until the user exited the routine. This forced the user to leave the familiar program interface and work with a separate and different interface with each IPAS routine. This is not the case in 3ds max. Most 3ds max plug-ins are modeless in operation. This means their operation comes in and out of focus as the system requires processing on their part. The user is free to operate the remainder of the standard 3ds max interface even as the plug-in has its user interface up. In this way the user never leaves their familiar working environment.

## User Interface

3ds max provides a set of custom controls that provide a consistent user interface between the plug-in and the system. This provides a level of familiarity for users who often use many different plug-ins. See the Advanced Topics section [Custom Controls](#) for more details on user interface implementation in MAX.

IPAS provided a set of graphic functions that let developers access the screen directly and use standard dialogs. These were the GFX calls such as **gfx\_xorline**, **gfx\_c\_blitput**, and **gfx\_file\_selector**. Most of these functions were low level (get/put pixel, draw line, etc.), but there are also a number of higher-level interface routines, such as standard 3D Studio file selectors, material selectors and other commonly used dialogs. These capabilities are provided by 3ds max as well. Review the section [Interactive Renderer: Graphics Window](#) for details on the low level graphics routines supplied by MAX. For higher level interface routines such as access to standard 3ds max dialog boxes see the methods of the [Interface class](#). The Windows API is also often employed in graphics work in MAX.

IPAS also provided routines to access the mouse and change the cursor form. 3ds max provides mouse and cursor control using the **CommandMode** class. See the section [Command Modes and Mouse Procs](#) for more details on working with the mouse.

There were also GFX routines to access 3D Studio bitmap services (load / save bitmaps, re-size bitmaps, color-cut bitmaps, etc.). 3ds max provides a comprehensive set of bitmap management tools. For details see the section [Working with Bitmaps](#).

Many interfaces built in IPAS were created using the 3DE utility. This simplified the creation of dialog boxes and custom user interfaces. 3ds max uses the Visual C++ tools for creating its dialog boxes. These tools are more comprehensive and robust than 3DE. The section on [Custom Controls](#) provides an general introduction to this tool although you will want to consult the VC++ on-line help for more detail.

## Graphics Programming Functions

IPAS routines could make use of some SDK-supplied include files that provided utility functions such as linear algebra, quaternion algebra, and other common graphics programming functions. The 3ds max architecture provides a very comprehensive set of these tools as well. These standard classes are used throughout 3ds max programming. The following list of classes all provide various graphic / geometric programming support: [Class Point2](#), [Class Point3](#), [Class Matrix3](#), [Class Quat](#), [Class Color](#).

## **IPAS Plug-In Types as MAX Plug-Ins**

The following sections discuss how each of the six IPAS types may be implemented as a 3ds max plug-in. In many cases there is not an exact one-to-one correspondence due to the differences in the two architectures. On the whole however, IPAS developers will find the 3ds max APIs much more powerful, comprehensive and better integrated with the core program.

## **IXP - Image Processing Modules**

IXPs were used to create new images or modify existing ones in the video post data stream. They were also used to create special effects such as image blur, star filters and glows. In 3ds max these types of effects are achieved using Filter plug-ins. These plug-ins are derived from class [ImageFilter](#). Filters may be used as image modifiers, image compositors or as transition effects. The 3ds max capabilities for image processing plug-ins are much greater than their IPAS counterparts. Sample code may be found in the sub directories of **\MAXSDK\SAMPLES\POSTFILTERS** for example **ADD.CPP**.

## **PXP - Procedural Objects Modules**

This type of IPAS routine was usually used to create an object from an algorithm or to modify an existing object. In 3ds max this type of plug-in translates into a procedural object, or an object modifier, or a utility plug-in.

In 3ds max a procedural object is best suited to create an object defined by an algorithm. Geometric procedural objects are derived from [SimpleObject](#), or [GeomObject](#). Sample code for this plug-in type may be found in

**\\MAXSDK\SAMPLES\OBJECTS\SPHERE.CPP.**

To modify an existing object, the object space modifier plug-in type is appropriate. Sample code for an object space modifier may be found in

**\\MAXSDK\SAMPLES\MODIFIERS\BEND.CPP.**

Certain utility IPAS routines were created as PXPs. 3ds max also supports these utility plug-ins. Utility plug-ins are derived from [Class UtilityObj](#).

Sample code for this plug-in type may be found in

**\\MAXSDK\SAMPLES\UTILITIES\ASCIOUT.CPP.**

## **AXP - Animated Procedural Objects**

These plug-ins were used to create hard-to-animate objects such as particle systems and real world data. These procedures animated at render-time on a frame by frame basis.

Certain AXP types could be implemented in 3ds max as procedural objects (particles systems), atmospheric plug-ins (some types of smoke, vapor, and fog), or space warps.

Particle systems may be derived from classes [ParticleObject](#), and [SimpleParticle](#). Sample code may be found in **\MAXSDK\SAMPLES\OBJECTS\RAIN.CPP**. Particle-system type effects may also be created using the [Atmospheric](#) class.

Geometric procedural objects are derived from [SimpleObject](#), or [GeomObject](#).

Space warp plug-ins may be used to modify existing geometry in world space. These plug-ins are derived from [Modifier](#) or [WSMModifier](#).

## **SXP - Solid Texture Procedures**

The IPAS SXP routines were 3D functions that could be assigned as maps in the 3D Studio Material Editor. 3ds max provides similar but more-extensive capabilities for both 2D and 3D procedural textures, compositors and color modifiers. 3ds max also provides APIs to develop complete plug-in materials. The sample code for these plug-ins can be found in the directory **\MAXSDK\SAMPLES\MATERIALS.**

## **KXP - Keyframe Procedures**

These IPAS routines were typically used to create or modify keyframe data of objects in the keyframer. In MAX, controller plug-ins are used to manage animation. See classes [Control](#) and [StdControl](#). Sample controller code is available in

**\MAXSDK\SAMPLES\HOWTO\PCONTROL\PCONTROL.CPP.**

There is also an interface into the Linear, TCB, or Bezier key frame controllers that allows a developer to add, delete, retrieve and store the keys of the controller. See [Class IKeyControl](#).

## **BXP - Bitmap Loading / Saving Procedures**

BXP plug-ins were used to provide extended image file loading and saving support. These image loader / saver plug-ins are called IO modules in MAX. These plug-ins are derived from class [BitmapIO](#). Sample code for these plug-in may be found in the sub-directories of `\MAXSDK\SAMPLES\IO`, for example `\MAXSDK\SAMPLES\IO\BMP.CPP`.

## Summary

This section presented an overview of the similarities and differences between the IPAS and 3ds max plug-in architectures. 3ds max provides a powerful development system for creating plug-ins that are tightly integration with the core program. IPAS developers are encouraged to implement their plug-ins using the standard 3ds max types in order to provide the best integration with the system. For those plug-ins that simply don't fit within the 3ds max architecture, the Utility API is provided to allow these plug-ins to work in MAX.

You may find it helpful to review the section [Plug-In Types Overview](#) to see a summary of the general categories of 3ds max plug-in applications.

## Updating MAX 1.0 Plug-Ins to Work with MAX 2.x

See Also: [What's New in the MAX 2.0 and 2.5 SDKs.](#)

## Overview

This section discusses some changes in the SDK APIs from 3ds max 1.0 to 2.0 or 2.5. Developers need to be aware of these changes as old plug-ins won't compile or operate properly unless they are made. The changes are broken down into specific areas that have changed. Some general notes related to all plug-ins appear at the beginning of this topic. For a brief overview of the API additions and changes for r2 see the section [What's New in the MAX 2.0 and 2.5 SDKs](#). Note that this section does not present all the areas of change -- only those where the change will prevent the plug-in from compiling/linking/executing under 3ds max 2.0.

## Recompilation

Plug-Ins that were developed for use with 3ds max 1.0 will need to be recompiled using the 3ds max 2.0 SDK in order to run. The **LibVersion()** function will prevent older plug-ins from running in 3ds max 2.0. These plug-ins will display the message:

**DLL <full pathname> is an obsolete version - not loading.**

In some cases a simple recompilation is all that is required. In other cases changes must be made. Developers need to be aware of the following changes to the API that affects their ability to run in 3ds max version 2.

Note: The **LibVersion()** method has been expanded to include information about the current version of MAX, the SDK, and the API. See the Advanced Topics section on [DLL Functions and Class Descriptors](#) for details.

**Also Note:** Generally, plug-ins developed for 3ds max 2.0 will run without recompilation on 3ds max 2.5. The only exception to this are those that use the NURBS API. Significant changes have taken place in the NURBS API and thus a recompile is required. If your plug-in **#includes** either **SURF\_API.H** or **TESSINT.H** (and you're using them, of course) you'll need to recompile. If you don't include these files, your plug-in developed for 3ds max 2.0 will run without problems on 3ds max 2.5 without recompilation.

## Development Environment

To use the 3ds max SDK you need to use the same version of the same compiler that is used to compile MAX. There are several reasons for this. The first reason is name mangling. Developers need a compiler with an identical name mangling scheme. The second reason is memory management. The plug-ins need to use the same memory manager as MAX.

Developers need to use Visual C++ Version 5.0 for developing plug-ins for 3ds max 2.0. The use of **any** previous version of VC++ is not supported.

Loading an existing project into VC++ 5.0 will attempt to convert it to the new format. If this process fails for some reason, or to build a new project from scratch, see the section [Creating a New Plug-In Project](#) for the proper settings.

## Checking the Current Version of MAX at Runtime

Developers should be aware that they cannot call methods provided in later versions of the 3ds max API from an earlier version of MAX. This can happen, for example, if a user tries to run a 3ds max 2.5 plug-in on 3ds max 2.0. The plug-in will load just fine, but the 2.5 specific functions would not exist. Developers can do something like the following to check the running version of MAX:

```
DWORD v = Get3DSMAXVersion();  
int r = GET_MAX_RELEASE(v);  
if (r >= 2500) {  
// Code that requires 2.5 API here  
}
```

## **New Parameters of Existing Methods**

Developers should watch out for the case where existing virtual methods have had new parameters added to them. This results in the compiler not seeing the new methods as being implemented. Rather the existing implementation of the method is just seen as a method of the sub-class. Instances of this are rare, however cases where new parameter have been added to existing methods are noted in the documentation for the methods.

## Passing Along Mouse Messages in Dialog Procs

In 3ds max 2.0 a change was made in the rollup window message handling so that the right mouse button menu works in all rollups. Developers can now remove the code from their dialog procs that passes the mouse button and move messages into **Interface::RollupMouseMove()**. For example, in 3ds max 1.x developers needed to pass along **WM\_LBUTTONDOWN**, **WM\_LBUTTONUP**, and **WM\_MOUSEMOVE** messages to MAX. This is no longer required. So, code such as the following can be removed from any dialog procedures.

```
case WM_LBUTTONDOWN: case WM_LBUTTONUP: case
WM_MOUSEMOVE:
    theUtility.ip->RollupMouseMove(hWnd, msg, wParam,
lParam);
    break;
```

To implement the change the **DWL\_USER** slot was used in the RollupPanel's window to store a pointer to the RollupPanel. This works fine with most plug-ins, which use **GWL\_USERDATA** to store a pointer to their context. Developers must be aware that if they are developing any code that uses **DWL\_USER** for a RollupPanel dialog proc to store its context, change it to use **GWL\_USERDATA**. There is a test in place to detect this which will put up an alert when the rollup is initialized, saying "Rollup Window Procs must use GWL\_USERDATA, not DWL\_USER", so developers breaking this rule will be informed pretty quickly.

## Working with the New Object Snap System

Developers who implement creation procedures for their plug-in that call **ViewExp::SnapPoint()** should add a call to **ViewExp::SnapPreview()** to their code where the **MOUSE\_FREEMOVE** message is processed. This allows the user to get proper visual feedback about object snaps that are in effect. See the Advanced Topics section on [Snapping](#), specifically the section 'Handling Snap Preview and Point Snap in Creation Procedures' for more details. Developers who don't do this will get snapping to occur in their plug-ins but won't get the visual feedback.

## Bitmap Memory Management

Previously bitmaps were deleted using the delete operator. This has changed. The bitmap class now has a **DeleteThis()** method that must be used to delete all bitmaps. For example, previously code to create a bitmap and delete it that looked like this:

```
Bitmap *bmap = TheManager->Create(&bi);  
// Do something...  
delete bmap
```

This is now written as follows:

```
Bitmap *bmap = TheManager->Create(&bi);  
// Do something...  
bmap->DeleteThis();
```

If you attempt to do it the old way you'll get a error message during compilation similar to **'Bitmap::~~Bitmap' : cannot access private member declared in class 'Bitmap'**.

## Space Warp Plug-Ins

It is important to note that source code examples in the 1.x SDK had the **space warp helper object** returning 1 from the **ClassDesc::IsPublic()** method. This indicated that the helper was available for use not just by the plug-in itself but by anyone. This was incorrect. However there was no user interface available to allow the user to choose and run these plug-ins independently so it was never caught as an error. With the new space warp architecture there is. Thus, the **ClassDesc::IsPublic()** method must return 0 instead of 1 for space warp helper objects to prevent the user from creating them without the space warp itself.

## Automatically Turning On Mapping Coordinates in Procedural Objects

The 3ds max renderer and plug-in procedural objects now support the ability to automatically turn on mapping coordinates when they are needed but aren't available. In 3ds max 1.x, if the renderer needed mapping coordinates for an object, and the 'Generate Mapping Coordinates' check box wasn't on, or a mapping modifier was not applied, an error message was presented to the user. In 3ds max 2.0 the renderer can request that the object turn on mapping coordinates itself and thus avoid having to put up the error message.

Procedural object plug-ins need to implement a few methods (**Object::HasUVW()** and **Object::SetGenUVW()**) to provide this capability. See the documentation for these methods in [Class Object](#) for more details.

## Supporting the Second Mapping Channel

In 3ds max 2.0 and later there are now two channels of mapping coordinates that can be assigned and carried by a mesh. This lets the user have two different sets of mapping coordinates, simultaneously, on the same face. Developers of plugins that provide mapping coordinate support should now also support the new second mapping channel.

The second mapping channel is stored in the mesh in the same arrays as the color per vertex information. Thus, the second mapping channel vertex array is stored in the **Mesh** data member **VertColor \*vertCol**, and the face array is **TVFace \*vcFace**. The number of texture verts is **int numCVerts**.

For additional details see [Class Mesh](#).

## Supporting the Drag and Drop System

Drag and drop functionality has been expanded to include all map and material buttons -- including those in plug-in materials and texmaps. As a result, a 3ds max user can drag the button over a like button to display the Swap/Copy/Cancel dialog. Developers creating plug-in materials and texmaps, as well as plug-ins with UI controls with a bitmap or material/texmap need to implement code to work with drag and drop. There are several new or revised classes to support this system. For additional details see [Class DADMgr](#), [Class TexDADMgr](#), [Class MtlDADMgr](#), [Class DADBitmapCarrier](#), [Class ICustButton](#), [Class IDADWindow](#).

## Splines

Three methods have been removed from [Class Spline3D](#) and replaced with six others. The **InVec()** and **OutVec()** methods were giving direct access to the spline data via a reference. The way the spline vectors are stored inside the class has been revised and this could no longer be allowed. Instead, there are new **GetInVec()**, **GetOutVec()**, **SetInVec()** and **SetOutVec()** methods. Also, all the data has been moved into the **private:** section to prevent improper access.

Developers using the **InVec()** methods will either be using it on the left side of the = or the right side. The changes that must be made are as follows:

Left Side of = (lvalue)

Example:

```
InVec(i) = Point3(0,0,0);
```

Change To:

```
SetInVec(i, Point3(0,0,0));
```

Right Side of = (rvalue)

Example:

```
Point3 p = InVec(0);
```

Change To:

```
Point3 p = GetInVec(0);
```

Also, the **KnotPoint()** method has been replaced. This is part of a larger series of changes which have made working with spline shapes much faster. By removing the direct access of the reference-returning methods, the **Spline3D** class will be able to control caching of various data within itself, preventing time-wasting repeated operations computing bezier control points. The **KnotPoint()** method has been replaced by two new methods:

```
Point3 GetKnotPoint(int i);
```

```
void SetKnotPoint(int i, Point3& p);
```

## ShapeObjects

**ShapeObjects** are now renderable. In order to accomplish this, they are now subclassed off of **GeomObject** rather than **Object**, as they were in previous versions. They are still **SHAPE\_CLASS\_ID** objects, though.

This has introduced a couple of important ramifications. See the remarks at the top of [Class ShapeObject](#) for the details.

## PatchMesh

The **PatchMesh** class has been updated so that it can contain multiple texture mapping channels, like the **Mesh** class. Unlike the **Mesh** class, however, the second texture mapping channel is stored in a new, second array member of the **PatchMesh** members, which are now defined as:

```
int numTVerts[PATCH_TEXTURE_CHANNELS];  
UVVert *tVerts[PATCH_TEXTURE_CHANNELS];  
TVPatch *tvPatches[PATCH_TEXTURE_CHANNELS];
```

At present, **PATCH\_TEXTURE\_CHANNELS** is defined as **2**, specifying the two texture mapping channels available. This could be extended in future versions, so keep this in mind.

Operations remapping texture patch information should perform the operations on all texture channels. See the file

**\MAXSDK\SAMPLES\MODIFIERS\EDITPAT.CPP** for examples.

All texture-mapping-related methods within the **PatchMesh** class have been adapted to deal with the new mapping channel. The old methods all access channel 0, to retain backwards compatibility. To access channels other than zero, new methods with "Channel" at the end of their names have been added:

## **NURBS**

The original 3ds max 1.x NURBS classes have been replaced in the SDK. Developers can get an overview of the new NURBS API by reading the section [Working with NURBS](#).

## Particle Systems

Several new methods have been added to class **ParticleObject** in 3ds max 2.0. These methods have default implementations but in order for the particle system to participate in Motion Blur when rendering these methods need to be implemented. These methods are: **ParticlePosition()**, **ParticleVelocity()**, **ParticleSize()**, **ParticleCenter()**, **ParticleAge()**, and **ParticleLife()**. See [Class ParticleObject](#) for details.

Also, the methods **CollisionObject::CheckCollision()** and **ForceField::Force()** both have a new parameter (**int index**), which is the index of the particle being forced or collided. See [Class CollisionObject](#) and [Class ForceField](#).

## Plug-Ins With an Interface in Track View

Plug-Ins that provide a user interface in Track View themselves will need to implement several new methods of [Class Animatable](#). These methods are all marked 'This method is available in release 2.0 and later only', and the methods themselves discuss the changes.

Track View has changed a bit. It now remembers the open/close state of the hierarchy as well as the selected/deselected state of the animatables. This information is saved with each Animatable in the data members maintained by the class:

**DWORD tvflags1, tvflags2;**

For example, several methods now have an additional parameter that specifies which Track View the method deals with.

There are other new methods that deal with the increased copy/paste functionality in R2 as well.

## Custom Creation Plug-Ins

In 3ds max 2.0 when a new object is created, if the hide by category flag for that object type is set to hidden, it is reset to visible before the object is created. In order to unhide a category (object, lights, particle systems, etc.) when an entity of that type is created, code was added to the creation routines of entities that don't use the default creation methods.

Making the required changes is simple. Just add the following statement during the first mouse down during creation:

```
GetCOREInterface()->SetHideByCategoryFlags(  
    GetCOREInterface()->GetHideByCategoryFlags() &  
    ~HIDE_X));
```

Where, **\_X** is **\_OBJECTS**, **\_LIGHTS**, etc. (or ~

**(HIDE\_OBJECTS|HIDE\_PARTICLES)** for particle systems.

Anyone moving custom-creation plug-ins from 3ds max 1.x to 2.x should add this functionality.

## Scaling Parameter Values

There is a new virtual method of **ReferenceMaker** named **RescaleWorldUnits(float f)**. Its purpose is principally to allow conversion of units between different unit scales (for instance centimeters to inches) when merging files.

What it is meant to do is to multiply any parameter that is expressed in world units by the scale factor **f**. This applies to things like a sphere's radius, a cylinder's radius and height, a control point position, etc.

To make this method work everywhere in 3ds max is going to require all plug-in developers to make sure it is implemented for their classes. In many cases the default implementations will work without change.

To detect if an object (or modifier) is rescaling correctly is simple:

- (1) Create the object and create a camera looking at it.
- (2) Render the camera view of the object.
- (3) In the Utility panel, bring up Rescale World Units utility, click "Rescale..." and set a scale factor to say 10. The "Affect" should be set to "Scene". Click OK. This will call **RescaleWorldUnits(10.0)** on the entire scene.
- (4) The camera view should have remained unchanged. Render it again to check.

Developers should check if your object/modifiers/space warp/whatever is rescaling correctly. If not, follow the steps below to make them do so.

The default **ReferenceMaker** implementation of **RescaleWorldUnits()** simply recursively calls **RescaleWorldUnits()** on the sub-references. (To avoid rescaling multiple instances more than once, a flag **A\_WORK1** is cleared before the whole process begins and is used to flag entities that have been rescaled.)

```
void ReferenceMaker::RescaleWorldUnits(float f) {  
    // This code should appear at the beginning of any  
    // RescaleWorldUnits implementation:  
    if (TestAFlag(A_WORK1))  
        return;  
    SetAFlag(A_WORK1);
```

```

// This code will be replaced in particular implementations
for (int i=0; i<NumRefs(); i++) {
    ReferenceMaker *srm = GetReference(i);
    if (srm)
        srm->RescaleWorldUnits(f);
}
}

```

The basic controllers have the method **RescaleWorldUnits()** implemented. Parameter blocks also have an implementation of **RescaleWorldUnits()** that will only rescale parameters for which the Param Dimension is **stdWorldDim**. (This is the value returned by **NotifyRefChanged()** in response to the **REFMSG\_GET\_PARAM\_DIM** message.) So if an object's parameters are all managed by a parameter block, making sure that **stdWorldDim** is returned only for those parameters which represent world units is all that need to be done. If some parameters are not handled by the parameter block, or there are some sub-references that you know need not be rescaled, you can implement **RescaleWorldUnits()**.

In certain cases, the default implementation for **ParamBlock::RedrawWorldUnits()** is not sufficient, or there may be conflict between your use of **stdWorldDim** and world unit rescaling. In this case, you will need to implement your object's **RescaleWorldUnits()** and use the method:

```

void IParamBlk::RescaleParam(int paramNum, float f)=0;

```

to rescale world unit parameters one at a time. For example, here is the **RescaleWorldUnits()** for the Fog atmosphere:

```

void FogAtmos::RescaleWorldUnits(float f) {
    if (TestAFlag(A_WORK1))
        return;
    SetAFlag(A_WORK1);
    pblock->RescaleParam(PB_TOP,f);
    pblock->RescaleParam(PB_BOTTOM,f);
}

```

In general it's pretty simple to get things scaling correctly. There can be some easy-to-miss subtleties, however. For instance, during 3ds max development

there appeared some anomalies in the shadows of scenes scaled by a large factor, and the fix involved scaling the shadow bias distance, which was overlooked. It's very important for plug-in developers to get this working across all entities so all developers should be careful in implementing this concept.

## Import / Export Plug-Ins

A change was made internally in 3ds max 2.0 that causes a bug in Export plug-ins. This bug will be addressed in a maintenance release, but for now, developers will need to make the following change in order for their description string to appear properly. Usually, this string is retrieved from a string table as in:

```
const TCHAR * AsciiExp::Ext(int n)
{
    switch(n) {
        case 0:
            return GetString(IDS_EXTENSION1);
        }
    return _T("");
}
```

This will be a problem in 3ds max 2.0 due to an internal change in the **ImpExp** code. The corrected code should just return the literal string, as in:

```
const TCHAR * AsciiExp::Ext(int n)
{
    switch(n) {
        case 0:
            return _T("ASE");
        }
    return _T("");
}
```

Developers should also be aware of one other change to these plug-in types.

There is a new parameter to the **SceneImport::DoImport()** and **SceneExport::DoExport()** methods. This is **suppressPrompts**. Developers of Import / Export plug-in may wish to respect this parameter to allow other 3ds max developers to use their plug-in for file IO (using **Interface::ImportFromFile()** and **ExportToFile()**). See [Class SceneImport](#) and [Class SceneExport](#) for details.

## Overview of the Principal Classes

This section lists the main classes of the SDK. The Overview hyperlinks take you to a brief summary of the methods of the class. The Reference hyperlinks jump to the full details on every method of the class.

## Overview Reference

[Animatable Class Animatable](#)  
[BaseObject Class BaseObject](#)  
[CameraObject Class CameraObject](#)  
[ConstObject Class ConstObject](#)  
[Control Class Control](#)  
[GeomObject Class GeomObject](#)  
[HelperObject Class HelperObject](#)  
[Inode Class INode](#)  
[LightObject Class LightObject](#)  
[LinearShape Class LinearShape](#)  
[Modifier Class Modifier](#)  
[Mtl Class Mtl](#)  
[MtlBase Class MtlBase](#)  
[ReferenceMaker Class ReferenceMaker](#)  
[ReferenceTarget Class ReferenceTarget](#)  
[Object Class Object](#)  
[ParticleObject Class ParticleObject](#)  
[PatchObject Class PatchObject](#)  
[ShapeObject Class ShapeObject](#)  
[SimpleMod Class SimpleMod](#)  
[SimpleObject Class SimpleObject](#)  
[SimpleParticle Class SimpleParticle](#)  
[SimpleShape Class SimpleShape](#)  
[SimpleSpline Class SimpleSpline](#)  
[SimpleWSMMod Class SimpleWSMMod](#)  
[SimpleWSMObject Class SimpleWSMObject](#)  
[SoundObj Class SoundObj](#)  
[SplineShape Class SplineShape](#)  
[StdControl Class StdControl](#)  
[Texmap Class Texmap](#)  
[TriObject Class TriObject](#)  
[WSMObject Class WSMObject](#)

## Main Plug-In Classes

See Also: [Plug-In Types Overview](#), [Overview of the Principal Classes](#).

The following is a list of plug-in types and the classes from which they may be derived:

Atmosphere: [Class Atmospheric](#).

Cameras: [CameraObject](#), [GenCamera](#).

Construction Grid Objects: [Class ConstObject](#).

Controllers: [Class Control](#), [Class StdControl](#).

File Import: [Class SceneImport](#).

File Export: [Class SceneExport](#).

Image Filter / Compositor: [Class ImageFilter](#).

Image Loader / Saver: [Class BitmapIO](#).

Helper Objects: [Class HelperObject](#).

Lights: [Class LightObject](#), [GenLight](#).

Materials: [Class Mtl](#).

Modifiers: [Class Modifier](#), [Class SimpleMod](#).

Particle Systems / Effects: [Class SimpleParticle](#), [Class Atmospheric](#).

Patch Objects [Class PatchObject](#).

Procedural Objects: [Class GeomObject](#), [Class SimpleObject](#).

Renderer [Class Renderer](#).

Spline Shapes: [Class SimpleSpline](#), [Class SplineShape](#).

Sound Plug-Ins: [Class SoundObj](#).

Space Warps: [Class WSMObject](#), [Class WSMModifier](#).

Textures: [Class Texmap](#).

Utility: [Class UtilityObj](#).

# Interface Classes

See Also: [Overview of the Principal Classes](#), [Interface Classes Overview](#).

The following classes provides interfaces into 3ds max:

General Interface into MAX: [Class Interface](#), [Class IObjParam](#), [Class IObjCreate](#).

Access node properties: [Class INode](#)

Viewport related methods: [Class ViewExp](#)

Loading and Saving data: [Class ILoad](#), [Class ISave](#).

Enumeration of nodes in the scene: [Class IScene](#).

Interface passed to materials and textures: [Class IMtlParams](#).

Access to the 3ds max Track Bar: [Class ITrackBar](#).

Access to XRef objects: [Class IXRefObject](#).

Access to the Layers functionality provided by 3D Studio VIZ. [Class ILayer](#).

Interfaces into the standard 3ds max plug-ins:

Access to Procedural Object and Space Warps parameters - See the methods in [Class BaseObject](#):

```
virtual IParamArray *GetParamBlock();  
virtual int GetParamBlockIndex(int id);
```

For an overview of this process see [Object Creation Methods](#).

Access to derived objects: [Class IDerivedObject](#).

Access to the standard 3ds max controllers: [Class IKeyControl](#).

Access to the standard 3ds max text object: [Class ITextObject](#).

Access to 3ds max Standard material properties: [Class StdMat](#).

Access to the 3ds max Bitmap texture properties: [Class BitmapTex](#).

Access into MAX's default WAV sound object. [Class IWaveSound](#).

Access to the Spline Select Modifier: [Class ISplineSelect](#), [Class ISplineSelectData](#).

Access to the Editable Spline Object: [Class ISplineOps](#).

Access to the Patch Select Modifier: [Class IPatchSelect](#), [Class IPatchSelectData](#).

Access to the Editable Patch Object: [Class IPatchOps](#).

Access to the FFD Modifier and World Space Modifier: [Template Class](#)

[IFFDMod.](#)

Interfaces into the parameter blocks and maps:

Access to parameter blocks: [Class IParamBlock](#), [Class IParamBlock2](#).

Access to parameter maps: [Class IParamMap](#), [Class IParamMap2](#).

## Geometry / Bitmap Classes

See Also: [Overview of the Principal Classes](#).

Mesh / Face: [Class Mesh](#), [Class Face](#), [Class TVFace](#), [Class TriObject](#).

Bitmap Images: [Class BitmapInfo](#), [Class Bitmap](#), [Class BitmapIO](#), [Class BitmapManager](#).

2D and 3D Points: [Class IPoint2](#), [Class Point2](#), [Class IPoint3](#), [Class Point3](#)

2D and 3D Boxes: [Class Box2](#), [Class Box3](#).

Matrices: [Class Matrix2](#), [Class Matrix3](#), [Structure AffineParts](#).

Angles / Quaternions: [Class AngAxis](#), [Class Quat](#), [Class ScaleValue](#).

## User Interface Classes

See Also: [Overview of the Principal Classes](#), [The Interactive Renderer](#).

Access the interactive renderer: [Class GraphicsWindow](#)

Create custom mouse modes: [Class CommandMode](#), [Class MouseCallBack](#)

Parameter Blocks / Maps: [Class IParamMap](#), [Class IParamBlock](#), [IParamArray](#).

Custom Controls: , [Class ICustomControl](#), [Class ICustEdit](#), [Class ISpinnerControl](#), [Class ICustImage](#), , [Class ICustStatus](#), [Class IColorSwatch](#), [Class ICustButton](#), [Class ICustToolbar](#), [Class IRollupWindow](#), [Class IOffScreenBuf](#).

## Miscellaneous Utility Classes

See Also: [Overview of the Principal Classes](#).

Bit flags using array access [Class BitArray](#)

Unique plug-in Class\_ID [Class Class\\_ID](#)

Class Descriptor [Class ClassDesc](#)

Gamma / de-gamma conversion [Class GammaMgr](#)

Palette computation [Class Quantizer](#)

Packing colors into a palette [Class ColorPacker](#)

Simple character string class [Class CStr](#)

Wide character string class [Class WStr](#)

Table of names [Class NameTab](#)

Table template class [Class Tab](#)

Color picker dialog [Class ColorPicker](#)

## Reference Introduction

This module of the documentation is a reference to the classes provided in the SDK.

There are sections on typographic conventions, data types, and a structured class list organized by category.

At the beginning of each class is an overview of the purpose of the class. All the data members, methods and operators used by each class are documented.

## Typeface Conventions

This documentation uses the following typographic conventions:

### **Typeface Use**

**Sans Serif** Used for most text.

**Sans Serif Bold** Used for titles and headings.

**Monospace bold** Program text, variables, classes, types and other program constructs.

## Data Types

This section lists the data types most commonly used in the SDK. Some of these data types are defined by 3ds max while others are part of the Win32 API.

### **BOOL**

A boolean value, either TRUE or FALSE.

### **BYTE**

An 8-bit unsigned value.

### **CIRCLE**

The CIRCLE structure defines the x and y coordinates of the circle center and radius value. A CIRCLE data structure has the following form (defined in **GFX.H**):

```
typedef struct tagCIRCLE
{
    LONG x;
    LONG y;
    LONG r;
} CIRCLE;
```

### **COLORREF**

A 32-bit value used as a color value. See [COLORREF](#).

### **DWORD**

A 32-bit unsigned integer or the address of a segment and its associated offset.

### **HFONT**

The handle of a font.

### **HCURSOR**

The handle of a cursor.

### **HDC**

The handle of a device context (DC).

### **HIMAGELIST**

The handle to an image list.

### **HINSTANCE**

The handle of an instance.

### **HMENU**

The handle of a menu.

## **HPALETTE**

The handle of a palette.

## **LONG**

A 32-bit signed integer.

## **LPARAM**

A 32-bit value passed as a parameter to a window procedure or a callback function.

## **LPSTR**

A 32-bit pointer to a character string.

## **LRESULT**

A 32-bit value returned from a window procedure or callback function.

## **POINT**

The POINT structure defines the x and y coordinates of a point. A POINT data structure has the following form:

```
typedef struct tagPOINT {  
    int x;  
    int y;  
} POINT;
```

## **RECT**

The RECT structure defines the coordinates of the upper-left and lower-right corners of a rectangle. A RECT data structure has the following form:

```
typedef struct tagRECT {  
    int left;  
    int top;  
    int right;  
    int bottom;  
} RECT;
```

## **USHORT**

A 16-bit unsigned short integer.

## **SIZE**

This structure specifies the width and height of a rectangle. The rectangle dimensions stored in this structure can correspond to viewport extents, window extents, text extents, bitmap dimensions, or the aspect-ratio filter for some extended functions.

**typedef struct tagSIZE {**

**int cx;**

**int cy;**

**} SIZE;**

**ULONG**

A 32-bit unsigned long integer.

**WNDPROC**

A 32-bit pointer to a window procedure.

**WORD**

A 16-bit unsigned integer.

**LPARAM**

A 32-bit value passed as a parameter to a window procedure or callback function.

Other various typedefs and structures used in the SDK:

**typedef unsigned long ulong;**

**typedef unsigned char uchar;**

**typedef uchar UBYTE;**

**typedef unsigned short USHORT;**

**typedef unsigned short UWORD;**

**typedef int TimeValue;**

**typedef unsigned short BMMRES;**

**typedef ulong SClass\_ID;**

**typedef unsigned int RefMessage;**

**typedef unsigned short MtlID;**

**typedef unsigned long ChannelMask;**

**struct Color24 {**

**uchar r,g,b;**

**};**

**struct Color48 {**

**UWORD r,g,b;**

**};**

**struct Color64 {**

**UWORD r,g,b,a;**

```
};
typedef Point3 UVVert;
typedef Point3 VertColor;
typedef AColor RGBA;

typedef enum {IO_OK=0, IO_END=1, IO_ERROR=2} IOResult;
typedef enum {NEW_CHUNK=0, CONTAINER_CHUNK=1,
DATA_CHUNK=2} ChunkType;
enum RefResult {
    REF_FAIL,
    REF_SUCCEED,
    REF_DONTCARE,
    REF_STOP,
    REF_INVALID
};
enum LightType { OMNI_LGT, SPOT_LGT, DIRECT_LGT,
    AMBIENT_LGT };
typedef enum {IOTYPE_MAX=0, IOTYPE_MATLIB=1} FileIOType;

typedef unsigned long NURBSId;
typedef Tab<NURBSId> NURBSIdTab;
typedef Tab<BOOL> BoolTab;

typedef Tab<WVert *> WVertTab;
typedef Tab<WEdge *> WEdgeTab;
typedef Tab<WFace *> WFaceTab;

typedef DWORD HJOB;
typedef INT_PTR_MSVC70 StringResID;
typedef INT_PTR_MSVC70 ResID;
typedef Animatable* AnimatablePtr;
typedef DWORD ActionTableId;
typedef DWORD ActionContextId;
typedef DWORD HJOB;
typedef INT_PTR_MSVC70 StringResID;
typedef INT_PTR_MSVC70 ResID;
```

```
typedef Animatable* AnimatablePtr;  
typedef DWORD ActionTableId;  
typedef DWORD ActionContextId;
```

## General Terminology

This section defines various terms used throughout the SDK documentation.

- [= 4\) BSPSPopupOnMouseOver\(event\);;">Abstract Class](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Affine Transformation](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Apparatus](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Base Object](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Callback](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Class](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Class Hierarchy](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Class Variable](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Controller](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Cache System](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Class Descriptor](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">ClassID](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Channel](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Deep Copy](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Device Context](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Deformable](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Derived Object](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Dialog Proc](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">DLL](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Geometric Pipeline](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Hit Testing](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Instance](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Interval](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Metadata](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Modifier](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Modifier Application](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Node](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Object Space Modifier](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Orthonormal Matrix](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">PRS Controller](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Reentrant Function](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Reference Maker](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Reference Target](#)
- [= 4\) BSPSPopupOnMouseOver\(event\);;">Shallow Copy](#)

- = 4) BSPSPopupOnMouseOver(event);;">Space Warp
- = 4) BSPSPopupOnMouseOver(event);;">Sub-Object Selection
- = 4) BSPSPopupOnMouseOver(event);;">SuperClassID
- = 4) BSPSPopupOnMouseOver(event);;">TriObject
- = 4) BSPSPopupOnMouseOver(event);;">Transform Controller
- = 4) BSPSPopupOnMouseOver(event);;">Transformation Matrix
- = 4) BSPSPopupOnMouseOver(event);;">Time Value
- = 4) BSPSPopupOnMouseOver(event);;">Validity Interval
- = 4) BSPSPopupOnMouseOver(event);;">Virtual Array
- = 4) BSPSPopupOnMouseOver(event);;">World Space Modifier

## **What's New in the MAX 2.0 and 2.5 SDKs**

See Also: [Updating MAX 1.0 Plug-Ins to work with MAX 2.x.](#)

## Overview

This section provides a general overview of the new capabilities of the 3ds max R2 API. New classes have been added, new methods to existing classes have been added, and new parameters to existing methods have been added as well. Any new classes in the SDK begin their Description section with the line:

**This class is available in release 2.0 and later only** or

**This class is available in release 2.5 and later only.**

Any new methods in a class begin:

**This method is available in release 2.0 and later only** or

**This method is available in release 2.5 and later only.**

Newly added parameters to existing methods begin:

**This parameter is available in release 2.0 and later only** or

**This parameter is available in release 2.5 and later only.**

Newly added data members begin:

**This data member is available in release 2.0 and later only** or

**This data member is available in release 2.5 and later only.**

The sub-sections below list some of the major areas of improvement or new capabilities in the SDK. This includes new plug-in types available, some animation capabilities for ImageFilter plug-ins, and the G-buffer system. This section does not discuss all the changes -- only the major ones. Developers of existing 1.x plug-ins can look at the reference information for their plug-in class to review any new capabilities. Note that there is a separate section in the SDK which discusses how to get existing 3ds max R1 plug-ins to work with R2. See [Updating MAX 1.0 Plug-Ins to work with MAX 2.x](#) for details.

## New in the MAX 2.5 SDK

The major change to the 2.5 API is related to NURBS. The additional capabilities of NURBS system (such as projected curves, trimmed surfaces, etc.) have been made available via the SDK. Developers will need to recompile plug-ins that use the NURBS API to run on 3ds max 2.5 because of these significant changes. If your plug-in **#includes** either **SURF\_API.H** or **TESSINT.H** you'll need to recompile. If you don't include these files, your plug-in developed for 3ds max 2.0 will run without problems on 3ds max 2.5 without recompilation. See the Advanced Topics section [Working With NURBS](#) for information on the new classes available in 2.5.

Also enhanced for 3ds max 2.5 are the Minnesota Mesh and related classes. See [Class MNMesh](#) for details.

There are two new callbacks available via [Structure NotifyInfo](#) for pre-save and post-save notification.

## New Plug-In Types in MAX 2.0

### World Space Modifiers Without Helper Objects.

There is a new type of space warp plug-in that doesn't require a helper object to operate in world space. This was done because some plug-in modifiers needed to operate in world space, and thus needed to be space warps, but didn't require being bound to the dummy helper object. The new WSMs are available to be assigned like OSMs via buttons in the Modify branch of the command panel. They don't need to be specifically bound to a helper object to be created and assigned. 3ds max uses the **ClassDesc::IsPublic()** method to determine if they may be directly assigned. If **IsPublic()** returns nonzero they may be assigned by the user. If it returns zero then they depend on the helper object and can only be created when bound to the helper.

### Easily created WSM versions of OSMs

Modifiers derived **SimpleMod** may also be used to easily create a Space Warp version as well. See [Class SimpleOSMToWSMObject](#) for details on how this is done.

### Track View Utilities

There is a new type of utility plug-in that operate inside of Track View. These are accessed by pressing the Track View Utilities icon in the Track View toolbar. Samples of this type of utility are the 'Randomize Keys', 'Create Out of Range Keys' and 'Select Keys by Time'. These plug-ins are derived from [Class TrackViewUtility](#). See that class for more details on their creation.

### Front End Controllers

These new plug-ins allow a developer to completely take over the 3ds max user interface. This includes the toolbar, pulldown menus, and command panel. See [Class FrontEndController](#) for details.

### Motion Capture Input Devices

Motion Capture Input Device plug-ins can now be written that plug-in to the 3ds max motion capture system. See [Class IMCInputDevice](#) for details. Sample code is available in the subdirectory **\MAXSDK\SAMPLES\MOCAP**.

### Notification Program

There is a new program whose source code is in **\MAXSDK\SAMPLES\UTILTIES\nOTIFY\nOTIFY.CPP**. This program gets invoked by the network manager to handle network progress

notifications. The network manager calls it with two command line arguments as follows:

### **Notify.exe datafile type**

Where:

#### **datafile**

This is the fully qualified path and filename of the actual notification notice. This is the text that may be sent out as email, fax or whatever the notification program wants to do.

#### **type**

This is a numeric value indicating what type of notification this is. The types are defined in `\MAXSDK\INCLUDE\ALERTS.H`. The options are:

**NOTIFY\_FAILURE**

**NOTIFY\_PROGRESS**

**NOTIFY\_COMPLETION**

A developer may write another "Notify" program in order to do any proprietary type of notifications. Note that "Notify" can be either a "\*.exe", a "\*.bat", or a "\*.cmd" executable. This allows a user to create a simple script file to do something without having to resort to writing a binary program.

The current **Notify.exe** is very simple as it is used simply as a demonstration. It plays a different wave file for each of the event types. If invoked with no command line, it will bring up a dialog box asking the user to define each of the three wave files. The dialog has "Browse" buttons next to each wave file field which puts the user right into the Windows' "Media" directory where wave files are saved. There are also "play" buttons next to each sound so they can be tested.

## Animated Parameter in ImageFilter Plug-Ins

A new system has been provided to allow Filter plug-ins to use animated parameters. These parameters are just like the other parameters in 3ds max that are controlled by an animation controller. These parameters appear in Track View as a separate branch under 'World' labeled 'Video Post'. In order for this to work a developer must create something called a Track View Node. Controllers may then be added to this node. Using the **SetValue()** and **GetValue()** methods of the controller allow for values to be stored and retrieved.

Also, Filters may have interactive dialog boxes that may be open and operated at the same time as 3ds max and Track View. This allows a user to press the Animate button and adjust the properties of the filters parameters to animate the value. See [Class ImageFilter](#), [Class ImageFilterInfo](#) and [Class ITrackViewNode](#) for the new methods available.

## The G-Buffer System

Three new channels has been added to the G-buffer. These are **BMM\_CHAN\_COVERAGE**, **BMM\_CHAN\_BG**, and **BMM\_CHAN\_NODE\_RENDER\_ID**. See [List of Image Channels](#) for more details.

## **New Notifications (Callbacks)**

3ds max 2.0 supports a system where a plug-in can ask to receive a callback when events such as the system unit settings change, system time settings change, or the user executes File/Reset or File/New. The documentation for [Structure NotifyInfo](#) describes how this system works.

## New Mesh Related Classes

The **MNMesh** class is provided in 3ds max 2.0 for temporary use by plug-ins, to help with complex topology-based modifications to Meshes. It has capabilities, such as the ability to recognize faces with more than 3 sides, that are useful in certain applications. See [Class MNMesh](#) for details.

## **New NURBS API**

The NURBS API from 3ds max 1.0 has been replaced in 3ds max 2.0 and extended in 3ds max 2.5. Developers now have a more complete API for developing with NURBS. See the Advanced Topics section [Working With NURBS](#) for details.

## **New Pre and Post Save Callbacks**

For use with 3ds max 2.5 and later, two new options have been added for use with [Structure NotifyInfo](#). These allow plug-ins to get called before and after a 3ds max file is saved.

## **Network Rendering Manager Access**

There is a new Advanced Topics section that documents the protocol used to communicate with the network rendering manager. This is for developers who want to write their own Queue Manager and/or submit net render jobs on their own. This is not an SDK and there is no code -- it is just a detailed description of the TCP/IP protocol used to communicate with the manager. See the Advanced Topics section on [Network Rendering](#) for information.

## **Plug-In Project AppWizard**

New to the 3ds max 2.0 SDK is an AppWizard which may be used to easily create new plug-in projects. See the Advanced Topics section on [Creating a New Plug-In Project](#) for details on using the AppWizard.

## **On-Line Help Inside the Visual C++ IDE**

A program is provided in `\MAXSDK\HELP\SDKLINK.ZIP` to allow developers to launch the SDK help file and jump to a specific class or method from within VC++. See the **README.DOC** file inside that ZIP for a description for how to set this up inside the Developer Studio IDE.

# What's New in the MAX 3.0 SDK

See Also: [Required Changes to MAX 2.x Plug-Ins for MAX 3.0](#), [What's New in the MAX 2.0 and 2.5 SDKs](#), [Updating MAX 1.0 Plug-Ins to work with MAX 2.x](#).

## Overview

This section provides a general overview of the new capabilities of the 3ds max 3.0 API. New plug-in types, new classes, and new methods to existing classes have been added. In some cases new parameters to existing methods have been added as well.

Any new classes in the SDK begin their Description section with the line:

**This class is available in release 3.0 and later only**

Any new methods in a class begin:

**This method is available in release 3.0 and later only**

Newly added parameters to existing methods begin:

**This parameter is available in release 3.0 and later only**

Newly added data members begin:

**This data member is available in release 3.0 and later only**

The sub-sections below list some of the major areas of improvement or new capabilities in the SDK. This includes new plug-in types, new parameter map system, new texture map architecture, new keyboard accelerator system, as well as numerous other changes.

## New Plug-In Types

The following are the new plug-in types introduced in the 3ds max 3.0 SDK. Each has link to the base class for the creation of that plug-in type.

### Rendering Effects

There is a new item under the Rendering menu which displays the Rendering Effects dialog. From this modeless dialog, the user can select and assign a new class of plug-in, called a “Rendering Effect,” which is a post-rendering image-processing effect. This lets the user apply image processing **without** using Video Post, and has the added advantage of allowing animated parameters and references to scene objects. The base

class for these plug-ins is [Class Effect](#). Sample code is available in the directory `\MAXSDK\SAMPLES\RENDER\RENDEREEFFECT`.

### **Anti-Aliasing Filters**

There is a new plug-in type for filtering and anti-aliasing the image. Documentation for the base class for these filters is in [Class FilterKernel](#). Sample Code is available in the subdirectory `\MAXSDK\SAMPLES\RENDER\AAFILTERS`.

### **Shader Plug-Ins**

This new plug-in type works with the new Standard material. It allows plug-in developers to add additional shading algorithms to the drop down list of available options (previously Constant, Phong, Blinn, Metal). This was only possible previously by writing an entire Material plug-in (which could be a major undertaking). See the base class for this plug-in type [Class Shader](#) for details.

### **Sampler Plug-Ins**

This plug-in type works with the Standard material of release 3. A Sampler is a plug-in that determines where inside a single pixel the shading and texture samples are computed. The user interface of Samplers appears in the Super Sampling rollout in the Sampler dropdown. See [Class Sampler](#) for details.

### **Shadow Generator Plug-Ins**

The generation of shadows is now accessible via this new plug-in type. The standard 3ds max mapped and raytraced shadows have been revised to be plug-ins of this form. See [Class Class ShadowType](#) and [Class ShadowGenerator](#) for details. There is also a handy class for creating shadow map buffers. See [Class ShadBufRenderer](#).

### **Color Selector Plug-Ins**

This new plug-in type provides the user with a custom color picker that appears whenever a standard 3ds max color swatch control is clicked. These plug-ins are selected in the General tab of the Preferences dialog. The color picker chosen is saved in the 3DSMAX.INI file in the "ColorPicker" section so that the choice is maintained between sessions. If the DLL for the selected color picker is not available, it will always default back to the "Default" color picker. See [Class ColPick](#) for details.

### **Global Utility Plug-Ins**

These simple utility plug-ins are loaded at boot time, after initialization, but before the message loop starts, and remain loaded. This is how the new 3ds max COM/DCOM interface is implemented. For details see [Class GUP](#).

## Ready To Build Plug-In Projects

The 3ds max 4.0 SDK comes provided with an SDK appwizard that can be used with Microsoft Visual C++. This appwizard allows you to quickly generate skeleton source code for a variety of different plugin types and essentially replaced the previous R3\_SKELETON projects included in the previous release of the software.

The generated code from the appwizard provides a quick way to start building plugins. Each project follows a standard structure with regard to headers and code files. Developers should find it easy to take this generated code and can start creating plugins without manually setting up the entire project from scratch.

## New Example Code for R3 Programming Concepts

The SDK has sample code for demonstrating several programming concepts and exposed APIs introduced in R3. This code can be found in the sub-directories of **\MAXSDK\SAMPLESHOWTO\EXAMPLES**. The API exposed by the Morpher modifier is demonstrated in **MORPHERAPI**. In the directory **SCRIPTPLUGIN** is a utility that demonstrates SDK access to a scripted plugin's parameters. Code which demonstrates using the Custom User Interface APIs is in **CUI-TEST**.

## MAXScript SDK

The MAXScript SDK is a set of Visual C++ headers and import libraries that programmers can use to extend MAXScript. The MAXScript SDK source is now a part of the main 3ds max SDK source tree. It is available in the directory **\MAXSDK\INCLUDE\MAXSCRIPT**. In order to use the MAXScript SDK, you'll need to add **\MAXSDK\LIB\MAXSCRIPT.LIB** to your project and add **\MAXSDK\INCLUDE\MAXSCRIPT** to your preprocessor include directories. See the Advanced Topics section [MAXScript SDK](#) for documentation on this SDK.

There are two samples programs available in the directory

**\MAXSDK\SAMPLES\HOWTO\MXS\_SAMPLES.** These are **TESTDLX** and **MXSAGNI**. **TESTDLX** shows how to add new types of classes to MAXScript. It also shows how to add custom controls. The other is **MXSAGNI** which shows how to expose globals, struct globals, primitives and struct primitives. It also shows how to add new types of classes like BigMatrix, TrackViewPick, and Physique and Biped interfaces. There is also sample code in the SDK showing how to add a custom function to MAXScript, add a custom UI element, add a system variable and add a class. For this see the projects in **\MAXSDK\HOWTOR3\_SKELETONS\MAXSCRIPTSDK.**

## **COM/DCOM Interface**

The COM/DCOM interface available in 3ds max 3.0 may be used to load and save scenes, import files, and drive the 3ds max renderer. The interface for doing this is discussed in the Advanced Topics section [COM/DCOM Interface](#).

## **Deferred Loading of Plug-Ins**

3ds max 3.0 has introduced the concept of delay-loading plug-ins. In previous releases every plug-in was loaded at 3ds max load time. Now, certain plug-ins are only loaded as needed. This results in a smaller memory footprint for 3ds max and shorter load time at startup. The impact of this on developers is that the method **ClassEntry::CD()** may return a pointer to an instance of class **DataClassDesc** rather than a full **ClassDesc** as it did previously. This derived class implements **Create()** by returning NULL. Developers need to call **ClassEntry::FullCD()** in order to ensure that the class is actually there. See [Class DataClassDesc](#) and [Class ClassEntry](#) for information on these methods. For additional details on delay-loading see the Advanced Topics section [Deferred Loading of Plug-Ins](#).

## **New Parameter Map System**

The existing parameter map system remains for backwards compatibility but a new and improved system is available in the 3ds max 3.0 SDK. This system makes user interface coding even simpler than before and allows plug-ins to properly integrate with 3ds max 3.0's Macro Recorder, 3ds maxScript and Schematic View. For details on this system see the Advanced Topics section

## [Parameter Blocks and Maps in Release 3.](#)

### **Multiple Map Support**

New in 3ds max 3.0 is the ability to use more than two texture maps per face of a mesh. Now users may use up to 100. This has impacted how mapping coordinates are stored by the mesh as well as how the color per vertex data is stored. There are corresponding new methods in [Class Object](#) and [Class Mesh](#). Also see the Advanced Topics section [Working with Meshes](#) (Mapping Channels in Release 3.0 and Later) for additional information.

This has also affect Patches and NURBS. See the sections [Required Changes to Patch Related Plug-Ins](#) and [Required Changes to NURBS](#) for details.

### **Access to Scene XRefs and XRef Objects**

3ds max 3.0 introduces Scene and Object External References (XRefs). Scene XRefs are stored as complete hierarchies with the XRef scene's root node as a child of the client scene's root node. There are new methods in [Class INode](#) to access these Scene XRefs subtrees. There are also two new methods in [Class Interface](#) for controlling the scene node traversal. Object XRefs are derived from [Class IXRefObject](#) and this class provides access to the Object XRef parameters.

### **New G-Buffer Capabilities**

The G-Buffer has been significantly enhanced for R3. The concept of 'layers' has been added as well as four new channels. The multiple layers of the G-Buffer allow image processing effects to do better anti-aliasing and handling of transparency. The new channels are:

#### **BMM\_CHAN\_COLOR**

This is the color returned by the material shader for the fragment.

#### **BMM\_CHAN TRANSP**

This is the transparency returned by the material shader for the fragment.

#### **BMM\_CHAN\_VELOC**

This gives the velocity vector of the fragment relative to the screen, in screen coordinates.

#### **BMM\_CHAN\_WEIGHT**

This is the sub-pixel weight of a fragment.

See [Class GBuffer](#) and [List of Image \(G-Buffer\) Channels](#) for details.

## **New NURBS Related Classes**

The NURBS API has been extended and altered in release 3.0. The primary change has been in the way texture surface are handled. See the sub-topic **Texture Mapping** in the [Working with NURBS](#) topic for an overview of the classes. Also see the section [Required Changes to NURBS Related Plug-Ins](#) for an overview of the other changes. The new classes introduced in release 3.0 are:

[Class NURBSTextureChannelSet.](#)

[Class NURBSTextureChannel.](#)

[Class NURBSTexturePoint.](#)

[Class NURBSSurfaceEdgeCurve.](#)

[Class NURBSFilletSurface.](#)

[Class NURBSProceduralCurve.](#)

[Class NURBSProceduralSurface.](#)

## **Access to the File Properties Data**

The 3ds max File Properties dialog allows a user to enter various information to be stored in the 3ds max file. Developers can access this information via the SDK. Methods of class Interface provide this access (see the [PropertySet methods of Class Interface](#) for details). There is a sample project in the SDK which demonstrates how this is done. See the code in

**\MAXSDK\SAMPLES\UTILITIES\PROPERTYTEST\PROPERTYT**

## **New Notifications (Callbacks)**

The SDK provides a notification system where a plug-in can ask to receive a callback for events such as the user executing File/Reset, changing the viewport layout, rendering the scene, etc. There are several new types of notifications codes that may be registered in R3. The documentation for [Structure NotifyInfo](#) and [List of Notification Codes](#) describes them.

## **Ability For Non-3D Windows to Appear in a 3ds max Viewport**

Programs that want to have their user interface appear in a 3ds max viewport (much like Track View and the Asset Manager do) can now accomplish this. See [Class ViewWindow](#) for details.

## New Keyboard Shortcut System

This new system is used to register keyboard shortcuts in a uniform manner. In 3ds max 3.0 in the Customize / Preference Settings / Keyboard tab dialog there is a section to assign Plug-In shortcuts to commands. For details on how this works see [Keyboard Shortcut System](#). Sample code is available in the directory `\MAXSDK\SAMPLES\MODIFIERS\FFD`.

## New Custom User Interface API

3ds max now allows users to customize the user interface. There is an API that allows developers to do some customization as well. Developers can create their own custom toolbars and tab panels which may be docked or floated. The buttons can run 3ds max keyboard commands and execute 3ds max script commands in addition to the usual custom toolbar functionality (icon buttons, flyoffs, etc.)

There is a sample program in the SDK which demonstrates the use of these API. It's in `\MAXSDK\SAMPLES\HOWTO\EXAMPLES\CUI-TEST`.

Follow the instruction in the comments at the top of the code to build and run this plug-in.

The main classes used in working with the custom UI are:

### [Class ICustToolbar](#)

This existing class has been enhanced to support the new CUI functionality. Toolbars may now support multiple rows, or appear vertically. They may also have macro buttons (added with the **MacroButtonData** class) which may have icons or text.

When a Toolbar is part of a CUI frame it's called a Tool Palette. Tool Palettes can either float or dock (whereas a Toolbar must be placed by the developer in a dialog using the resource editor). When you want to create a Tool Palette you create a Toolbar (as before) and then you create a **CUIFrame** and then link the two using a method of **CUIFrame**.

### [Class MacroButtonData](#)

A Macro Button is a button which can execute either a keyboard macro or a MAXScript macro. This class contains the data and access methods for such a UI button.

### [Class ICUIFrame](#)

This class provides access to the individual windows (frames) that contain the toolbars, menus, etc.

### [Class CUIFrameMgr](#)

There is one instance of the **CUIFrameMgr** and it controls the overall operation of the individual CUI frames. This class has methods for things like docking a tool palette, bringing up the toolbar right click menu, and returns button IDs and window handles.

### [Class CUIFrameMsgHandler](#)

Since the CUI frame is just a window, it needs a window proc. There is one built into the CUI system, but certain frames may need additional information that is specific to how the frame is being used. For example, the command panel is a CUI frame which can't be resized horizontally. To manage this, the application must install a **CUIFrameMsgHandler** object. This class has one method, **ProcessMessage()**.

### [Class CUIPosData](#)

This is the object that provides the position data when the **CUIFrameMsgHandler::ProcessMessage()** method receives a **CUI\_POSDATA\_MSG** message. The developer creates an instance of this class and implements the **GetWidth()** and **GetHeight()** methods which return size information based on the size type and orientation passed.

### [Class ToolMacroItem](#)

This class allows a Macro button control to be added to the toolbar.

### [Class MacroDir](#)

This class provides access to Macro scripts. Macro scripts (or macros) are scripts that live in buttons and menus in the customizable UI. Methods of this class are available to access macros using IDs or category and name strings, methods to edit macro scripts, methods to execute macros, and methods for directory scanning and loading.

### [Class MacroEntry](#)

This class provides access to a single macro entry. There are methods

provided to access the macro ID, name, category, file name, tooltip, UI button text, and the UI button icon. MacroEntries are returned from methods of class **MacroDir**.

## **New Custom Slider and Curve Controls**

There is a new custom horizontal slider control. This control is functionally similar to the custom spinner control with a few handy features. An example of this control can be seen in the Color Balance Render Effect. See [Class ISliderControl](#) for details.

There is also a new custom curve control. An example of this control in the 3ds max user interface can be seen in the Color Map section of the Output rollup of a 2D Texture map. Sample code using these APIs is available in `\MAXSDK\SAMPLES\UTILITIES\CCUTIL\CCUTIL.CPP`. See [Class ICurveCtl](#) for reference information.

## **New RightClickMenuManager Methods**

These new methods let plug-ins add sub-menus to the viewport right-click menu. Developers can call the methods

```
int BeginSubMenu(LPCTSTR name);
```

```
int EndSubMenu();
```

to start and end sub-menus. See [Class RightClickMenuManager](#).

## **Access to the Track Bar**

Directly below the time slider is a new Track Bar, which offers a quick way to manipulate keyframes for selected objects. Keys are displayed on the track bar just like they are in Track View. Developers have access to a class which may be used to manipulate the Track Bar. See [Class ITrackBar](#) for details.

## **New Multiple Viewport Enabled Mouse Proc**

This new mouse proc allows drawing in multiple viewports, offsetting from the construction plane, and orthogonal and angle snapping. This is the mouse proc that NURBS uses for curve creation. This allows developers to support orthogonal snapping and angle snapping on creation like the Bezier Line tool does. Also if the user presses Shift while dragging the mouse, the point is

snapped to the nearest quadrant (ortho snapping). If the Alt key is held, the point is snapped using the setting of the angle snap system.

Users of this mouse proc need to sub-class from the [Class DataEntryMouseProc](#) and implement some virtual methods to use it.

## **Ability to Replace the 3ds max Load and Save File Dialogs**

There are two new classes and two methods of **Interface** that allow developers to replace the load and save file dialogs with custom ones. See the methods:

**virtual void SetMAXFileOpenDlg(MAXFileOpenDialog\* dlg)=0;**

**virtual void SetMAXFileSaveDlg(MAXFileSaveDialog\* dlg)=0;**

in [Class Interface](#) for more information.

## **Schematic View API**

The Schematic View window allows users to review, and perform certain operations on, many of the objects that make up the scene. See the Advanced Topics section on [Schematic View](#) for an overview of the API. Developers have access to method for working with Schematic View as part of the following classes:

### [Class IGraphObjectManager](#)

This class represents an instance of a schematic view window and provides methods for adding nodes and node pointers, refreshing the schematic view, accessing filter bits and updating and controlling the various editors within 3ds max.

### [Class IGraphNode](#)

This represents a node in the schematic view graph and provides a few methods for querying information about the node.

### [Class IGraphRef](#)

IGraphRef represents a node pointer in Schematic View.

### [Class Animatable](#)

There is a set of methods in Animatable that can be overridden, all or in part, to specialize the behavior of the schematic view node(s) which represents the Animatable object.

### [Class SubClassList](#)

This existing class also has some Schematic View drawing related methods (**GetUIInfo()** and **SetUIInfo()**).

## **Random Number Generator**

There is a new class that may be used to generate pseudo-random number in either floating point or integer format within a specified range. Each instantiation of this class is independent, permitting several uncoupled random number generators to be present in the system at once. See [Class Random](#).

## **Macro Recorder API**

3ds max R3 introduced the concept of macro recording. The macro recorder generates MAXScript code for operations the user performs when running 3ds max. These scripts can then be played back to automate operations inside 3ds max. Since 3ds max is made up of plug-ins there needs to be a way that the plug-ins can record their own changes as the user operates them. This is the purpose of the Macro Recorder API.

Most common operations performed by a plug-in are handled automatically. For example the getting and setting of parameter block values automatically generates script for the Macro Recorder. In such cases a plug-in developer doesn't need to do anything. There are other operations that can't be handled automatically, however. For example a plug-in may have a button in its user interface that internally results in some looping. To generate code to record a loop the methods of **MacroRecorder** would be used. See [Class MacroRecorder](#).

## **Access To 3D Studio VIZ Layers**

For use with the layer capabilities of 3D Studio VIZ several classes have been added to the 3ds max API. See [Class ILayer](#) and [Class ILayerManager](#) for details.

## **New Methods for the IK Master Controller**

The new methods provide additional access to the controls. See [Class IKMasterControl](#).

## New and Revised Mesh Related Classes

The following classes track various aspects of an edit operation to a mesh. They are the principal means of keeping track of what's going on in the Edit Mesh modifier, and have many standard mesh edits available for use by other plug-ins.

### [Class MeshDelta](#)

This is a class that represent some kind of change to a mesh. This “delta” can include topological, geometric, map, and/or selection changes. Most standard mesh “edits” available in the Editable Mesh or Edit Mesh interface are available through the MeshDelta SDK, giving developers a powerful way to manipulate meshes while not having to “sweat the details” of maintaining maps to match the mesh changes, updating edge selections, etc.

### [Class VertMove](#)

This class represents the notion of a mesh edit vertex move. The public data members provide the index of the vertex moved as well as the amount of the move in X, Y, and Z.

### [Class UVVertSet](#)

This class represents the notion of a mesh edit UVW vertex assignment. The public data members provide the index of the vertex as well as the UVWVert.

### [Class FaceRemap](#)

This class represents the notion of a mesh edit Face Remap, which changes one or more of the verticies a face uses. It can also alter the visibiliy of the face's edge, its hidden state and its material ID.

### [Class FaceChange](#)

This class represents the notion of a mesh edit Face Change, which changes the visibiliy of the face's edges, its hidden state and/or its material ID.

### [Class FaceSmooth](#)

This class represents the notion of the edit mesh Face Smooth operation. This updates the smoothing group infomation in the face.

Similar to the above classes, but used for topological operations in the Edit Mesh modifier and Editable mesh are the classes below:

### [Class MeshDeltaUser](#) and [Class MeshDeltaUserData](#)

Both Edit Mesh and Editable Mesh have a current "state", which can be modified by MeshDelta objects. In Editable Mesh, this "state" is an actual

mesh, while in Edit Mesh, this is one MeshDelta per LocalModData. These are the two new classes which provide a standard interface to these.

#### [Class VDataDelta](#)

VDataDelta is simply a way for a MeshDelta to keep track of per-vertex information.

#### [Class MapDelta](#)

This class that represents some kind of change to a mesh map. This “delta” can include changes in map vertices and/or faces.

#### [Class MeshTempData](#)

This is a class for caching winged edge lists, face adjacency lists, face and edge clusters, vertex normals, and other derived data about a mesh.

#### [Class MeshChamferData](#)

This is a class to maintain chamfer information between several **MeshDelta** methods.

#### [Class EdgeClusterList](#)

This is a list of edge "clusters" for a given mesh. A typical application would be in Edit(able) Mesh, where the user has selected a two separate groups of edges on different parts of the mesh and wants to extrude them both, or rotate both around their local centers. Each "cluster" is a contiguous group of selected edges.

#### [Class UVWMapper](#)

Prior to release 3.0, developers could implement the **Object::ApplyUVWMap()** in their objects, but they haven't had access to the algorithm 3ds max uses to turn the mapping types (MAP\_BOX, MAP\_PLANE, etc) into an actual vertex-to-mapping-coordinate function. Now this is available as a class, **UVWMapper**. The mesh **ApplyUVWMap()** method has been changed to take advantage of this.

The **IMeshSelect** and **IMeshSelectData** classes have been extended to support the Editable Mesh and Edit Mesh modifier (not just the Mesh Select modifier).

#### [Class IMeshSelect](#)

This class provides access to the Editable Mesh object, Edit Mesh modifier and Mesh Select modifier selection-level data as well as a selection-change-notification method.

### [Class IMeshSelectData](#)

This class may be used to get and set the vertex, face, and edge selection state of the Edit Mesh or Mesh Select modifier.

## **New Interface Classes**

The following new interface classes provide access to various objects, modifiers, controllers etc.

[Class ISplineSelect.](#)

[Class ISplineSelectData.](#)

[Class ISplineOps.](#)

[Class IPatchSelect.](#)

[Class IPatchSelectData.](#)

[Class IPatchOps.](#)

[Template Class IFFDMod.](#)

# Required Changes To Geometric Objects

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

## Overview

This section discusses the changes to Geometric Objects. Note that not all these changes are required of every object. It depends on the type of object and whether the default behavior is undesirable. A brief description of the new or changed methods is provided here. Please refer to the reference section for the method for details.

## Mapping Methods

The new multiple mapping methods are for object which support mapping channels. These methods describe how many mapping channels are available and how many are currently used. Both are from [Class Object](#).

```
virtual int NumMapChannels();
```

```
virtual int NumMapsUsed();
```

## Counting Face and Vertex Quantities

There are many places in 3ds max that compute face and vertex counts of objects in the scene. For instance the object properties dialog, summary info, the polygon counter utility, and on file saving to write that info into the file. This was previously done by doing a **ConvertToType()** to a **TriObject** representation and counting up the faces and vertices. For certain object types, for instance NURBS, this could be very expensive since the code would need to re-tessellate the whole object. This would also consume a great deal of memory. There is now a new method in the **Object** class used to compute the number of faces and vertices in the mesh representation of the **Object**. This is:

```
virtual BOOL PolygonCount(TimeValue t, int& numFaces, int& numVerts);
```

Rather than doing a **ConvertToType()** many objects can simply iterate over their cached mesh to compute this. Geometric objects that represent themselves as a Mesh can also compute this quickly. Therefore developers of Geometric

objects that are mesh-based should implement this method. See [Class Object](#) for the details.

There is also a global function developers can call to get this information from any geometric object. It is:

```
void GetPolygonCount(TimeValue t, Object* pObj, int& numFaces, int& numVerts);
```

## Multiple Parametric Surface Access

Methods have been added for accessing multiple parametric surfaces within an object. Previously, only one surface within each object could be accessed. The new methods are:

```
virtual int NumSurfaces(TimeValue t);
```

```
virtual Point3 GetSurfacePoint(TimeValue t, int surface, float u, float v, Interval &iv);
```

```
virtual void SurfaceClosed(TimeValue t, int surface, BOOL &uClosed, BOOL &vClosed);
```

Please see [Class Object](#) for details.

## Access to Selected Point Information

Geometric Object which support the selection of points (verticies, control points, etc) can make the selection data available to other plug-ins via two new methods of [Class Object](#). These are:

```
virtual BOOL IsPointSelected(int i);
```

```
virtual float PointSelection(int i);
```

## Aborting Generating a Mesh

Geometric Object which implement **GeomObject::GetRenderMesh()** may wish to check for aborted renderings while they generate the render mesh. Some plug-ins do extensive calculations inside this method and to allow 3ds max to respond more rapidly to user abort requests these calculations should be stopped as soon as possible. By calling a methods of the **View** class,

**CheckForRenderAbort()**, inside of **GetRenderMesh()**, 3ds max can be made to feel all the more responsive. See [Class GeomObject](#) and [Class View](#) for

details.

## **IMeshSelectData Change**

The return types of the **GetVertSel()**, **GetFaceSel()** and **GetEdgeSel()** methods of the [Class IMeshSelectData](#) class have changed. They no longer return a reference to a **BitArray** but rather a **BitArray** itself.

## **Supporting AutoGrid**

AutoGrid allows for on-the-fly creation of a construction plane during another object's creation mode. This results in objects aligning to one another during creation. Objects which handle their own creation will need to add some code to their creation proc. To make it work you can do the following in the creation proc.

```
...  
case MOUSE_FREEMOVE:  
    vpt->TrackImplicitGrid(m);  
...  
case MOUSE_POINT:  
    if (point == 0)  
        vpt>CommitImplicitGrid(m, flag );  
...  
if( "returning" CREATE_STOP)  
    vpt>ReleaseImplicitGrid();
```

See [AutoGrid Related Methods](#) in class **ViewExp** for information on these methods.

## Required Changes To Particles

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### New Particle Methods

New methods have been added to the [SimpleParticle](#) class to set the position, velocity and age for an individual particle:

```
void SetParticlePosition(TimeValue t, int i, Point3 pos);
```

```
void SetParticleVelocity(TimeValue t, int i, Point3 vel);
```

```
void SetParticleAge(TimeValue t, int i, TimeValue age);
```

### Inter-Particle Collision

The 3ds max 3.0 SUPRPRTS particle systems support Inter Particle Collision (IPC). IPC is not built into the 3ds max core, or even into PARTICLE.DLL.

Rather, it's implemented at a relatively high level in SUPRPRTS. Developers of particle systems that want to support IPC will have to implement their own IPC system using the 3ds max code as an example. Also, see the

**CheckCollision()** method of [Class CollisionObject](#) which has two new parameter for IPC support.. The source code for SUPRPTRS is in the directory **\MAXSDK\SAMPLES\OBJECTS\PARTICLES**.

## Required Changes To Helper Objects

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

Any **ConstObject**-based classes will need to have their **GetSnaps()** method changed to return a **Point3**, not a **Point3&**.

## Required Changes To Light Plug-Ins

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### Atmospheres and Effects Rollup Support

Light plug-ins should use the new 'Atmospheres & Effects' rollup in their user interface. This is a new rollout that lets you assign Atmosphere and Render Effects to a selected light from the command panel. Note that you can do the same thing from within the Environment dialog or the Render Effects dialog, but this provides the controls for the light you're working on, and lets you easily see, just by selecting a light in the scene, if that light has an effect assigned to it.. See [Class Interface](#) and [Class Atmospheric](#) for details.

### Several New Methods of LightObject

There are a number of new light methods. See [Class LightObject](#) to review them.

### New Shadow Generator Plug-Ins

Lights may now have a corresponding shadow generator if desired. See the [What's New in the MAX 3.0 SDK](#) section discussing the new Shadow Generator Plug-in type or see [Class ShadowType](#).

## Required Changes To Atmospheric Plug-ins

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### Atmospheric Gizmos

Developers may want to use the Atmospheres & Effects rollup which appears in some of the core 3ds max plug-ins. This is a new rollout that appears only in the Modify panel. (It does not appear when you create the atmosphere gizmo.) The items in the new rollout let you assign Atmosphere and Render Effects to a selected atmosphere gizmo from the command panel. Note that you can do the same thing from within the Environment dialog or the Render Effects dialog, but this provides the controls for the gizmo you're working on and lets you easily see, just by selecting a gizmo in the scene, if that gizmo has an effect assigned to it.

See [Class Interface](#) and [Class Atmospheric](#) for the required methods.

### Handling Undo for Adding and Deleting Gizmo References

Atmosphere plug-ins need to implement undo for the adding and deleting of gizmo references.

Below is a summary of what needs to be done:

Implement the **SpecialFX** methods **AppendGizmo()** and **DeleteGizmo()** to handle the undo. See [Class SpecialFX](#). There are helper classes for this purpose. See [Class AppendGizmoRestore](#) and [Class DeleteGizmoRestore](#).

Anywhere in the plug-in where a gizmo is deleted, it should call **DeleteGizmo()** so the undo will work uniformly. The **DeleteGizmo()** calls should be surrounded with **theHold.Begin** and **theHold.Accept**.

**AppendGizmo()** calls (for instance inside pick procs) should be surrounded by **theHold.Begin** and **theHold.Accept**.

Use code in **NotifyRefChanged()** to update any list box controls, where changes can happen due to an Undo. For example:

```
case REFMSG_SUBANIM_STRUCTURE_CHANGED:
case REFMSG_NODE_NAMECHANGE:
if (dlg) dlg->UpdateNames();
break;
```

## Required Changes To Snap Plug-Ins

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

Snap plug-ins are now required to implement a **Osnap::Category()** method. Previously, if this method returned NULL, the snap was loaded into the default category. This is no longer true. Snaps which return NULL from this method will not be registered with the system. See [Class Osnap](#).

## Required Changes To Space Warps

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

There is a new method used as a way of identifying the 'parent' Deflector for a CollisionObject available to a particle system. This must be implemented by all Deflectors. It returns the object pointer to the Deflector from which the Collision object is derived. See **GetSWObject()** in [Class CollisionObject](#).

## Required Changes To File Import Plug-Ins

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

File Import Plug-Ins have a single new method used to control how Zoom Extents is handled after a scene import takes place. See the method **ZoomExtents()** in [Class SceneImport](#).

## Required Changes To File Export Plug-Ins

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

File Export Plug-Ins have a new parameter in the method **DoExport()** to support the export of only the selected nodes. There is also a related new method called **SupportsOptions()**. See [Class SceneExport](#).

Developers should also consider the implications of the new multiple texture mapping channels present in 3ds max 3.0. These will have an impact on what needs to be done to export a faithful visual representation of the scene. For more information see the Advanced Topics section on [Working with Meshes](#) as well as the section on **Multiple Mapping Channels** in the [What's New in the MAX 3.0 SDK](#) section.

## Required Changes To Plug-In Materials

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### Change to BuildMaps

The method **MtlBase::BuildMaps()** has had its return value behaviour altered. If this method now returns zero the render is cancelled (previously rendering would continue).

### Maximum Pixel Size

The maximum pixel size preference has been removed from the UI and thus the following methods were removed from the **Interface** class.

**virtual float GetRendMaxPixelSize()=0;**

**virtual void SetRendMaxPixelSize(float s)=0;**

There are new methods in [Class RenderGlobalContext](#) to retrieve the filter and the filter size. These are **GetAAFilterKernel()** and **GetAAFilterSize()**.

### New Return Value from MapSlotType

There is a new value to be returned by the method **MtlBase::MapSlotType()** to identify slots that hold displacement maps. This is **MAPSLOT\_DISPLACEMENT**. Developers should modify their **MapSlotType()** method to return this new value for the displacement channel.

### API Modifications to Handle 3D Bump Maps

The following changes are needed to correct a problem with 3D bump maps when objects in the scene are scaled. This scaling results in a very noticeable inappropriate change in the bumping effect.

The problem is occurring when the bump perturbation vectors must be transformed from object space to camera space, so they are oriented correctly as the object rotates. If the object has been scaled, this transformation causes the perturbation vectors to be scale also, which amplifies the bump effect. What is needed is a way to rotate the perturbation vectors so they are correctly oriented

in space, without scaling them. To do a new method has been added to [Class ShadeContext](#).

**ShadeContext::VectorFromNoScale(const Point3& p, RefFrame ifrom);**

There is also a complimentary new method **VectorToNoScale()** added for completeness.

**ShadeContext::VectorToNoScale(const Point3& p, RefFrame ito);**

The coding ramifications of this are as follows:

- Any plug-in that creates a shade context in which materials are evaluated needs to implement these two new functions (**VectorFromNoScale()** and **VectorToNoScale()**). This applies to plug-in renderers and materials.
- A change is required related to loading old files by plug-in materials which support bump maps. This change is required since if nothing is done to modify old files, bump maps on scaled objects will not render as they did previously (the bump amount for scaled-up objects will become much less). When loading old files, the Standard material uses a post-load callback that looks to see if the node that references the material is scaled up, and if so, multiplies the bump amount by a similar factor. Note that if the material is used by more than one node, there's nothing that can be done to make them all render the same if they have different scales. The same is true if a node is scaled non-uniformly. A postload callback similar to the one the Standard material uses will have to be used by plug-in materials which support the use of bump maps.

# Required Changes To NURBS Related Plug-Ins

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

## Overview

This section discusses the changes to the NURBS API that affects plug-ins. Note that not all these changes are required of every NURBS related plug-in. A brief description of the new or changed methods is provided here. Please refer to the reference section for the methods for details.

## New NURBS Texture Surface API

The API for NURBS Texture Surfaces has changes significantly to reflect the underlying change. This reflects a significant architectural change and will mean that users of the API will need to rebuild this functionality. See the subsection **Texture Mapping** in [Working with NURBS](#) for a list of the new or revised classes.

## Replaced or Eliminated Methods

Two methods have been removed from [Class NURBSExtrudeSurface](#). These are:

**void SetEVec(TimeValue t, Point3& evec);**

**Point3& GetEVec(TimeValue t);**

They are replaced by the methods:

**void SetAxis(TimeValue t, Matrix3& ray);**

**Matrix3& GetAxis(TimeValue t);**

## Tessellation Related Changes

The tessellation methods have changed in this release. This impacts several areas such as [Class NURBSSet](#) and [Class TessApprox](#). There are also some new tessellation related global functions available. See [Working with NURBS](#) to review these.

## Material IDs

Material IDs have been added to NURBS curves. [Class NURBSCurve](#) now has a new data member **int mMatID**.

## Required Changes To Textures Plug-In

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### Using an XYZGen or UVGen

All 2D textures that use UVGen or otherwise select mapping channels need to implement the method **MtlBase::LocalMappingsRequired()**

Here is a typical implementation:

```
void LocalMappingsRequired(int subMtlNum, BitArray &  
mapreq, BitArray &bumpreq) {  
    uvGen->MappingsRequired(subMtlNum,mapreq,bumpreq);  
}
```

All 3D Textures that use the XYZGen to put up a coordinates rollup must now implement the new **MtlBase::LocalMappingsRequired()** method

An example is shown below:

```
void LocalMappingsRequired(int subMtlNum, BitArray &  
mapreq,BitArray &bumpreq) {  
    xyzGen->MappingsRequired(subMtlNum,mapreq,bumpreq);  
}
```

Materials that need bump mapping for certain channels also need to implement this function so they can set the bits in **bumpreq** for those channels.

### GetUVWSource Return Values

The method **Texmap::GetUVWSource()** returns a value indicating where to get the texture vertices. One of the options has changed meaning. The value **UVWSRC\_EXPLICIT2** has been changed to always mean the vertex color channel.

### New GetMapChannel Method

There is a new method in Texmap related to the new mapping channels, **GetMapChannel()**. This method returns the map channel being used when

**GetUVWSource()** returns **UVWSRC\_EXPLICIT**.

## List of Material Requirement Flags

The old material requirement flags returned from the **MtlBase::Requirements()** method **MTLREQ\_UV**, **MTLREQ\_UV2**, **MTLREQ\_BUMPUV**, and **MTLREQ\_BUMPUV2** are still recognized by the renderer for backwards compatibility, but they will probably go away at some time. See [Class MtlBase](#) for details.

## Maximum Pixel Size

The maximum pixel size preference has been removed from the UI and thus the following methods were removed from the **Interface** class.

**virtual float GetRendMaxPixelSize()=0;**

**virtual void SetRendMaxPixelSize(float s)=0;**

There are new methods in [Class RenderGlobalContext](#) to retrieve the filter and the filter size. These are **GetAAFilterKernel()** and **GetAAFilterSize()**.

## API Modifications to Handle 3D Bump Maps

The following changes are needed to correct a problem with 3D bump maps when objects in the scene are scaled. This scaling results in a very noticeable inappropriate change in the bumping effect.

The problem is occurring when the bump perturbation vectors must be transformed from object space to camera space, so they are oriented correctly as the object rotates. If the object has been scaled, this transformation causes the perturbation vectors to be scale also, which amplifies the bump effect. What is needed is a way to rotate the perturbation vectors so they are correctly oriented in space, without scaling them. To do a new method has been added to [Class ShadeContext](#).

**ShadeContext::VectorFromNoScale(const Point3& p, RefFrame ifrom);**

There is also a complimentary new method **VectorToNoScale()** added for completeness.

**ShadeContext::VectorToNoScale(const Point3& p, RefFrame ito);**

The coding ramifications of this are as follows:

All 3D Textures need to replace calls to **ShadeContext::VectorFrom()** with calls to **ShadeContext::VectorFromNoScale()**.

Here's an example of the use of **VectorFromNoScale()** in an **EvalNormalPerturb()** function:

```
Point3 Marble::EvalNormalPerturb(ShadeContext& sc) {  
  float del,d;  
  Point3 p,dp;  
  if (!sc.doMaps) return Point3(0,0,0);  
  if (gbufID) sc.SetGBufferID(gbufID);  
  xyzGen->GetXYZ(sc,p,dp);  
  if (size==0.0f) size=.0001f;  
  p *= FACT/size;  
  
  d = MarbleFunc(p);  
  del = 20.0f;  
  Point3 np;  
  
  Point3 M[3];  
  xyzGen->GetBumpDP(sc,M);  
  np.x = (MarbleFunc(p+del*M[0]) - d)/del;  
  np.y = (MarbleFunc(p+del*M[1]) - d)/del;  
  np.z = (MarbleFunc(p+del*M[2]) - d)/del;  
  
  np *= 100.0f;  
  return sc.VectorFromNoScale(np,REF_OBJECT);  
  }
```

## Required Changes To Renderer Plug-In

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### Render Change to Allow View-Oriented Lights

The 3ds max renderer has been modified so that if a `DefaultLight` is passed into **Renderer::Open** with a transformation matrix that is all zeros, the renderer will interpret this to mean that on each frame it should create a light located at the view point, pointing in the view direction. This allows the implementation of the new viewport 1-light option so that it tracks the camera during an animated camera move. See [Class Renderer](#).

### Class, Parameter and Data Member Changes Affecting Renderers

The **ShadowBuffer** and **ShadowQuadTree** classes have been removed. Shadows are a new plug-in type in release 3.0. See [Class ShadowType](#).

The following parameters were moved from [Class RendParams](#) to [Class FrameRendParams](#), so they can be varied frame to frame.

**int regxmin,regxmax;**

**int regymin,regymax;**

Added these data members to [Class FrameRendParams](#):

**Point2 blowupCenter;**

**Point2 blowupFactor;**

Thus you can now control the horizontal and vertical blowup factors independently.

Added the following method to `RenderInstace`.

**CastsShadowsFrom(ObjLightDesc &lt)**

Added several new data members to [RenderGlobalContext](#).

**BOOL force2Side;**

**float wire\_thick;**

**Color globalLightLevel;**

Added a new data member to the [ObjLightDesc](#) class.

**int renderNumber;**

Added a parameter to the **ObjLightDesc::Update()** method:

**RenderGlobalContext \*rgc**

Two methods from [RenderInstance](#) have been removed:

**virtual int NumShadLights()=0;**

**virtual LightDesc \*ShadLight(int n)=0;**

These methods from [RendContext](#) have been removed.

**virtual ShadowBuffer\* NewShadowBuffer() const=0;**

**virtual ShadowQuadTree\* NewShadowQuadTree() const=0;**

## Default Light Changes

The default lights passed to the renderer no longer have the global light level included. This must be multiplied in in the implementation of the **::Update()** method of the default light in the renderer (using the **RendContext::GlobalLightLevel()** method passed in).

## API Modifications to Handle 3D Bump Maps

The following changes are needed to correct a problem with 3D bump maps when objects in the scene are scaled. This scaling results in a very noticeable inappropriate change in the bumping effect.

The problem is occurring when the bump perturbation vectors must be transformed from object space to camera space, so they are oriented correctly as the object rotates. If the object has been scaled, this transformation causes the perturbation vectors to be scale also, which amplifies the bump effect. What is needed is a way to rotate the perturbation vectors so they are correctly oriented in space, without scaling them. To do a new method has been added to [Class ShadeContext](#).

**ShadeContext::VectorFromNoScale(const Point3& p, RefFrame ifrom);**

There is also a complimentary new method **VectorToNoScale()** added for completeness.

**ShadeContext::VectorToNoScale(const Point3& p, RefFrame ito);**

The coding ramifications of this are as follows: Any plug-in that creates a shade

context in which materials are evaluated needs to implement these two new functions (**VectorFromNoScale()** and **VectorToNoScale()**). This applies to plug-in renderers and materials.

## Required Changes To Bitmap Loader/Saver Plug-Ins

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

BitmapIO plug-in have two changed methods. The methods

**BitmapIO::GetOutputPixels()** and

**BitmapIO::GetDitheredOutputPixels()** now have a new parameter

**BOOL preMultAlpha** which defaults to TRUE. Setting it to FALSE will

cause pixels with non-premultiplied alpha to be returned. See [Class BitmapIO](#).

## Required Changes To Controller Plug-Ins

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### Deleting Controllers

A new "Delete Controller" button has been added to the Track View toolbar that is enabled when one or more delete-able tracks are selected. This method allows a plug-in to indicate to the Track View that one or more of its sub-controllers are delete-able. This provides a way for the user to delete node sub-controllers such as the visibility track, "Image Motion Blur Multiplier", "Object Motion Blur On/Off", etc.

There are two new methods of the **Control** class to support this. Each has a default implementation so overriding it is optional but developers may wish to do so.

These methods are **CanDeleteSubAnim()** and **DeleteSubAnim()**. See [Class Control](#) for details.

## Required Changes To Shape Objects

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### Material IDs in Shape Related Classes

New in the 3ds max 3.0 SDK is material ID access to various shape-related classes. This results in various methods being added as shown below:

ShapeObject:

```
virtual MtlID GetMatID(TimeValue t, int curve, int piece) { return 0; }
```

SimpleShape:

```
virtual MtlID GetMatID(TimeValue t, int curve, int piece);
```

SimpleSpline:

```
MtlID GetMatID(TimeValue t, int curve, int piece);
```

BezierShape:

```
MtlID GetMatID(int poly, int piece);
```

PolyShape:

```
MtlID GetMatID(int poly, int piece);
```

PolyLine:

```
MtlID GetMatID(int segment);
```

SplineShape:

```
MtlID GetMatID(TimeValue t, int curve, int piece);
```

LinearShape:

```
MtlID GetMatID(TimeValue t, int curve, int piece);
```

Note that ShapeObject has a default implementation so it is not necessary to implement it if the shape class does not support material IDs.

### Number of Vertices in ShapeObjects

There is a new method of **ShapeObject** called **NumberOfVertices()**. This method is used by the Summary Info and Object Properties dialogs to inform the user how many vertices or CVs are in the object. Developers may wish to

implement this method.

## **Attach Shape Method**

Developers implementing the **ShapeObject::AttachShape()** method need to add the **weldEnds** and **weldThreshold** parameters. If any endpoints of the curves in the shape being attached are within the threshold distance to endpoints of an existing curve, and the weld flag is TRUE, they should be welded.

## **RescaleWorldUnits Implementations**

Objects derived from the **ShapeObject** class which have **RescaleWorldUnits** methods implemented need to call the **ShapeObject::RescaleWorldUnits** method. See [Class ShapeObject](#).

## Required Changes To Patches

See Also: [What's New in the MAX 3.0 SDK](#), [Required Changes to MAX 2.0 Plug-Ins for MAX 3.0](#).

### Overview

This section discusses some of the changes to the patch related classes in the SDK. The main changes relate to the fact that Patch objects became editable Patches objects and that patches now support more than two mapping channels.

### Mapping Channel Changes

**PatchMesh** objects no longer use a hard coded **PATCH\_TEXTURE\_CHANNELS** number of maps, but rather a dynamically allocated amount up to **MAX\_MESHMAPS** (100). The following data members have been changed from fixed-size arrays to tables:

// **Texture Coord assignment**

**Tab<int> numTVerts;**

**Tab<UVVert \*> tVerts;**

**Tab<TVPatch \*> tvPatches;**

As before, the patch is considered to have a map on channel 'i' if **tvPatches[i]** is not NULL. There are **PatchMesh::numPatches** elements in each tvPatches array, and **numTVerts[i]** UVVerts in array **tVerts[i]**.

**IMPORTANT NOTE:** previously, developers could only have 2 map channels. Map channel 0 represented traditional texture verts and map channel 1 represented vertex colors. As with meshes, this has been reversed for R3: channel 0 in these tables and methods is the Vertex Color channel, and channels 1-99 are map channels 1-99.

To maintain the number of maps, the following method has been introduced:

**void setNumMaps(int ct, BOOL keep=TRUE);**

If keep is FALSE, all the existing maps will be freed; otherwise, only those over 'ct' will be freed. New map channels are initialized with NULL arrays and 0 numTVerts. (**numTVerts**, **tVerts**, and **tvPatches** are all set to a size of 'ct'.)

However, most developers won't have to worry about calling this method, since it's called by **setNumTVertsChannel()**, which has been used to set the number of mapping verts in each channel.

There are now two versions of most methods handling maps in patches, to cope with a necessary change in map indexing between 3ds max 2.5 and 3.0 (This problem comes up in patches, but not meshes, since patches already had a channel indexing scheme in methods like **setNumTVertsChannel()**. Mesh only had methods like **setNumTVerts()**, with no channel argument.)

Old TV/VC methods are given with "TV" in the name. For these methods, channel 0 is the original map channel, while any nonzero channel is vertex colors. If no index is given, the original map channel is assumed. This is all consistent with usage in the 2.5 SDK.

New methods are given with "Map" in the name. For these methods, channel 0 is the VC channel, as it is now in Object. Channel 1 is the original map channel, while 2-99 are the new channels. The plan here is that the "Map" methods will be used in future, the "TV" methods will be considered obsolete.

There is one incompatibility with the 2.5 SDK, in that the public data members numTVerts, tVerts, and tvPatches all now use the new indexing scheme, and are now Tab<> tables instead of simple arrays as noted above.

### **Links to Changed Classes:**

[PatchObject](#)

[Class PatchMesh](#)

[Class Patch](#)

[Class PatchVec](#)

[Class PatchVert](#)

[Class PatchEdge](#)

[Class TVPatch](#)

# What's New in the MAX 4.0 SDK

See Also: [Required Changes to MAX 3.x Plug-Ins for MAX 4.0.](#)

## Overview

This section provides a general overview of the new capabilities of the 3ds max 4.0 API. New plug-in types, new classes, and new methods to existing classes have been added. In some cases new parameters to existing methods have been added as well.

Any new classes in the SDK begin their Description section with the line:

**This class is available in release 4.0 and later only**

Any new methods in a class begin:

**This method is available in release 4.0 and later only**

Newly added parameters to existing methods begin:

**This parameter is available in release 4.0 and later only**

Newly added data members begin:

**This data member is available in release 4.0 and later only**

## New Plug-In Types

The following are the new plug-in types introduced in the 3ds max 4.0 SDK. Each has a link to the base class for the creation of that plug-in type.

### Manipulator Plug-Ins

One of the new UI features for R4 is a system of direct manipulation plug-ins called manipulators. These plug-ins are a special kind of helper object that can be used to modify parameters of Objects, Modifiers, INodes and Controllers by using gizmos in the 3D viewports. See [Class Manipulator](#) or [Class SimpleManipulator](#) for more details.

The relevant classes are located in

**\MAXSDK\INCLUDE\MANIPULATOR.H**

A sample on how to write these plugins is located in

**\MAXSDK\SAMPLES\HOWTO\BENDMANIP**

### Tone Operator Plug-Ins

In order to provide better quality images a tone operator plug-in has been

added to 3ds max and the SDK. This is used to convert lighting energy values to RGB values for rendering and display after shading is performed. See [Class ToneOperator](#) for more information on this topic.

The relevant classes are located in `\MAXSDK\INCLUDE\TONEOP.H`

### **Render Element Plug-Ins**

A new plug-in type called Render Elements has been added. A render element isolates a component of a rendering like specular, diffuse, emission, etc. and outputs it to either a separate bitmap or a multi-channel RPF or RLA file. The new plug-in type communicates with the ShadeContext, retrieving color data stored there by the shaders, compositing and saving the resulting image to file. See [Render Elements](#) for more details.

The relevant classes are located in

`\MAXSDK\INCLUDE\RENDERELEMENTS.H`

### **Multi-Pass Render Camera Effect**

The multipass camera effect allows modification of viewpoints & view directions or time for each pass of a multipass rendering. Algorithms such as Depth of Field, Scene motion blur can be implemented using multipass techniques. See [Class ImultiPassCameraEffect for more details](#).

The relevant classes are located in `\MAXSDK\INCLUDE\OBJECT.H`

### **Inverse Kinematics, Solver Plugins.**

IK solvers are now plug-able. The IK system will recognize a plug-in solver by putting it in the solver list wherever it appears. For more information refer to the section on [Inverse Kinematics](#).

The API of the IK solver is defined in

`\MAXSDK\INCLUDE\IKSOLVER.H, IKHIERARCHY.H,`

`IIKSYS.H, and IKCTRL.H.` For an example, please look at

`\MAXSDK\SAMPLES\IKSOLVERS\IKLIMB`

## **New Features By Category**

### **Rendering**

#### **Interactive Reshading**

The relevant classes are located in `\MAXSDK\INCLUDE\MTL.H`. See [Class IReshading](#).

## Interactive Rendering

With the likelihood of evolving rendering and shading techniques which are going to be markedly different from what is being used now, the 3ds max SDK provides the infrastructure to support interactive rendering. Since renderers are a plugin to 3ds max and since each renderer has a different set of resources and capabilities the interactive rendering and shading API is made into as much a general API and independent as possible. See [Class InteractiveRender](#).

## Image Motion Blur

The new motion blur effect plug-in, MotionBlur.dlv, now contains code that used to be part of the renderer, and uses function publishing to allow the renderer to call functions to do image blur. For details on how this is done see [Class IMBOps](#). Sample code is available in

**\MAXSDK\SAMPLES\RENDER\RENDEREFFECT\MOTIONBLUI**

Also, Image Motion Blur has been improved so that objects that are moving behind transparent object will blur fairly realistically. See [Class GBuffer](#) for details. The relevant classes are located in **\MAXSDK\INCLUDE\GBUF.H, IMBLUR.H**

## Vertex Colors from a Non-Internal Color Array

A new method of the Mesh class allows the source data for vertex colors to come from other than the default, internal vertex color array (also known as map channel 0). [Class Mesh \(Color Per Vertex Methods\)](#) . An overview of this can be found in the Advanced Topics section [Working With Meshes](#). The relevant classes are located in **\MAXSDK\INCLUDE\MESH.H**.

## Material Surface Evaluation

Material surface evaluation allows extraction of more precise color information from a material. The material interface now exports a set of functions in order to enable "interactive display" to show the material correctly in the viewport. Note that these functions do not take into consideration texture maps and other shading parameters. Relevant changes were made to [Class Mtl](#) and [Class Texmap](#).

## Multiple Map Display in the Viewports

The code in the Mesh class and associated classes has been modified so that multiple textures are available for display in the interactive viewport. See [Class Mesh](#). Plug-In Materials have several new methods to implement related to this. These are **SetupGfxMultiMaps** and **SupportsMultiMapsInViewport** from [Class MtlBase](#). The relevant classes are located in `\MAXSDK\INCLUDE\MESH.H, IMTL.H`.

## New Material Handlers for New Particle Systems

This interface allows particle systems to support material per face and particle ID per face. This class may also be used by plug-ins that want to support this capability. See [Class IChkMtlAPI](#). The relevant classes are located in `\MAXSDK\INCLUDE\CHKMTLAPI.H`

## New Network Rendering API

This is the new API for network rendering. The relevant classes are located in `\MAXSDK\INCLUDE\MAXNET_MANAGER.H`. See the Advanced Topics section [Network Rendering](#) for details.

## A New Interface to Allow Multiple Image Viewers (VFBs)

Up until this point, the VFB has been tightly coupled with the 3ds max Bitmap implementation. This new abstract interface will let multiple image viewer implementations be used to view Bitmaps.

See [Class IImageViewer](#) for details.

The relevant classes are located in

`\MAXSDK\INCLUDE\IIMAGEVIEWER.H`

## User Interface

### Action Tables

New in R4 is a class called ActionTable that is a generalization of the ShortcutTable class added for R3. ActionTable contains operations that can be tied to various UI elements in 3ds max including toolbar buttons, keyboard

shortcuts and menu items. See the Advanced Topics section [UI Customization](#) for details. The relevant classes are located in **\MAXSDK\INCLUDE\ACTIONABLE.H, MAXAPI.H, PLUGAPI.H.**

## **Function Publishing**

See the Advanced Topics section on [Function Publishing System](#) for details.

## **Track Bar Update**

The Track Bar API has been updated to support the new functionality provided in R4. See [Class ITrackBar](#). The relevant classes are located in **\MAXSDK\INCLUDE\MAXAPI.H**

## **Menu Manager**

This is an interface to the menu customization system. See [Class IMenuManager](#). The relevant classes are located in **\MAXSDK\INCLUDE\IMENUMAN.H**

## **Color Manager**

The color manager allows plug-ins to register colors that can be customized and saves / restores them from a file. See [Class IColorManager](#). The relevant classes are located in **\MAXSDK\INCLUDE\ICOLORMAN.H**

## **OLE DragAndDrop Manager**

The DragAndDrop manager is a new component in R4 that provides a general framework for handling OLE-based drag-and-drop in 3ds max. The prime motivations for adding this new component were to accommodate the new iDrop™ tool for dragging 3ds max content from the web and to provide a general way to implement drop targets for the new dropScript capability in 3ds max.

The new manager is a generalization of the VIZ R3 OLE-based DragAndDrop manager that has been ported into 3ds max 4. It is exposed through a public API in the 3ds max SDK and the Function Publishing system, supports an extensible

set of clipboard formats and drop types and permits individual 3ds max windows to specialize drop handling. This allows you to support new iDrop™ file types, such as dropScripts, fold the 3ds max internal drag-and-drop into the same system that handles external drag-and-drop, and allows custom drops to various interesting windows in 3ds max, such as the material editor, schematic view, the new modifier stack, etc. or other 3rd-party windows. See [Class IDragAndDropMgr](#) for more details.

The relevant classes are located in  
**\MAXSDK\INCLUDE\IDRAGANDDROP.H**

## **Dialog Position / Size Manager API**

This feature may be used to increase UI consistency by having all major floating dialogs remember their previous position and size, like Track View and Medit do now. This is done by calling methods to save the size / position to the CUI file.

## **R4 Parameter Wiring System**

The parameter wiring features are encapsulated in an interface class that allows you to directly control the various parameter wiring operations. See [Class IParamWireMgr](#) for more information.

The relevant classes are located in **\MAXSDK\INCLUDE\IPARAMWIRE.H**

## **New Mouse Manager Functionality**

Newly added to the mouse manager is the ability to add a callback function that lets a command mode get all the raw mouse windows messages from a viewport. This way developers can avoid using the **MouseCallBack::proc()** method, which does a lot of filtering on the messages that it sends.

The relevant classes are located in  
**\MAXSDK\INCLUDE\MOUSEMAN.H**

## **Modeling**

### **Node Handles**

This is a new unique ID for each node in the scene. See [Class INode](#) and [Class Interface](#), in particular the methods **Interface::GetINodeByHandle()** and **INode::GetHandle()**. The relevant classes are located in **\MAXSDK\INCLUDE\INODE.H** and **MAXAPI.H**

## Node Display Callback

The feature enables a plug-in developer to register a callback function that gets called whenever a node needs to be drawn. The developer can hide the standard mesh and display their own image for the node. This allows one to control a node's display without being part of the modifier stack. See [Class NodeDisplayCallback](#).

The relevant classes are located in **\MAXSDK\INCLUDE\NODEDISP.H**

## New Callback Classes

Two new callbacks are available. One is used to filter the display of nodes in the scene by category. See [Class DisplayFilterCallback](#). The other is to filter the selection of nodes. See [Class SelectFilterCallback](#). Also see the methods in **Class Interface** associated with activating these callbacks ([Interface callback methods](#)).

## Extension Object -- New Extension Channel

This feature enhances the geometry pipeline by allowing a developer to add a custom object to the pipeline object. This object will get notified whenever something in the pipeline changes. This extends the flexibility of the Modifier Stack by being able to implement a combination of a custom data notification system. An example of this in use is a game developer who wants to indicate when a certain object becomes invalid for export to their game engine. By inserting an Extension Object into the pipeline they can accomplish this, by constantly checking the structure of the object and displaying wrong faces/vertices etc. in the viewport. See [Class XTObject](#).

Sample code is available in:

**\MAXSDK\SAMPLES\EXTENSIONCHANNELOBJECTS\XMODII**

The relevant classes are located in **\MAXSDK\INCLUDE\CHANNELS.H**, **XTOBJECT.H**

## Geometry Pipeline Enumeration

There is a new class and several new global functions that can be used to enumerate the geometry pipeline.

For details see [Class GeomPipelineEnumProc](#).

## Vertex Alpha

This feature adds map channels to support "Vertex Alpha" and "Vertex Illumination" in all object types and gives the user editing controls for these channels in Editable Objects and Edit Modifiers. See [Class Mesh](#). The relevant prototypes are located in: `\INCLUDE\MESH.H`

## Custom Face Data Storage

This feature allows a Mesh object to support up to 100 custom data channels for Faces. Each channel provides an array to store data as defined by the developer. The types of data will often be standard items such as floats and ints, however, pointers to more sophisticated objects may be stored as well. The number of elements in the array for each channel will correspond to the number of faces in the Mesh.

The main advantages and differences over the current per-vertex-data support are: face-data channels are identified by a `Class_ID` rather than an integer; face-data channels can store objects of a user defined type, not just float; 3rd party developers can derive their own face-data channel from **Class**

**IFaceDataChannel**. The Mesh notifies all its face-data channel objects of events and operations related to faces, so 3rd parties can really manage their per-face data the way they want, they have a lot of flexibility. This is different from the current per-vertex-data because the vertex data channels are entirely exposed to and managed by the Mesh. Note that only Meshes support the **IFaceDataMgr** interface. See [Class IFaceDataChannel](#), [Class IFaceDataMgr](#).

## Object Conversion Between Types

There is a new class that allows developers to convert between their object types and the native 3ds max types. See [Class ObjectConverter](#) for details.

## Conversion Modifiers

There are now modifiers to perform explicit object type conversions in the pipeline. Specifically, modifiers which will operate on all object types:

Turn to Mesh

Turn to Patch

Turn to Poly

There are new global functions for doing this conversion. See the [Explicit Conversion Functions](#).

The relevant prototypes are located in **\MAXSDK\INCLUDE\PATCH.H** and **POLYOBJ.H**

## **SubObjType API**

There is a new API for working with Sub-Objects. The new Stack View in the command panel uses this API. There are several new classes related to it. See [Class BaseObject](#) [the Class BaseObject page has the wrong font on its title], [Class ISubObjType](#), [Class GenSubObjType](#), [Class MaxIcon](#). The relevant classes are located in **\MAXSDK\INCLUDE\OBJECT.H**.

## **Polymesh Object**

See [Class PolyObject](#).

The relevant classes are located in **\MAXSDK\INCLUDE\POLYOBJ.H**

## **Editable Poly**

See [Class EPoly](#).

The relevant classes are located in **\MAXSDK\INCLUDE\IEPOLY.H**

## **Patches**

A large number of new methods and data members have been added to the various classes relating to patches. Details on these new methods and data members can be found by browsing through the classes listed below and looking for the R4 specific availability notes. In this section a shortlist of general additions and changes will be listed.

Class **PatchMesh** has gained a number of methods dealing with unifying and flipping normals, welding and cloning, vertex weights, and linear and curved

mapping.

Class **PatchObject** has gained a number of methods dealing with setting the Show End Results option and handling the status thereof, getting and setting vertex and patch colors, selecting vertices by color and illumination, creating shapes from edges, welding and breaking, flipping and unifying normals, deleting selected patches, and changing the mapping type of patches.

Class **Patch** has gained a few methods dealing with linear and curved mapping, finding edges and vertices based on vertex indices.

Class **TVPatch** has gained some extra data members and methods dealing with patch handles and interiors.

Class **PatchVert** will now deal with an array of edges used by a vertex.

See [Class PatchMesh](#), [Class PatchObject](#), [Class Patch](#), [Class PatchEdge](#), [Class TVPatch](#), [Class PatchVert](#)

See [Required Changed from R3.x to R4](#).

The relevant classes are located in `\MAXSDK\INCLUDE\PATCH.H` and `PATCHOBJ.H`

## Animation

### New Interface Classes for Various Controllers

There are several new classes for accessing the parameters of new or modified controllers in R4. These are listed below. The relevant classes are located in `\MAXSDK\INCLUDE\ISTDPLUG.H`

#### **LookAt Constraint**

This is the new Look At Controller (rotation only as opposed to the previous LookAt Transform controller). An interface to its parameters is available in [Class ILookAtConstRotation](#).

#### **Orientation Constraint**

The Orientation Constraint matches the orientation of an object to its target without affecting its position. See [Class IOrientConstRotation](#) for the interface details.

#### **Position Constraint**

A 3ds max user can use the Position Constraint to make an object move to

and be coincident with another target object. The Position Constraint can also move an object to the position of the weighted average of several different targets. See [Class IPosConstPosition](#) for access to this controller.

### **Path Constraint**

The old path controller has been modified and renamed the Path Constraint. For an interface to the parameters of this controller see [Class IPathPosition](#).

## **New System Utilities**

### **New Notification Codes**

Numerous new notification codes have been added for R4. See [Structure NotifyInfo](#).

The relevant structure is located in `\MAXSDK\INCLUDE\notify.h`

### **New Reference Messages**

There are a few new reference message that may be used. See [List of Reference Messages](#).

The relevant defines are located in `\MAXSDK\INCLUDE\ref.h`

### **Scanline Pager template**

This new template is used to minimize memory usage when rendering. Bitmaps (for instance textures and the frame buffer) are normally accessed on a scanline basis when rendering. Currently they are fully loaded into memory when rendering starts. This class is used to try to keep only the active block of scanlines in memory and the rest of the bitmap on disk – making the extra RAM available for geometry and other memory buffers.

The relevant classes are located in

`\MAXSDK\INCLUDE\scanlinepagemgr.h`

### **New Miscellaneous Global Functions**

See [List of Miscellaneous Utility Functions](#) for a few new global functions related to checking the number of processors in the system, determining the version of Windows running, and checking the screen width and height.

## **Custom Attributes**

An unlimited number of custom attributes (e.g. parameters) can be added dynamically to individual objects. These can be very useful for game developers and/or other project specific data. The SDK provides a way to define and add these custom attributes to objects in the scene.

A sample on how to write these plugins is located in  
`\MAXSDK\SAMPLESHOWTO\CUSTATTRIBUTIL`

See [Class CustAttrib](#) for details on custom attributes.

## **Particle Collision Encapsulation**

The SDK adds new support for particle collision detection whereby other systems such as Flex and MaxScript can also benefit from this more open architecture. These collision API's provide an interface to determine if a particle hits a surface. Three basic collision detection classes are provided, a planar, spherical, and a mesh deflection class.

See [Class ICollision](#) for more information.

## **Random Number Generator**

A new and improved random number generator class has been provided in order to deal with the shortcomings of the standard random number generation functions.

See [Class RandGenerator](#) for more information.

## **Support for Hardware Shaders**

3ds max 4.0 supports custom hardware vertex shaders and pixel shaders using Direct-3D, right in the graphics viewports. The primary aim of this support is to enable a 1:1 relation between what users see in the viewport and the content they are creating for other Direct-3D enabled applications.

See [Class ID3DGraphicsWindow](#), [Class IDX8PixelShader](#), and [Class IDX8VertexShader](#) for more information.

# What's New in the MAX 4.0 SDK

See Also: [What's New in the MAX 4.0 SDK](#).

## Overview

This section provides general information on the changes required to all plug-ins to get them running in 3ds max 4.0. It also provides links to topics that discuss the specific changes for many affected plug-in types. Some of these changes are required while others are optional but advantageous.

## Microsoft Platform SDK

It is recommended to obtain and install the Microsoft Platform SDK which many of the examples depend on and require in order to compile properly. In order to ensure proper compilation of plugins and voiding any linker errors, please make note of the following;

- · When compiling plugins using the command line, ensure that the path environment variables for library and include files precede the paths set by Microsoft Visual Studio.
- · When compiling plugins using the Visual Studio IDE, ensure that the path configuration (options menu-> directories) for library and include files are listed as the top entries.

Note: The Microsoft Platform SDK can be obtained through the MSDN web site.

## Patches

As of R4.0 a vector can now be used by more than two patches. Previously the [Class PatchVec](#) kept an index to two patches sharing the vector as an integer array of 2 elements. This has changed to an IntTab. While this would most likely not invalidate any code it is advisable to keep in mind that there could now be more than 2 patches sharing a vector.

Some structural changes have been made to the [Class PatchEdge](#) that will

require some changes in the plugin code. Previous to R4.0 the patches that used an edge were kept in **int patch1** and **int patch2**. These two integers have been replaced with an integer table, **IntTab patches** because edges can now be used by more than two patches. The plugin code should be changed to reflect these changes.

In order to facilitate the new topology tracking code for the Edit Patch modifier a new class has been introduced, [Class PatchTVert](#). This new class has been integrated with the class **PatchMesh** and brings with it a number of changes. The previous table of **UVVerts** has been replaced with a table of **PatchTVert's**. And a number of methods in [Class PatchMesh](#) have been altered to take advantage of this new class. These are; **mapVerts()**, **getTVertChannel()**, **getTVert()**, **getTVertPtrChannel()**, **getTVertPtr()**, **getMapVert()**, and **getMapVertPtr()**.

Slight changes have been made to the **PatchObject::DoBevel()** and **PatchObject::DoExtrude()** methods. Both now take a **Timevalue** as a parameter. Please make sure you adapt your code according to this new behavior if you are using these methods.

## Splines

A few changes were made to the [Class ShapeObject](#) methods in order to support the new animated renderable shapes. These changes should be observed and adjusted in the plugin's code. **ShapeObject::SetThickness(float t)** now accepts a TimeValue prior to the thickness parameter, and was changed to; **SetThickness(TimeValue t, float thick)**. The **ShapeObject::GetThickness()** method now accepts a TimeValue and an Interval parameter, and was changed to; **SetThickness(TimeValue t)**. Currently the previous syntax of these methods will get or set the thickness at frame 0, however Sparks Developers who are using the debug build should take note of the fact that the debug build will raise an assertion to indicate that the old syntax is obsolete and that the new syntax should be used.

## User Interface Notes

Developers should no longer use the Win32 calls **GetSysColor()** and **GetSysColorBrush()**. Due to the new customizable color schemes in 3ds max 4 developers should now use the encapsulated functions provided by the SDK; **GetCustSysColor()** and **GetCustSysColorBrush()**. These will take the same parameters as the Win32 functions but will retrieve the color information from the custom color database.

## Keyboard Shortcuts

The previous keyboard shortcut system has been replaced by an enhanced system. The new system uses what are called Action Tables. These tables unify all the actions that can be assigned to keyboard shortcuts, CUI buttons, right-click menus and main menu. See [Class ActionTable](#) for more information.

Some of these methods have the same name as ActionItem methods, and a more complete

## Sending WM\_COMMAND messages to 3ds max

If you are sending WM\_COMMAND messages to 3ds max' main window you will now need to make sure that **HIWORD(wParam)** is 1, not 0. If you ever get an assertion failure in **MenuManager::ExecuteAction()**, then it is probably because a WM\_COMMAND message was sent with a 0 HIWORD.

## Registering window classes for custom controls.

Previously every DLL had to initialize the window classes through a call to **InitCustomControls()** in DllMain. This caused a substantial hit on the available system resources in Windows 9x for every plug-in that was loaded. The classes are now globally registered for the whole process. Plug-ins no longer have to call **InitCustomControls()** at startup - but it doesn't hurt since once the classes have been registered they will not be registered again.

## Additions to Class BaseObject

Objects and modifiers, that support subobjects have to overload the two new methods **NumSubObjTypes()** and **GetSubObjType()**. and return a class derived from **ISubObjType** in **GetSubObjType()**. Developers can use the **GenSubObjType** for convenience. If the parameter passed into

**GetSubObjType** is -1, the system requests a **ISubObjType**, for the current SubObjectLevel that flows up the modifier stack. If the subobject selection of the modifier or base object does not affect the subobj selection that flows up the stack, the method must return NULL.

Note that this replaces the way subobjects were handled prior to this release. RegisterSubObjectLevel() is obsolete and should no longer be used. Instead the **NumSubObjTypes()** and **GetSubObjType()** methods should be used. For more information see [Class ISubObjType](#) and [Class BaseObject](#).

## Rendering

Plugin renderers should implement **ShadeContext::BumpBasisVectors()**.

**Interface::DoExclusionListDialog()** accepts an ExclList instead of a NameTab. This will cause the compile to break for plugin renders, which need to be modified to use the new node lists. There is a method

**Interface::GetINodeByHandle(handle)** which can be used to get the node from the handles in the ExclList. Additionally

**Interface::ConvertNameTabToExclList()** allows you to convert name tables to the new exclusion lists. A number of methods now return an ExclList instead of a NameTab, these are; **ObjLightDesc::GetExclList()**, **LightObject::GetExclList()**, **GenLight::GetExclusionList()**, **GenLight::SetExclusionList()**.

In Class IllumParams a few members were replaced; **ULONG shFlags** and **mtlFlags**, and **Point3 N**, and **V**. The class now contains pointers to the shader and material from which flags and other information can be retrieved. The shading normal **N** was a copy of the normal in the shadeContext, which is now provided to shaders and renderElements as well as IllumParams, thus **sc.Normal()** should be used to get the bumped shading normal.

## Programmatically collapsing the stack

Developers, who are collapsing the stack programmatically, should pay attention to the new Extension Channel ([Class XTCObject](#)) and the [Class BaseObject](#) methods **NotifyPreCollapse()** and **NotifyPostCollapse()**. These methods will be called by the collapse code. It will give the modifier (or BaseObject); that adds an XTC object to the stack the possibility to apply a modifier, that inserts

these XTC objects onto the stack after the collapse. Through this mechanism, the XTC will survive a stack collapse. In case these method are not called, the Extension Channel Objects will by default be copied as well, since they are part of the object in the wsCache. However, they won't survive a save/load operation. Please see the [Class XTCObject](#) for more details.

## **Parameter change in Animatable::GetSystemNodes()**

The method Animatable:GetSystemNodes() has its signature changed from previous versions and could potentially require an adjustment of plugin code. If you use the previous signature while having it overridden, the compiler will let this pass unnoticed because it assumes this is a new function (while the actual method itself is empty and hidden in the super class). As a result 3ds max will call the method with the new signature and, because it wasn't overridden, it will call the default implementation as defined

## **Trivial changes to samplers**

Samplers had to be generalized for use with Render Effects. All samplers had to be rewritten slightly, as will other plug-in samplers. The change is quite trivial, however.

## **Bump vectors in ShadeContext**

A problem in the way bump basis vectors were being calculated has been dealt with.

The code was trying to deduce a single set of 3 vectors for U,V,and W, and this led to problems. Since the bump vectors are only used in pairs ( UV,VW, WU) a better approach turns out to be to compute 2 vectors for the specific pair of axes. This required some new API calls. There is a new method in [Class ShadeContext](#), **BumpBasisVectors()**, which should replace **DpDUVW()** over time. **DpDUVW()** is left in so as not to break a lot of plugins.If **BumpBasisVectors()** returns 1, that is assumed to mean it is implemented, and it will be used instead of **DpDUVW()**.

## **Systemwide Clone implementation change**

A call to **BaseClone(this, newob,remap)** has been added to all plugins in 3ds

max, that implement the **Clone** method. **BaseClone** is a virtual member function of [Class ReferenceTarget](#). This method allows base classes to copy their data into a new object created by clone. All overwrites of **BaseClone** must call their base classes implementation. The implementation in [Class ReferenceTarget](#) is copying the CustAttrib objects into the newly created object.

All plugins that implement a **Clone** method have to call **BaseClone** with the old and the new object as parameters. The ordering in regards to when this method is called is unimportant. It obviously has to be called after the cloned object is created. The **BaseClone** method has to check for the cases of from, or to to be NULL and from and to to be equal. It is important to mention for **BaseClone**, that all overrides have to call their BaseClass' implementation. With **ReferenceTarget::BaseClone** we have a central method, that gets called for all clone operations. This allows us to add notifications etc. for cloning.

## Code Cleaning for 64 bit

**INT\_PTR**, **DWORD\_PTR** and other **\_PTR** types are new types defined by Microsoft to support UDM (Unified Data Model), These are polymorphic types that are defined to be of the type they indicate but large enough to hold a pointer. For example an int is 32 bits on both Win32 and Win64, so if you cast a pointer to an int it will loose the high bits. To solve this they added the **\_PTR** types so if you need to do pointer arithmetic (with int types) you need to have the type **INT\_PTR** type and the type will be large enough to hold a pointer (i.e. 32 bits on Win32 and 64 bits on Win64). Mesh methods, which dealt with the geometry pipeline had their channel parameters revised from unsigned along to **ULONG\_PTR**.

## Changed to GetLocalTMComponents

The API of the **GetLocalTMComponents()** method has been changed. In the current version, the parent matrix is not given as a pointer to the parent node, but as an indirect matrix representation.

## Changed support for multi-meshes in the renderer

Two important methods are added to the [Class GeomObject](#). These are

**GetMultipleRenderMesh()** and **GetMultipleRenderMeshTM()**. Please refer to the class documentation for more information.

## Changes to **GBufReader** to allow writing and bug fix

Two new methods have been added to [Class GBufReader](#):

**ModifyChannelData()** and **ModifyAllData()**. These methods allow values in the current layer to be written. This may seem strange, writing data from the reader, but developers asked for the capability of writing to the already created gbuffer, and it is much simpler to add this capability to the **GBufReader** than to **GBufWriter**, which is designed to construct gbuffers from scratch, not modify existing ones.

## Transparency and opacity maps in viewports

For the purposes of implementing transparency and opacity maps in the viewports, a new flag has been defined that indicates if transparency is required in the viewport: **MTLREQ\_TRANSP\_IN\_VP**. See the [List of Material Requirement Flags](#) for more details.

## Removal of hidden vertices

MNMeshes used to support "hidden vertices". These would be vertices that exist "in the middle" of faces, taking part in their triangulation but considered more a feature of the face than vertices in their own right. These vertices were useful in the past, when MNMeshes were constantly being converted to and from Meshes. We needed to remove them for PolyMeshes to be an efficient pipeline object; keeping track of them was drastically increasing the size of every face, whether it had such vertices or not.

## Changes in Class **MeshDelta**

In class **MeshDelta**, the data member **fCreate** was switched from **Tab<Face>** to **Tab<FaceCreate>**. In conjunction with this a new method **FCreate()** was added. More information can be found in [Class MeshDelta](#) and [Class FaceCreate](#).

## Changes in Class MNMapFace

Data members **hdeg**, and **hvtx** have been removed. The constructor **MNMapFace()** now only accepts one argument, **SetAlloc()** and **SetSize()** also accept only one argument, **HInsert()** and **HDelete()** have been removed, and **MNDebugPrint()** no longer accepts any arguments. More details can be found in [Class MNMapFace](#).

## Changes in Class MNFace

Data members **hdeg** and **hvtx** have been removed. The constructor **MNFace()** no longer accepts a second argument. **SetAlloc()** no longer accepts a second argument, **HInsert()** and **HDelete()** have been removed and **MNDebugPrint()** now only accepts one argument. The "**tri**" array was replaced by the new "**diag**" array. The diag array's allocated size is always  $(dalloc-3)*2$ . If  $dalloc==3$  (triangle), this pointer is NULL. The method "**TriVert**" was removed. Use **GetTriangles()** to access this sort of information. More information can be found in [Class MNFace](#).

## Changes in Class MNMesh

The methods **HVNum()** and **KillUnusedHiddenVerts()** have been removed. There is no longer an **MNM\_SL\_TRI** selection level. Methods **FindExternalTriangulation()** and **BestConvexTriangulation()** were changed to **FindDiagonals()** and **BestConvexDiagonals()**, respectively.

The [Class MNMap](#) data member, "**m**", became private. The accessor method, **M**, remains public, and now takes values of -1 (**MAP\_SHADING**) and -2 (**MAP\_ALPHA**) as well as 0-99. **SetMapSeamFlags()**, which previously took an argument of (mp=-1), where -1 meant "set map seam flags based on all maps", had to be split into two methods as follows: void **SetMapSeamFlags()** and **SetMapSeamFlags(int mp)**;

Several methods that accept a selection level as an argument were changed to take an MNMesh selection level (such as **MNM\_SL\_VERTEX**) instead of the old Mesh selection levels they used to take (such as **MESH\_VERTEX**). This has affected the following methods in class MNMesh; **GetBorder()**, **TargetVertsBySelection()**, **TargetEdgesBySelection()**,

**TargetFacesBySelection(), Slice(), PropagateComponentFlags(), SabinDoo(), SabinDooVert(), and AndersonDo().**

## **Change internal triangulation storage from triangles to diagonals for Class MNMesh**

This is another space-saving measure. We always store a particular triangulation, or way to convert the polygon into triangles, in each MNFace. This is necessary to preserve face orientation when converting to and from regular meshes, or to allow users to explicitly edit the triangulation in R4. We used to store this info as triangles, based on the face's "internal" indices. For instance, if we're dealing with an octagon, the vertices would be numbered 0,1,...7, corresponding to the order in the "vtx" array. However, this was very inefficient, as it required 3 int's to store the triangulation of a triangle (0,1,2), or 6 int's for a quad (0,1,2,2,3,0), even though there's only 1 way to "triangulate" a triangle, and only 2 possibilities for a quad. (The other would be (0,1,3,1,2,3).) Now we're storing diagonals.

For instance, in a quad, we really only need to know whether the diagonal goes from 0 to 2 or from 1 to 3. We can list all the diagonals of an n-sided polygon with  $(n-3)*2$  ints - 0 for a triangle, 2 for a quad, 4 for a pentagon, etc. (The old scheme took  $(n-2)*3$  ints - 3 for a triangle, 6 for a quad, 9 for a pentagon.) This is probably not totally optimized, but it strikes a nice balance between memory usage and ease of use.

## **CUI Image Lists**

CUI Image lists are no longer used. Everything now goes through the icon manager and as such the **GetDefaultImageList()** and **AddToImageList()** methods were removed from the Class CUIFrameMgr.

## **Modifier Sets and Categories**

The categories shown in the modifiers drop-down list are modifier sets which you can configure using the Configure Modifier Sets dialog. Modifier Set information is stored in the 3dsmax.ini file. In order to add your own custom Modifier Set to have your plugin showing under its own heading you can create the heading with the appropriate class ID in the 3dsmax.ini file.

## **Note about member alignment**

Please make sure that when you save data from your plugin you save individual data members using a chunk ID instead of saving the image of a class. Saving (and loading) a class image puts you at risk of running into member alignment problems and as such could potentially corrupt saved files. File IO would be put further at risk when you keep Intel's IA-64 architecture in mind which depends on member alignment. What you should not do is outlined in the following example when loading a class image;

**iload->Read(&myclass, sizeof(MyClass), &ab);**

Once you change the class in such a way that it affects the data size you run the risk of having to support different versions, file IO incompatibility, and member alignment issues.

## What's New in the MAX 5.0 SDK

See Also [Required Changes to Certain MAX 4.0 Plug-Ins](#), [New and Unsupported Changes](#)

### Overview

This section provides general descriptions and links to the associated classes/interfaces of new feature functionality in 3ds max 5.0.

## Global Illumination

**Description:** 3ds MAX 5.0 currently has two solutions for global illumination: Radiosity, and Light Tracer.

**Documentation:** [Class RadiosityEffect](#), [Class ISpecularCompositeShader](#), [Class IEmissionColor](#), [Class INodeGIProperties](#)

## Spline IK

**Description:** Refers to the new modifier which, when assigned to a spline (or a NURBS) curve, generates a certain number of helper objects attached to the knots of the curve. This interface provides access to most of this new functionality

**Documentation:** [Class ISplineIKControl](#)

## Texture Baking

**Description:** Provides the interfaces for creating Texture Baking plugins in 3ds max. The texture baking plugins are controlled through the Maxscript “Render To Texture”. There are methods available in this class that provide the ability to produce a dynamic UI in Maxscript.

**Documentation:** [Class MaxBakeElement](#)

## Viewport Shaders

**Description:** These interfaces give you the opportunity for you to write your own DX Hardware Shaders and have a mechanism for displaying them in the MAX viewports

**Documentation:** [Class IViewportShaderManager](#) , [Class IDXDataBridge](#) , [Class IHardwareMaterial](#)

## Skin Posing

**Description:** This interface provides access for getting and setting a special,non-animated, transformation pose.

**Documentation:** [Class RotationValue](#) , [Class ISkinPose](#)

## UVUnwrap

**Description:** This new interface allows for Normal, Flatten, and Unfold mapping. You can bring them up through a dialog or through a script command.

**Documentation:** [Class IUnwrapMod2](#)

## Node Exposure

**Description:** This interface provides the ability for a node to define whether it is visible in any of max's dialog boxes. This interface will be extended and used by more of 3ds max's core utilities, but **currently ONLY TrackView and the Select Object/HideObject dialog box use this interface.** By default this interface is not available through the default nodes, it needs to be added.

**Documentation:** [Class INodeExposure](#)

## Node Layer Properties

**Description:** This class defines an interface for accessing a node's global illumination properties.

**Documentation:** [Class INodeLayerProperties](#)

## Assemblies

**Description:** Assemblies combine the collective nature of a group with the programmability of a system. These interface classes allow control over which nodes belong to an assembly and which is the head. Methods are also provided to control the accessibility of members.

**Documentation:** [Class IAssembly](#) , [Class IAssembly2](#)

## Edit Normals Modifier

**Description:** This interface provides access to the functionality in the Edit Normals modifier, which allows you to edit three categories of normals: Unspecified, Specified, and Explicit. Please see the associated documentation for a full description of this modifier's functionality.

**Documentation:** [Class IEditNormalsMod](#)

## New and Unsupported SDK Changes for MAX 5.0

See Also: [What's New in the MAX 5.0 SDK](#) , [Required Changes to Certain MAX 4.0 Plug-Ins](#)

### Overview

This section provides general information on the SDK additions which are accessible yet unsupported in 3ds max 5.0.

## **Set Key**

**Discreet will not be supporting Set Key functionality for 3<sup>rd</sup> party plugins.**

## Painter Interface

**Discreet will not be supporting Painter Interface functionality for 3<sup>rd</sup> party plugins.** However, you may view extensive comments by the developer of this feature in the IPainterInterface header file located here:  
**[/samples/painterinterface/include/ipainterinterface.h](#)**

## **BackBurner2 – Network Rendering**

**Backburner is the new Network rendering system for 3dsmax r5. It provides a unified model for both 3dsmax and Combustion rendering.**

Although the API for backburner has not changed from the API available in 3dsmax 4, some changes have been introduced with the way backburner interacts with the 3dsmax application. Some DLLs required for the API have a very close connection to Backburner, so if backburner is not present or is not loaded these DLLs may fail to load, causing plugin failure.

## Required Changes to Certain MAX 4.x Plug-ins for MAX 5.0

See Also: [What's New in the MAX 5.0 SDK](#), [New and Unsupported Changes](#)

### Overview

This section provides general information on the changes required to a small list of plug-in types to get them running in 3ds max 5.0.

## **IK Solver**

**All plug-ins implementing IK Solver functionality will require a recompile to run in 3ds MAX 5.0.**

## **TrackView Utilities**

**All TrackView Utility plug-ins will require a recompile to run in 3ds MAX 5.0.**

## What's New in the MAX 6.0 SDK

### Overview

In an effort to provide the 3ds max SDK development community with up-to-date SDK changes and documentation updates, we're moving our SDK updates presentation to a more dynamic, online location. With this move, we can assure the development community that all changes and updates will be reflected as soon as they occur, without the requirement of downloading a new 3ds max SDK documentation build.

Please click the link below to view the current SDK and documentation updates page online:

**Internet Address**

[http://sparks.discreet.com/knowledgebase/techdocs/sdk\\_change\\_update](http://sparks.discreet.com/knowledgebase/techdocs/sdk_change_update)

# Game Export Interface

## Overview

The Game Export Interface has been developed to greatly simplify data export from 3ds max. The following is a list of associated components and materials for this plugin:

1. The .lib file is located in /maxsdk/lib/IGame.lib.
2. The help documentation is located as IGame.chm in the /maxsdk/help/ directory.
3. The sample exporter can be found through /samples/impexp/IGameExporter/ directory.

## Internet Address

More Game Exporter related materials can be located online through the Sparks developer website: <http://sparks.discreet.com/downloads/>

# **VC7 memory wrapper for 3ds max**

## Overview

To accommodate for development of 3ds max 5.1 plug-ins using the VC7 development environment, Discreet has developed a system to address the incompatibility between the VC7 memory subsystem and 3ds max, which is VC6 native. Developers can employ this solution by including the `MAX_Mem.h` header in their plug-in project, which serves as a replacement for `malloc.h`, `new.h` and `crtDBG.h`. The result is to redirect memory allocation and de-allocation functions into equivalents implemented in the 3ds max core, operating on a VC6 native heap.

You can find this header in the `maxsdk/include` directory.

## Limitations

The memory wrapper is necessary only in situations where memory allocated by a plug-in is later deallocated by 3ds max, or vice versa. In these cases, both operations must occur on the same VC6 heap. Any other memory used by a plug-in may reside on a VC7 heap, and in those cases the use of MAX\_Mem is not necessary.

A limited set of memory allocation functions is supported under MAX\_Mem. These include malloc(), free(), new, delete (including array forms) and heap inspector functions like \_heapchk(). Functions which implicitly allocate memory, such as strcpy(), are not generally supported.

## **Expanding Memory Wrapper**

Developers wishing to expand upon the memory wrapper may create their own VC6-compiled DLL which provides access to the necessary functions. The functions could be accessed using a preprocessor macro, which defines the original function name as a call to the VC6 exported function, or, in the case of `new()` and `delete()`, by using a function overload. These are only suggestions, and other schemes are possible at the developer's discretion.

## Further Information

For more information regarding VC7 compatibility with 3ds max, see the document “VC7 plugins for 3dsmax 5 and 5.1” in the Sparks Knowledgebase:

[http://sparks.discreet.com/search/index.cfm?url=/knowledgebase/techdocs/searchable/Techdoc\\_3dsmaxVC7/3dsmax](http://sparks.discreet.com/search/index.cfm?url=/knowledgebase/techdocs/searchable/Techdoc_3dsmaxVC7/3dsmax)

# Sparks Developer Program

## Introduction

[sparks.discreet.com](http://sparks.discreet.com) is the Discreet developer portal for 3ds MAX 4.0 and above. Join sparks to receive the highest level of 3ds MAX SDK support!

## How to Join Sparks

Apply for sparks directing your browser to the following internet address: <http://sparks.discreet.com/app/appmain.cfm>. For more information, either read on or go here: <http://sparks.discreet.com/info/fohome.cfm>

## Sparks Membership Tiers

**The sparks program provides three levels of access to 3ds MAX SDK developers. These tiers are Public, Standard, and Premium.**

**Public** users of sparks.discreet.com have access to the [Sparks Developer Knowledgebase](#), the [Sparks Webboard](#), and public [Sparks Downloads](#). There is no cost for access to these support channels.

**Standard** users of sparks.discreet.com have access to everything public, as well as additional downloads in the form of the latest 3ds MAX Debug Builds, and other private samples. *The cost for this membership is \$400 US.*

**Premium** users of sparks.discreet.com are entitled to everything public and standard, as well as access to the Sparks Incident Reporting system which allows direct, web-based, contact with the Discreet Sparks SDK Support Team. *The cost for this membership is \$1200 US.*

*Please note that all prices are subject to change.*

## Sparks Developer Support Channels

**The following methods of support are available through the sparks.discreet.com domain. Please follow the links below for brief description of each topic, as well as its associated internet address.**

[Sparks Developer Knowledgebase](#)

[Sparks Message Archives](#)

[Sparks Documentation Updates](#)

[Sparks Webboard](#)

[Sparks Downloads](#)

[Sparks Incident Reporting](#)

## **Further Information**

For more information regarding the sparks program, please direct your browser to the following internet address:

<http://sparks.discreet.com/info/infohome.cfm> .

# Sparks Developer Knowledgebase

See Also: [Sparks Developer Program](#)

## Overview

### **See Also:**

The sparks developer knowledgebase offers search-engine access to a plethora of 3ds MAX SDK support materials, including techdocs/labs, solutions, support threads, and the documentation contained in this help file. *Please note that additional support materials are available for Premium members.*

**The Sparks Developer Knowledgebase is available to the public, with additional support materials provided to Premium sparks members only.**

## Internet Address

<http://sparks.discreet.com/search>

# Sparks Message Archives

See Also: [Sparks Developer Program](#)

## Overview

The sparks message archives provide an offline archive of all threads in the Sparks Webboard. You can download the most recent archive from the public downloads or you can find a less recent version of the file (sparks\_archive.chm) in the help directory in the maxsdk directory. Note that as long as that file resides in that directory, all searches in this this file (sdk.chm) will have access to those threads.

**The Sparks Message Archive is available to the public.**

## Internet Address

[http://sparks.discreet.com/downloads/downloadshome.cfm?  
f=2&wf;\\_id=56](http://sparks.discreet.com/downloads/downloadshome.cfm?f=2&wf;_id=56)

# Sparks Documentation Updates

See Also: [Sparks Developer Program](#)

## Overview

The sparks team is constantly updating the help documentation for the MAX development community. To access the latest compiled help file, please follow the go the [Sparks Downloads Area](http://sparks.discreet.com/downloads/) (<http://sparks.discreet.com/downloads/>) or visit the [Sparks Developer Knowledgebase](http://sparks.discreet.com/search) (<http://sparks.discreet.com/search>) and click the “knowledgebase” link in the upper-left of the site.

**New for 3ds max 6, we are offering an online SDK and documentation update list.** To view this list, please go here: [http://sparks.discreet.com/knowledgebase/techdocs/sdk\\_change\\_update](http://sparks.discreet.com/knowledgebase/techdocs/sdk_change_update).  
[to](#)

**All Sparks Documentation Updates are available to the public.**

# Sparks Webboard

See Also: [Sparks Developer Program](#)

## Overview

The sparks developer webboard offers a community source for all items related to the MAX SDK. Join the forum and participate in discussions with fellow MAX SDK developers

**The Sparks Developer Knowledgebase is available to the public.**

## Internet Address

<http://sparks.discreet.com/webboard/wbpx.dll/~maxsdk>

# Sparks Downloads

See Also: [Sparks Developer Program](#)

## Overview

The sparks downloads area provides the latest samples, documentation, debug builds (for members only), and other support materials for the MAX development community.

**The Sparks Downloads area is available to the public, with additional downloads such as MAX Debug Builds provided for sparks members only.**

## Internet Address

**Public Downloads:**

<http://sparks.discreet.com/downloads/downloadshome.cfm>

**Private Downloads:** You must be logged in to the sparks members area and click 'downloads' on the left menu to access additional downloads such as 3ds MAX Debug Build(s) and private samples.

# Sparks Incident Reporting

See Also: [Sparks Developer Program](#)

## Overview

The sparks incident reporting system allows direct correspondence with the Sparks SDK Support Team. As a Premium member, you'll be able to file questions and receive professional responses to your development queries straight from the experts at Discreet.

**The Sparks Incident Reporting system is available Premium sparks members only.**

## **Internet Address**

**You must be logged in to the sparks members area and click “SDK Support’ on the left menu to access this resource.**

## Advanced Topics

See Also: [Must Read Sections by Plug-In Type](#), [Must Read Sections for All Developers](#).

This section presents detailed information about key concepts in using the 3ds max APIs. This section contains more explanation than is available in the reference section. The topics presented here cross class boundaries and are instead based on specific concepts developers need to understand when creating 3ds max plug-ins.

For a list of these topics important to read when creating a specific plug-in type see [Must Read Sections by Plug-In Type](#). For those sections which should be read by everyone see [Must Read Sections for All Developers](#).

Action Tables

[Anti-Piracy Protection](#)

[Character Strings](#)

[Creating a New Plug-In Project](#)

[COM/DCOM Interface](#)

[Command Modes and Mouse Procs](#)

[Computing Face and Vertex Normals](#)

[Custom User Interface Controls](#)

[Custom Node Properties and App Data](#)

[Debugging](#)

[Deferred Loading of Plug-Ins](#)

[DLL/Lib Functions and Class Descriptors](#)

[External Icons](#)

[Foreground / Background Planes.](#)

[Function Publishing System](#)

[Geometry Pipeline System](#)

[Getting and Setting User Preferences](#)

[Globalization](#)

[Hit Testing](#)

[Interactive Renderer: Graphics Window](#)

[Interface Class Overview](#)

[Intervals](#)

[Inverse Kinematics](#)

[Keyboard Accelerators and Dialog Messages](#)

[Keyframe and Procedural Controller Data Access](#)  
[Loading and Saving Plug-In Data](#)  
[MAXScript SDK](#)  
[Matrix Representations of 3D Transformations](#)  
[Memory Allocation](#)  
[Modifier Stack Branching](#)  
[Network Rendering](#)  
[Nodes](#)  
[Node and Object Offset Transformations](#)  
[Object Creation Methods](#)  
[Object Modification](#)  
[Palettes](#)  
[Parameter Blocks](#)  
[Parameter Editing in the Command Panel](#)  
[Parameter Maps](#)  
[Parameter Blocks and Maps in Release 3](#)  
[Parent-Child Hierarchy](#)  
[Plug-In Configuration System](#)  
[Plug-In Directory Search Mechanism](#)  
[Plug-In Help System](#)  
[Profiling Plug-In Performance](#)  
[Read Only Plug-Ins](#)  
[References](#)  
[Render Elements](#)  
[Rotation Concepts](#)  
[RPF Files and the G-Buffer](#)  
[Schematic View](#)  
[Snapping](#)  
[Space Warp Plug-Ins](#)  
[Style Guidelines for Creating Pipeline-Friendly Meshes](#)  
[Sub-Anims](#)  
[Sub-Object Selection](#)  
[Sub-Object Coordinate Systems](#)  
[Thread Safe Plug-Ins](#)  
[Time](#)  
[Track View](#)  
[UI Customization](#)

[Units of Measurement](#)

[Undo/Redo](#)

[Working with ActionTables](#)

[Working with Bitmaps](#)

[Working with Character Strings](#)

[Working with Controllers](#)

[Working with Lights](#)

[Working with Materials and Textures](#)

[Working with Meshes](#)

[Working with NURBS](#)

[Working with Patches](#)

[Working with Shapes and Splines](#)

## Anti-Piracy Protection

See Also: [Read Only Plug-Ins](#).

Developers may wish to implement an anti-piracy protection scheme for their plug-ins. The 3ds max SDK provides a function developers may use as part of their copy protection strategy. This function retrieves the unique hardware lock ID attached to a particular machine. 3ds max itself requires the machine to have a hardware lock, and every hardware lock used with 3ds max has a unique ID associated with it. The function **HardwareLockID()** returns this ID as an unsigned integer. A developer can call this function using the code shown below:

### Prototype:

```
unsigned int HardwareLockID();
```

### Remarks:

This function returns 0 if no lock is found and the lock ID if one is found, regardless if 3ds max is in slave mode or not.

### Sample Code:

```
unsigned int hLockID = HardwareLockID()
```

Developers may key off this ID in numerous ways to authorize a plug-in to run on only a particular machine. The actual copy protection strategy employed using this lock ID is the developers responsibility. 3ds max is only responsible for insuring the integrity of the **HardwareLockID()** function. Internally, 3ds max checks the integrity of this function to prevent tampering that attempts to affect the values returned from the function.

Developers are, of course, free to implement any additional copy protection schemes as they see fit. The two basic suggested methods are to either be "tricky" about what they do to the plug-in code / functionality once it has been detected that the lock check has been tampered with or they can go to the extreme of providing an additional hardware lock for use with their own plug-ins.

In any anti-piracy / authorization strategy employed, developers should be careful how their code affects users. For example, putting up an authorization dialog when 3ds max is performing a network render is very destructive and mustn't be done. Also, plug-ins that pop up dialogs when the command panel branch is changed are also far too inconvenient for the user. Anti-Piracy dialogs should not be implemented where they will pop up from any calls within the

Class Descriptor. It may be appropriate to put lock checking code inside ClassDesc methods, but putting up dialogs that interrupts the users work flow should not be done there. The best places to show dialogs are within methods such as **CreateParamDialog()** or **BeginEditParams()**. If you have a rendering related plug-in (renderer, material, texmap, atmospheric effect, etc.) that you want licensed per CPU, it's better to use a watermark or to disable functionality, but don't disrupt the user's workflow.

Sample code using an authorization scheme can be found in **\MAXSDK\SAMPLES\MODIFIERS\TWIST.CPP**.

# Character Strings

See Also: [Class CStr](#), [Class WStr](#), [Class FilterList](#).

This section presents information developers need to know when using character strings in MAX. It discusses the meaning of the **TSTR**, **\_T**, and **TCHAR** macros that are commonly seen in the source code examples. This section also covers important substitutes for the standard C **str\*** functions, and some general functions that are available for use with character strings.

## The TSTR, \_T and TCHAR Macros

When 3ds max was designed there was originally the possibility of using UNICODE for the storage of character strings. During development it was decided that 3ds max would use MBCS (multi-byte character sets) instead. To provide the flexibility to switch between the two (perhaps some day to make it possible to also support UNICODE once it's fully supported under Windows NT), a macro is used to substitute either wide character strings (UNICODE) or single character strings (MBCS). This macro is **TSTR**. Its definition is shown below:

```
#ifndef _UNICODE  
#define TSTR WStr  
#else  
#define TSTR CStr  
#endif
```

If UNICODE is in use, the macro is defined to mean **Class WStr**. If UNICODE is not in use, **Class CStr** is used. As mentioned above, the decision was made to not go with UNICODE, and thus **TSTR** is equivalent to **CStr**.

**Class WStr** is used for wide character strings (16 bits per character). **Class CStr** is for single and double-byte character sets. DBCS **may** need 2 bytes to represent a single character, whereas UNICODE **always** uses two bytes per character. Note: Windows code often refers to "MBCS", but in reality, the only multi-byte type supported is "double-byte". I.e. Windows doesn't support any character sets that need 3 bytes for some characters.

Both **Cstr** and **WStr** provide methods and operators for calculating lengths, concatenation, sub-string operations, character searching, case conversion, comparison, and formatted writing. For additional information on these classes see [Class CStr](#) and [Class WStr](#).

Developers are encouraged to use the **TSTR** macro for string definitions, rather than going straight to **CStr**. In this way if a future version of 3ds max is developed that uses UNICODE, the text definitions will convert with a simple re-compile.

Developers will also encounter the **\_T** macro in the source code examples. Literal string definitions use **\_T** preceding the string, for example

**\_T("Parameters")**). This macro is defined in a Microsoft include file **\MSDEV\INCLUDE\TCHAR.H**. This macro will convert a string to the proper form for either UNICODE or non-UNICODE. In general, any code that is written for internationalization should use the **\_T** macro for literal string definitions.

A developer may also encounter the **TCHAR** macro in the source code. This macro will be either a **char** (8 bit) or a **wchar\_t** (16 bit UNICODE char) depending if UNICODE is in use.

## Standard String Functions from C

The proper way to deal with strings (other than the methods of the **CStr** and **WStr** classes) is to use the **\_t** version of the standard string functions. Because 3ds max uses MCBS (multi-byte character sets), the **\_t** versions sometimes map to functions other than the traditional **str\*** ones (which are only guaranteed to work with single-byte character sets). Below is a list of several common mappings from the standard Microsoft VC++ include file (usually **\MSDEV\INCLUDE\TCHAR.H**).

**strcat** maps to **\_tcscat**

**strcpy** maps to **\_tcscopy**

**strlen** maps to **\_tcslen**

**sprintf** maps to **\_stprintf**

**scanf** maps to **\_stscanf**

For example, instead of using the standard C **strcpy** function, use the one shown below:

```
TCHAR buf[64];  
_tcscopy(buf, _T("[ 0, 0, 0 ]"));
```

Developers who need to use the standard C string functions in their 3ds max code should refer to **\MSDEV\INCLUDE\TCHAR.H** to see if there is a **\_t** version of the string function defined for use.

## The GetString() Function

Developers may also see the **GetString()** functions used in the sample source code. This function is simply used to return a string from a resource library. It is defined as follows:

```
TCHAR *GetString(int id) {  
    static TCHAR buf[256];  
  
    if (hInstance)  
        return LoadString(hInstance, id, buf, sizeof(buf)) ? buf : NULL;  
    return NULL;  
}
```

This function is used so that the 3ds max program can be easily translated into other non-English languages. By loading the strings from a resource table, only the resource string table has to be updated to allow the user interface of 3ds max to appear translated into another language. For more details on using resource libraries, see the Advanced Topics section [Globalization](#).

## Summary of Rules for Using Strings

The following rules summarize what developers need to be aware of when working with characters strings.

- Use the **TSTR** string class as much as possible, since these strings automatically allocate the right amount of memory for their storage. (As an added bonus, they're also a lot easier to use than standard strings, because a lot of functionality is built into the string class.)
- If you have to use static character arrays, make sure the array allocation is adequate. Also, make sure that system limits (like 259 for path+filename) are always met or exceeded.
- Don't use **char** or **w\_char** arrays: instead use **TCHAR** as the character type.
- Don't use any standard string manipulation functions (like **strcat**, **strcpy**, **sprintf**, etc.) directly. Rather, use the character-set independent versions found in **\MSDEV\INCLUDE\TCHAR.H**. For example, use **\_tcscat** instead of **strcat**. Note that even routines like **fopen()** that take a string have character-set independent versions in **TCHAR.H**.
- Put all literal strings inside the **\_T** macro, as in **\_T("This is literal")**. The **\_T** macro will convert strings to Unicode automatically if we ever choose to compile with that enabled.
- When comparing characters, use the same macro as for strings but with single quotes. Also, if for any reason you are comparing with a slash for directory separation, make sure to also check for forward slashes as they are valid in NT and used when mounting Unix drives. For example:

```
if (my_path[x] == _T('/') || my_path[x] == _T('\\')) {  
    // do something...  
}
```

For internationalization / localization consider the following when working with strings:

- Try to avoid any string-specific manipulation (like keying off of the first letter in a string, or concatenating strings to form a longer token), since much of this won't translate easily into other languages.
- Keep all literal strings in a resource string table. If all goes well, the internationalization/localization process will consist of only editing resources, not code. In fact, the French, Italian, Spanish, German, and Kanji

versions of 3ds max all use the same code.

## Miscellaneous String Functions

The following two functions are part of the 3ds max API and are useful for breaking up filenames.

### Prototype:

```
void SplitFilename(TSTR& name,TSTR* p, TSTR* f, TSTR* e);
```

### Remarks:

Splits filename **name** into its path, filename, and extension.

### Parameters:

**TSTR& name**

The filename to split apart.

**TSTR\* p**

The path portion.

**TSTR\* f**

The filename portion.

**TSTR\* e**

The filename extension portion.

### Prototype:

```
void SplitPathFile(TSTR& name,TSTR* p, TSTR* f);
```

### Remarks:

Splits filename **name** into its path and a filename.extension.

### Parameters:

**TSTR& name**

The filename to split apart.

**TSTR\* p**

The path portion.

**TSTR\* f**

The filename AND extension portion.

### Prototype:

```
BOOL MatchPattern(TSTR &s, TSTR &ptrn, BOOL  
ignoreCase=TRUE);
```

**Remarks:**

This function is available in release 2.0 and later only.

This method is used to check if a string matches a wildcard pattern.

**Parameters:****TSTR &s**

The string to check.

**TSTR &ptrn**

The wildcard pattern. The \* character matches 1 more characters. The ? character matches a single character.

**BOOL ignoreCase=TRUE**

If TRUE the check is not case sensitive; otherwise it is case sensitive.

**Return Value:**

TRUE if the string matches the pattern; otherwise FALSE.

**Prototype:****int MaxAlphaNumComp(TCHAR \*a, TCHAR \*b);****Remarks:**

This function is available in release 2.0 and later only.

A "smart" alphanumeric compare that sorts things so that numerical suffixes come out in numerical order. This version is case sensitive.

**Parameters:****TCHAR \*a**

The first string to compare.

**TCHAR \*b**

The second string to compare.

**Return Value:**

< 0 means string **a** is less than string **b**.

= 0 means that string **a** is identical to string **b**.

> 0 means that string **a** is greater than string **b**.

**Prototype:**

**int MaxAlphaNumCompI(TCHAR \*a, TCHAR \*b);**

**Remarks:**

This function is available in release 2.0 and later only.

A "smart" alphanumeric compare that sorts things so that numerical suffixes come out in numerical order. This version is case insensitive.

**Parameters:**

**TCHAR \*a**

The first string to compare.

**TCHAR \*b**

The second string to compare.

**Return Value:**

**< 0** means string **a** is less than string **b**.

**= 0** means that string **a** is identical to string **b**.

**> 0** means that string **a** is greater than string **b**.

# Creating a New Plug-In Project

See Also: [Plug-In Directory Search Mechanism](#), [Building the Sample Files](#).

## Overview

This topic presents step by step instructions on how to create a project file for a 3ds max plug-in.

Note: The 3ds max 2.0 SDK now includes a set of AppWizards to create projects for many of the standard plug-ins. Currently it supports: Atmosphere, Anti-Aliasing Filters, Color Selector, File Import, Front End Controllers, Global Utility Plug-Ins, Image Filter / Compositor, Materials, Modifiers, Procedural Objects, Rendering Effects, Textures 2D, Samplers, Shaders, Shadow Generator, Textures 2D, Textures 3D, Track View Utility, and Utility plug-ins.

Using these wizards makes it much easier to create new projects. In addition to setting up most of the compiler / link settings they create CPP files containing the basic class methods that need to be implemented.

To install the AppWizard, copy the **\MAXSDK\HELP\SDKAPWZ.ZIP** file to the Developer Studio Template directory. For example, copy it to the **\Microsoft Visual Studio\Common\MSDev98\Template** directory. Then simply unzip the file. The next time you start VC++, from the **File / New ...** menu, you may choose **3ds max Plugin AppWizard** and follow the onscreen prompts. The AppWizard has online help available but the steps are very self explanatory.

If you want to create a plug-in project 'by hand', the process is described next.

## Beginning a New Project

To create a new 3ds max plug-in project from within Microsoft Developers Studio follow the steps below:

- 1 From the File pulldown menu choose **New...**
- 2 From the **New** dialog make sure the **Projects** tab is selected.
- 3 Select **Win32 Dynamic Link Library** from the list of project types.
- 4 In the **Location:** box enter the path to where you want the project directory created (for example **C:\MYPLUGS**).
- 5 In the **Project Name:** box enter the name for the directory and plug-in project (for example **PRJNAME**). A new directory with this name will be created as

a **sub-directory** of the directory entered in the **Location:** box. The starter project files will also use this name.

6 Press the **OK** button to exit the dialog.

7 At the prompt **What kind of DLL would you like to create?** select **A simple DLL project** and press the **Finish** button to exit the dialog. Press **OK** at the next dialog to complete creating the initial files.

Result: Developer Studio creates a new directory and puts starter **OPT, DSP, DSW, MAK,** and **NCB** files in this directory. These files are explained below

**PRJNAME.DSP** -- This is the project file used within the development environment. In previous versions of Visual C++ this file extension was .MAK. It stores the information specific to your project. There will be a separate .DSP file for each project you create. .DSP files are not compatible with NMAKE. You must export a makefile to build with NMAKE.

**PRJNAME.DSW** -- This is the workspace file used within the development environment. It organizes all the projects into a single workspace.

**PRJNAME.NCB** -- This is the No compile Browser file. It contains information generated by the parser which is used by ClassView, WizardBar, and Component Gallery. If the file is accidentally or deliberately deleted, it is automatically regenerated.

**PRJNAME.OPT** -- This is the workspace options file used within the development environment. It stores all the user options you create for your workspace, so that each time you open the project workspace it has the look and feel you want and includes any customizations you have made.

**PRJNAME.MAK** -- The project makefile.

## **Creating and Adding DEF and CPP files to the Project**

When you create a 3ds max project using the Microsoft Developer Studio, you also need the following file types:

**CPP** -- The C++ source code.

**DEF** -- The module-definition file. A module-definition file is an ASCII text file containing one or more module statements that describe various attributes of a DLL. A DLL requires a DEF file to create an import library (LIB) file and an export (EXP) file. The linker uses the import library to build any executable module that uses the DLL and uses the export file to build the DLL file.

You now need to create the **DEF** file and edit the **CPP** file. It is usually most convenient if these files reside in the same directory as the **DSP** file. In the instructions shown below it is assumed your plug-in is called **MYPLUGIN**.

**1** From the File menu choose **New. . .**

**2** From the New dialog choose the **Files** tab, check the **Add to project** checkbox, enter the C++ source code file name in the **File name:** box (for example **MYPLUGIN**) and select **C++ Source File**. This brings up a new text file window where you'll add your source code. You can begin the **CPP** file with the standard 3ds max include file. This is done by adding the following line to the **CPP** file:

```
#include "MAX.H".
```

Next you need to create the **DEF** file.

**1** From the File menu choose **New. . .**

**2** From the New dialog choose the **Files** tab, check the **Add to project** box, enter the DEF file name in the **File name:** box (for example **MYPLUGIN.DEF**), select **Text File** and press **OK**. This brings up a new text file window where you'll edit the **DEF** file text.

**3** Copy the text shown below from this help file into your new text window. This is the basic structure of a **DEF** file.

```
LIBRARY MyPlugIn
```

```
EXPORTS
```

```
LibDescription @1
```

```
LibNumberClasses @2
```

```
LibClassDesc @3
```

```
LibVersion @4
```

```
SECTIONS
```

```
.data READ WRITE
```

**4** You'll need to perform a single edit to this **DEF** file to work with your project. The only thing you need to change is the Library name. Simply change it to the name of your DLL. Do not include a file name extension. As shown above the library name is **MyPlugIn**.

### **Adding a Resource Script to the Project**

You now need to create a Resource file. The resource has an extension of RC.

**RC** -- The resource file. A resource is binary data that a resource compiler adds to an application's executable file. The data in a resource typically describes an icon, cursor, menu, dialog box, bitmap, etc. Any rollup pages or dialogs you create to implement your user interface will be stored in the RC file. The resource compiler will generate an include file which you must include in your source file.

To create the resource file follow these steps:

**1** From the File menu choose **New. . .**

**2** Choose **Resource Script**.

**3** In the **File name:** box enter your resource name (for example **MyPlugIn.RC**).

Result: A resource script window will pop up.

You now need to add a dialog template to this resource script.

**1** From the **Insert** pull down menu choose **Resource. . .**

**2** From the **Insert Resource** dialog choose **Dialog** and press **New**.

Result: A default dialog will pop up.

**1 Right Click** on the dialog box itself, and from the popup menu choose **Properties...**

**2** In the **ID:** field enter the ID for the dialog, for example **IDD\_MYPLUGIN**

If your dialog is for use in the command panel or materials editor (not a floating dialog), follow the steps below:

**3** Click on the **Styles** Tab

**4** From the **Style:** drop down list choose **Child**

**5** From the **Border:** drop down list choose **None**

**6** Close the Properties dialog box and resize the plug-in dialog frame size to a width of **108 units**. This is the standard width for the command panel in 3ds max for use in the United States. You can see the size displayed in the lower right corner of the IDE. If you are creating a material or texmap plug-in whose interface shows up in the materials editor dialog the proper width is **217 units**. If you're doing an atmospheric or renderer plug-in the correct width is **212 units**. If you are creating a dialog that will not be used in the command panel you may, of course, use any size.

Result: You have a dialog template to which you may add controls. For use in the command panel you may wish to delete the **OK** and **Cancel** buttons.

When you compile the RC file an include file is generated. The default name is

**resource.h.** This file will be included in your **CPP** source file to specify the resource ID's. To add the include file to your **CPP** file edit the **CPP** file and add the line:

```
#include "resource.h"
```

## Adding a String Table

Now you need to add a string table to your project. This allows you to put your literal strings into a table for easier translation to other languages. Rather than searching your text for literal strings you can simply change them in the string table. The sample code in the SDK uses this technique to manage strings. To add a string table, do the following:

- 1 From the **Insert** pull down menu choose **Resource...**
- 2 Choose **String Table** and press the **New** button.

Result: You have added the string table to your project.

When you want to add a literal string to your plug-in enter the value into the string table and then use the **GetString()** function to load that string. To enter the value you simply double click on an empty slot in the table and enter an ID (the ID passed to **GetString()**) and a Caption (the text). Consider the **Animatable::GetClassName()** method as an example.

Without the string table, the code would be written:

```
void GetClassName(TSTR& s) { s = _T("Ring Array"); }
```

With the string table, it would be written:

```
void GetClassName(TSTR& s) { s =  
GetString(IDS_DB_RING_ARRAY_CLASS); }
```

The **GetString()** function loads the string from the table and returns a pointer to it. You need to include this function, listed below, in your CPP source file:

```
TCHAR *GetString(int id) {  
    static TCHAR buf[256];  
    if (hInstance)  
        return LoadString(hInstance, id, buf, sizeof(buf)) ? buf : NULL;  
    return NULL;  
}
```

For more information see the Advanced Topics section on [Globalization](#).

## Specifying Output and Include File Settings

You now need to specify the output file name. This is the name of the DLL created. You specify the output file name for your plug-in using the procedure below:

- 1 From the **Project** pull down menu choose **Settings...** Make sure all the configuration are being changed by selecting **All Configurations** from the **Settings For:** dropdown list.

- 2 From the **Project Settings** dialog box choose the **Link** Tab.

- 3 Under **Output file name:** Enter the path to 3ds max plug-ins and your file name. For example **\3DSMAX\STDPLUGS\MYPLUGIN.DLO**

Result: When the project is compiled and linked successfully, the DLL is written to this location.

The example above (**MYPLUGIN.DLO**) is the extension typically used by procedural object plug-ins. For a list of the standard filename extensions for all plug-ins see [Plug-In Directory Search Mechanism](#). Note that 3ds max will only load DLLs that use these standard extensions.

You now need to specify the path of the include files for the SDK:

- 1 From the **Project Settings** dialog box choose the **C/C++** Tab.

- 2 From the **Category:** drop down list box choose **Preprocessor**.

- 3 Under **Additional include directories:** enter the path **\MAXSDK\INCLUDE**.

You also need to add **COMCTL32.LIB** which is not included by default by Visual C++ in the list of libraries to link.

- 1 Still within the **Project Settings** dialog box choose the **Link** Tab (you may need to scroll to see it).

- 2 In the list of Object/library modules add **COMCTL32.LIB** to the list of existing choices.

- 3 Select **OK** to exit the dialog.

## Updating the Project Configurations

A new plug-in project contains two default configurations. These are named **Win32 Release** and **Win32 Debug**. The sample code that comes with the 3ds max SDK has an additional configuration: **Win32 Hybrid**. This section discusses the need for these different configuration.

There are three conditions for plug-ins working with MAX. These are:

- 1) Release 3ds max and Release Mode (non-debug) plug-ins. When you create a plug-in for general distribution you'll compile in **Release** mode.
- 2) Release 3ds max and Debug Mode plug-ins. While developing a plug-in and using source code level debugging you'll compile in **Hybrid** mode.
- 3) Debug 3ds max (only registered developers have access to this special version of MAX) and Debug Mode plug-ins. While using the special Debug SDK, and Debug 3ds max you'd compile in **Debug** Mode.

In Microsoft Developer Studio the Debug and Release C runtime libraries use different heap management. This means that allocating an object in debug mode and de-allocating it in release mode (and vice versa) can cause a crash. Most developers will work with a version of 3ds max compiled in Release mode. Thus it uses the run-time libraries called **Multithreaded DLL**. Plug-Ins that work with this version of 3ds max need to match this run-time library. If they don't crashes will occur.

To prevent this from happening you need to create a new configuration and change some of the settings of the existing configurations.

To create the **Win32 Hybrid** configuration follow these steps:

- 1 From the **Build** menu choose **Configurations...** and then choose **Add...**
- 2 Enter **Hybrid** in the **Configuration** edit box then choose **Win32 Debug** from the **Copy settings from:** list box and press **OK** and then **Close** to exit the **Configurations** dialog. This creates a new Hybrid configuration using the same initial settings as the **Win32 Debug** configuration.
- 3 From the **Project** menu choose **Settings....**
- 4 From the **Settings For:** section choose the **Win32 Hybrid** configuration.
- 5 From the **C/C++** tab, under the **Categories:** drop down list choose **Code Generation**.
- 6 From the **Use run-time library:** drop down list choose **Multithreaded DLL**.
- 7 Choose the **General** tab of the dialog.
- 8 In the edit boxes under both **Intermediate files:** and **Output files:** enter **Hybrid**.
- 9 Select **OK** to exit the dialog.

Now update the **Win32 Release** configuration:

- 1 From the **Project** menu choose **Settings....**
- 2 From the **Settings For:** section choose the **Win32 Release** configuration.

- 3 From the **C/C++** tab, under the **Categories:** drop down list choose **Code Generation**.
- 4 From the **Use run-time library:** drop down list choose **Multithreaded DLL**.
- 5 Select **OK** to exit the dialog.

Now update the **Win32 Debug** configuration:

- 1 From the **Project** menu choose **Settings...**
- 2 From the **Settings For:** section choose the **Win32 Debug** configuration.
- 3 From the **C/C++** tab, under the **Categories:** drop down list choose **Code Generation**.
- 4 From the **Use run-time library:** drop down list choose **Debug Multithreaded DLL**.
- 5 Select **OK** to exit the dialog.

Result: You now have three configurations available, each used for a different purpose. When you create a plug-in for distribution to the public use your **Win32 Release** configuration. During development of the plug-in when you want to use source level debugging use the **Win32 Hybrid** configuration. If you are a registered developer using the special Debug SDK and Debug 3ds max then use the **Win32 Debug** configuration.

## Specifying Library files for the Project

You will also need to add the 3ds max library files (**LIB**) to the project. The **LIB** files required varies depending on the type of plug-in that is being created. Here is the list of the available library files and their use:

### **ACAP.LIB**

The Call Attributed Profiler library. When a plug-in is using the profiler to test performance this library is required. See the Advanced Topics section on [Debugging](#) for more information on profiling.

### **BMM.LIB**

Bitmap manager library. Image loader / saver plug-ins, and those that deal with bitmaps require this library.

### **CLIENT.LIB**

Network rendering client library. This one is used internally for network rendering. A plug-in developer won't use this library.

### **CORE.LIB**

Functions exported from MAX. All plug-ins use this library.

## **EDMODEL.LIB**

The NURBS library. Plug-Ins that want to use the NURBS API require this library.

## **EXPR.LIB**

Expression library. Plug-Ins that use the mathematical expression parser ([Class Expr](#)) require this library.

## **FLILIBD.LIB**

## **FLILIBH.LIB**

## **FLILIBR.LIB**

Flic file (FLI/FLC) libraries (debug, hybrid, and release versions). Plug-ins that work with FLI/FLC files require the appropriate version of this library. Plug-Ins for release (using the Win32 Release configuration) should use FLILIBR.LIB. Plug-Ins used for source level debugging should use FLILIBH.LIB. Registered developers using the special 3ds max Debug SDK should use FLILIBD.LIB.

## **FLT.LIB**

Filters library. Image Filter plug-ins use this library.

## **GCOMM.LIB**

This library is used internally.

## **GEOM.LIB**

Geometry library. Procedural objects, modifiers, controllers, and utility plug-in use this library.

## **GFX.LIB**

Graphics library. Plug-ins that use MAX's GraphicsWindow calls require this library.

## **LVSIO.LIB**

This is a library designed for reading and writing the Lightscape files.

## **MAXSCRIPT.LIB**

Users of the MAXScript SDK require this library.

## **MAXUTIL.LIB**

Utility library. All plug-ins use this library.

## **MESH.LIB**

Mesh library. Plug-Ins that call methods of MAX's Mesh class require this library.

## **MNMATH.LIB**

The 'Minnesota Math' library. Plug-Ins that use the MNMesh and related classes require this library. See [Class MNMesh](#) for details.

## **PARAMBLK2.LIB**

The Parameter Block 2 library. Plug-ins that use the parameter block / map system introduced in R3 require this library.

## **PARTICLE.LIB**

Particles library. Plug-ins that relate to particle systems use this library.

## **PATCH.LIB**

Patch library. Plug-ins that deal with Patch objects use this library.

## **RENDERUTIL.LIB**

Render utilities library. It includes the implementation of Quantizer and ColorPacker methods.

## **TESSINT.LIB**

Tesselation library. Plug-ins that deal with the tessellation of NURBS or patches use this library.

## **VIEWFILE.LIB**

This is the file viewer library used by VIEWFILE.DLL.

To add the library files to the project follow these steps:

- 1** From the **Project** menu choose **Add To Project** then **Files...**
- 2** Change the **Files of type:** drop down list to **Library files (.lib)**.
- 3** Choose the required **LIB** files from the list (from the **\MAXSDK\LIB** directory) (you may **Ctrl** click on them to select more than one). Select **OK** to exit the dialog.

Result: The **LIB** files are added to the project.

Note: You may remove the files **ODBC32.LIB** and **ODBCCP32.LIB** which are included by default but not needed.

## **Finishing Up**

To complete the new project creation follow these steps:

Since you'll likely need to spend time debugging the Hybrid configuration should be your initial choice. When you've finished development and want to create your DLL for distribution change this to Win32 - Release and rebuild. To set your default configuration to Hybrid do the following.

- 1** From the **Build** menu choose **Set Active Configuration...**

2 Choose **Win32 Hybrid** and select **OK** to close the dialog.

Now you need to update the project dependencies and save your work.

1 From the **Build** pull down choose **Update All Dependencies...** and choose the all your configurations using the check boxes and then select **OK**.

2 From the **File** menu choose **Save Workspace** to save your new project.

### **Disabling Exception Handling**

This step is not necessary, but developers may wish to disable exception handling for their plug-ins. This can be done using the **Project Settings** dialog, **C++ Tab**, **C++ Language** category and un-checking the **Enable exception handling** box. This makes the code both smaller and faster.

# COM/DCOM Interface

See Also: [Class GUP](#).

## **Overview**

The idea behind this COM (Component Object Model) object is to expose the core of 3ds max so applications can invoke 3ds max to generate images. The whole implementation of the COM interface was done as a plug-in itself so developers have the option of enhancing it in any manner they wish. Not just by adding interfaces to this object but also by creating entire new objects using this implementation as a code base. The code is extensively documented and the test "Client Application" is also extensively documented in a way where even developers who don't have any experience with COM objects can make use of it. For those with experience, the 3ds max COM interface is implemented entirely using ATL which makes it very simple to add and/or modify any aspect of the code.

COM Interfaces and their types are defined in an "**IDL**" file. They may be found in **\MAXSDK\SAMPLES\GUP\COMSRV\COMSRV.IDL**.

## **Registering 3ds max as a DCOM Server**

The following steps register 3ds max as a DCOM server:

- 1) Build **\MAXSDK\SAMPLES\UTILITIES\COMSRV.MAK** and copy the resulting **COMSRVUI.DLU** to the 3ds max PLUGINS directory.
- 2) Start 3ds max and go the Utility Panel. Choose More and pick COM/DCOM Server Control.
- 3) If the button in the comand panel says "Register" then click it, or if it says "Unregister", then do nothing (as 3ds max is already registered).

Now 3ds max is registered as a DCOM server and an instance of it can be created from any COM client.

It is also possible to register and unregister 3ds max from the command line. There are two command line options that can be passed to MAX:

**3DSMAX -RegisterMAXRenderer**

**3DSMAX -UnregisterMAXRenderer**

## **The MaxRenderer Interface**

This is the main interface exported by this COM object. Through its methods you can load a 3ds max scene, define the parameters for a resulting image, request 3ds max to render any number of frames, collect the resulting images, and so on. The other interfaces defined are support interfaces used by the different methods of the 3ds max Renderer Interface.

## Properties

These properties are the same ones found in the 3ds max API. They are exposed here so applications can query or define the different attributes of a scene to be rendered.

**HRESULT AnimationStart([out, retval] float \*pVal);**

**HRESULT AnimationStart([in] float newVal);**

Sets or returns the frame number defined as the starting frame for the current animation. The frame number argument is a float in order to isolate the foreign application from MAX' frame granularity. A 3ds max frame can be divided into around 4096 time slices. In other words, it is possible to generate animation images with a  $1/(30 * 4096)$  of a second resolution.

**HRESULT AnimationEnd([out, retval] float \*pVal);**

**HRESULT AnimationEnd([in] float newVal);**

Same as above but for the last frame of the current animation.

**HRESULT RenderFieldRender([out, retval] BOOL \*pVal);**

**HRESULT RenderFieldRender([in] BOOL newVal);**

Defines (or gets the current state) if the rendered images will be field rendered.

**HRESULT RenderColorCheck([out, retval] BOOL \*pVal);**

**HRESULT RenderColorCheck([in] BOOL newVal);**

Defines (or gets the current state) if the generated images will be checked for invalid colors.

**HRESULT RenderSuperBlack([out, retval] BOOL \*pVal);**

**HRESULT RenderSuperBlack([in] BOOL newVal);**

Defines (or gets the current state) how pure blacks are generated.

**HRESULT RenderHidden([out, retval] BOOL \*pVal);**

**HRESULT RenderHidden([in] BOOL newVal);**

Defines (or gets the current state) if hidden objects are to be rendered.

**HRESULT RenderForceTwoSide([out, retval] BOOL \*pVal);**

**HRESULT RenderForceTwoSide([in] BOOL newVal);**

Defines (or gets the current state) if both sides of an object are to be rendered. Usually, just the "outside" is rendered.

**HRESULT RenderAtmosphere([out, retval] BOOL \*pVal);**

**HRESULT RenderAtmosphere([in] BOOL newVal);**

Defines (or gets the current state) if atmospheric effects are to be rendered.

**HRESULT RenderFieldOrder([out, retval] long \*pVal);**

**HRESULT RenderFieldOrder([in] long newVal);**

Defines (or gets the current order) the order fields are rendered (if field rendering).

## Methods

These methods are a superset of the methods found in the 3ds max API. They encapsulate some of the API functions (specifically those related to rendering) in order to simplify the process as well as to isolate the foreign applications from classes and data not exposed.

### **HRESULT LoadScene([in] BSTR SceneName);**

Loads the given 3ds max scene. Note that the location of the scene is from the perspective of the computer where the 3ds max instance is being created. The use of physical drive letters should only be used if you know the scene file resides in the same computer. In doubt, use UNC path names.

### **HRESULT SaveScene([in] BSTR SceneName);**

The counterpart of the above to save the given 3ds max scene.

### **HRESULT ImportFile([in] BSTR FileName);**

Same idea as in LoadScene() but it deals with non 3ds max files. This can be used to write a "translator" where scene files are imported and then saved as 3ds max files.

### **HRESULT EnumCameras(void);**

Use this to enumerate all cameras defined in a scene. When you request 3ds max to render a scene, one of the arguments you need to pass is the name of the camera to use. This method will return all cameras defined in the current scene. The returned names are collected through the `_ImaxRendererEvents()` interface defined below.

### **HRESULT OpenRenderer([in] BSTR CameraName, [in] IMaxBitmapInfo \*pBif, [in] BOOL region);**

Opens the renderer and prepares it to render. The first argument is the camera name to use. You can enumerate all cameras defined in a scene using the `EnumCameras()` methods described above. The `pBif` argument is a pointer to a `IMaxBitmapInfo()` interface (defined below). This object will define the parameters of the resulting image to be created. The last argument defines if the render will be a full image or just a region.

### **HRESULT SetRegion([in] short x,[in] short y,[in] short w,[in] short h);**

If you are rendering just a region of the image, use this to define the region to be rendered. Given as an example a scene you want rendered at 640x480. If you

want to render just the first quarter of the image, you would pass a rectangle starting at 0,0, 320 pixels wide, and 240 pixels high.

### **HRESULT RenderFrame([in] float Time, [in] float Duration);**

You've loaded a scene, opened the renderer, now you want to render an image. Use this method to generate the image. The first argument defines the frame you want to render. The second defines the duration of the frame. Render progress is processed through the `_ImaxRendererEvents()` interface defined below. This method returns immediately. You keep track of the progress through the sink event defined above until you receive a `OnRenderDone()` event call.

### **HRESULT CancelRenderer(void);**

Cancels the rendering.

### **HRESULT CloseRenderer(void);**

When done, use this to close the renderer.

### **HRESULT GetLine([in] MAXchannelTypes type, [in] long line, [in] BOOL linear, [out, retval] SAFEARRAY(unsigned char) \*psa);**

Use this method to collect the image you just rendered. The first argument defines the bitmap layout. You can collect the image in many different formats depending on what channels you requested to be rendered in the `ImaxBitmapInfo()` argument passed to `OpenRenderer()`. The second argument defines which line you want to collect. The third argument defines if you want the image to be linear or gamma processing should take place.

### **HRESULT GetPreviewLine([in] long line, [in] long width, [out, retval] SAFEARRAY(unsigned char) \*psa);**

Used to collect sub-sampled lines used to maintain a preview of the render process as it is happening.

### **HRESULT ExecuteMAXScriptString([in] BSTR String);**

### **HRESULT ExecuteMAXScriptFile([in] BSTR Filename);**

These are used if you need to request 3ds max to do something and the method for doing this something is not exposed above, you could issue a 3ds max Script command (or a whole 3ds max Script file) in order to accomplish this "something" task. Note: These two calls are untested, and unfortunately, there is no error control.

### **The `_ImaxRendererEvents()` sink interface**

Use this interface to receive notifications and events from MAX.

#### **HRESULT OnEnumCameras([in] BSTR CameraName);**

When you issue a `ImaxRenderer::EnumCameras()`, `OnEnumCameras()` will be fired for every camera found in the scene.

#### **HRESULT OnRenderProgress([in] long Done,[in] long Total);**

This event will be fired during the render process to update the current progress of the render process.

#### **HRESULT OnRenderMessage([in] BSTR Message);**

Same as in above. This event will be fired with message updates describing the current stage of the render process.

#### **HRESULT OnRenderDone();**

This event is fired when the render is complete.

## **The MaxBitmapInfo Interface**

This interface is used to define the characteristics of the resulting image generated by MAX. There are only properties. The only one that might need explanation is the Channels() property. Use this to define what types of special channels you want generated. These are in addition to the standard RGBA channels.

## Command Modes and Mouse Procs

See Also: [Class CommandMode](#), [Class MouseCallback](#).

## **Overview**

In addition to any user interface that a plug-in may provide in the command panel via rollup pages, a plug-in may want to process mouse interaction in any of the viewports. Command modes allow the plug-in developer to define custom user / mouse interaction procedures. The system uses command modes as well. Examples are the viewport manipulation commands such as zoom and arc-rotate. Move, rotate and scale are implemented as command modes as well.

This section discusses command modes and the methods used to work with them.

## Methods

In MAX, there is always a current command mode. This is the instance of the class **CommandMode** that is currently at the top of the system's command stack. Command modes can either be pushed on the stack or replace the item on the top of the stack.

There are methods in MAX's [Interface class](#) to set and get the current command mode as well as getting the size of the command stack and finding a particular mode by index. See the methods:

```
virtual void PushCommandMode(CommandMode *m)=0;  
virtual void SetCommandMode(CommandMode *m)=0;  
virtual void PopCommandMode()=0;  
virtual CommandMode* GetCommandMode()=0;  
virtual void SetStdCommandMode(int cid)=0;  
virtual void PushStdCommandMode(int cid)=0;  
virtual void RemoveMode(CommandMode *m)=0;  
virtual void DeleteMode(CommandMode *m)=0;  
virtual int GetCommandStackSize();  
virtual GetCommandStackEntry(int entry)
```

Viewport manipulation commands, for instance, are always pushed on the stack so that if the user right clicks while in a viewport mode, the viewport command mode is popped off the stack, restoring the previous command mode in effect before the viewport command was engaged.

A command mode provides the system with two major things:

A callback procedure which flags nodes that belong in the foreground plane.

A mouse proc which handles mouse input in the viewports.

For the first item, plug-ins typically use a standard callback object provided by the system that flags all nodes dependent on the plug-in object. So when the plug-in object changes, any nodes that change as a result will be in the foreground plane, making redraw time faster. For more details see the section [Foreground / Background Planes](#).

The mouse proc is the object that allows plug-ins to process viewport mouse input. The **MouseCallback** class is a pure virtual class with a method named **proc()**. This method is called when a mouse event takes place. It gets passed several parameters:

The window handle of the window the user clicked in. This will be one of the viewports. An interface to the viewport can be obtained from the system, given this window handle.

A message parameter. This describes the type of event such as mouse down, mouse up, mouse move, etc.

The point number. this is 0 for the first click, 1 for the second, etc.

Flags. These specify the state of the Shift, Ctrl, and Alt keys.

The 2D screen point that the user clicked on. Methods in the viewport interface allow this point to be converted into a world space ray or a 3D view space point. A world space ray can be intersected with the active construction plane which results in a point on the active construction plane.

The mouse proc can specify how many points it wants for its specific task. For instance, the move transform needs two points. The first point represents the point where the user clicked down and the second point represents where the user released the mouse. A mouse proc can specify that it needs a large number of points and then return an abort code when it has completed its operation. For instance, the mouse proc that handles the creation of a spline expects to get some arbitrary number of points. When the user completes the spline, the mouse proc signals the system that its operation is completed.

In addition to providing the system with a foreground callback and a mouse proc, a command mode also has some other methods.

When a command mode becomes active its **EnterMode()** method is called. Usually it would respond by changing the state of a control somewhere to reflect to the user that they were in that mode. Typically this means pushing in a tool button. When the mode is finished the button should be returned to normal. The user may have activated the mode by pushing in the tool button, in which case it is redundant for the command mode to also set the button's state to 'in', however the mode could have been entered by right clicking while in a viewport manipulation mode in which case, the tool button would have been in the 'out' state. Note: A developer should use the standard color **GREEN\_WASH** for check buttons that instigate a command mode. While the command mode is active the button should be displayed in **GREEN\_WASH**. See [Class ICustButton](#) (specifically the method **SetHighlightColor()**) for more details.

When the active mode is replaced by a different mode, its **ExitMode()** method is called. Typically, the command mode would respond by setting its corresponding tool button to the 'out' state.

Command modes are identified by a class ID and an ID. The class ID is usually chosen from a pre-defined set of class IDs although developers may create their own. These define the 'type' of mode. The ID is the mode's unique ID.

There are two other methods a developer may find useful. These methods allow a plug-in to receive notification when the command mode has been changed by the user. These methods of class **Interface** are:

```
virtual void RegisterCommandModeChangedCallback(  
    CommandModeChangedCallback *cb)=0;
```

Register a callback object that will get called when the user changes the command mode. See [Class CommandModeChangedCallback](#).

```
virtual void UnRegisterCommandModeChangedCallback(  
    CommandModeChangedCallback *cb)=0;
```

Un-registers the command mode change callback object.

## Display of Messages in Command Modes

Developers cannot put up a message box using the Windows **MessageBox()** API while in the middle of a mouse operation inside a command mode. For example, if the user is dragging the mouse, it will cause problems to put up a message box. The prompt line may be used if a message needs to be sent to the user while in this state. This is done using the methods of class **Interface**. See [Class Interface](#) to review these methods.

## Summary

Command modes allow the plug-in developer to process mouse interaction in any of the viewports. A command mode provides the system with a foreground callback and a mouse proc. See the section [Object Creation Methods](#) for a discussion of how command modes participate in custom object creation procedures.

# Computing Face and Vertex Normals

See Also: [Class Point3](#), [Class Mesh](#), [Template Class Tab](#).

## Overview

Developers often need to compute the face and vertex normals when working with meshes in MAX. This section discusses the process of computing these normals and provides some sample code to do so.

## Face Normals

A face normal is a vector that defines which way a face is pointing. The direction that the normal points represents the front, or outer surface of the face. To compute the face normal of a face in a mesh you simply take the cross product of two edge vectors of the face. The following code shows how this is done using the 3ds max API. It loops through each face in a mesh and stores the computed face normal in a table of **Point3**s named **fnorms**. At the end of the code, the **DebugPrint()** API is used to display each normal in the table to the debug window of the VC++ IDE.

```
void Utility::ComputeFaceNormals(Mesh *mesh) {
    Face *face;;
    Point3 *verts;
    Point3 v0, v1, v2;
    Tab<Point3> fnorms;

    // Compute face (surface) normals for the mesh
    face = mesh->faces;
    verts = mesh->verts;
    fnorms.SetCount(mesh->getNumFaces());
    for (int i = 0; i < mesh->getNumFaces(); i++, face++) {
        v0 = verts[face->v[0]];
        v1 = verts[face->v[1]];
        v2 = verts[face->v[2]];
        fnorms[i] = (v1-v0)^(v2-v1);
        fnorms[i] = Normalize(fnorms[i]);
    }

    // Display the normals in the debug window of the VC++ IDE
    DebugPrint("\n\nFace Normals ---");
    for (i = 0; i < fnorms.Count(); i++) {
        DebugPrint("\nFace Normal[%d]=(%.1f, %.1f, %.1f)",
            i, fnorms[i].x, fnorms[i].y, fnorms[i].z);
    }
    DebugPrint("\n\n");
}
```

As a side note on face normals -- If you take the length of a face normal vector

(for example using the **Point3::Length()** method), you get a quantity equal to twice the surface area of that face.

## Vertex Normals

This section discusses the way vertex normals may be computed. Two algorithms are reviewed. The first ignores smoothing groups and returns an averaged normal at the vertex. The second one looks at the smoothing information and computes multiple normals when there are faces that have different smoothing groups that share a vertex. Sample code to handle the smoothing group aware case is shown.

First, consider the case where smoothing groups are not checked. In this case, there will be a single vertex normal for each vertex of the mesh. The vertex normal is the average of the face normals of each of the faces that share that vertex. The algorithm to compute such vertex normals is as follows:

First, allocate an array of normals, one for each vertex in the mesh, and initialize them to zero (**Point3(0,0,0)**). Then for each face, compute its face normal, and add it into each of the three vertex normals that the face contributes to. For example, if you have a vertex normal shared by five faces, each face will add in its normal to that vertex, and thus the result will be average normal of those five faces. When all the faces in the mesh have been processed, the average normal vector has been computed. As a last step, all the normals in the array can be normalized.

The above algorithm does not take smoothing groups into account. When smoothing groups are involved, you may have multiple normals for each vertex. For example, if you had a sphere that had the top and bottom hemi-spheres smoothed separately (i.e. not smoothed across the equator), then the vertices across the equator would have two normals for each vertex while the other vertices would have one. There may be as many normals as there are smoothing groups colliding at a vertex. However, it is by far the most common case to have one, and anything other than one or two is very rare.

The class used to compute vertex normals considering smoothing is shown below. This class, **VNormal**, is similar to the **RNormal** class used by 3ds max internally. The class contains a **Point3** which is the normal, a **DWORD** for the smoothing groups, and a pointer to the **next** normal -- this class is a linked list. The **init** variable is used as a flag to indicate if the first normal in the list has been initialized.

```
// Linked list of vertex normals
class VNormal {
```

```

public:
    Point3 norm;
    DWORD smooth;
    VNormal *next;
    BOOL init;

    VNormal() {smooth=0;next=NULL;init=FALSE;norm=Point3(0,0,0);}
    VNormal(Point3 &n,DWORD s)
    {next=NULL;init=TRUE;norm=n;smooth=s;}
    ~VNormal() {delete next;}
    void AddNormal(Point3 &n,DWORD s);
    Point3 &GetNormal(DWORD s);
    void Normalize();
};

```

A key method to this class is **AddNormal()**. It is used when a face is going to add its normal to a vertex. This method is passed the normal and the smoothing information for that face. It checks if the normal passed shares smoothing information with the existing normal. If it does, the normal is added in, and the smoothing bits are bitwise OR-ed in. If it does not, a new vertex normal is created. In this way, as normals that share smoothing information are added, they contribute to the overall normal for that smoothing condition at the vertex. If it is a normal whose face does not share smoothing information, a new vertex normal is allocated.

```

// Add a normal to the list if the smoothing group bits overlap,
// otherwise create a new vertex normal in the list
void VNormal::AddNormal(Point3 &n,DWORD s) {
    if (!(s&smooth) && init) {
        if (next) next->AddNormal(n,s);
        else {
            next = new VNormal(n,s);
        }
    }
    else {
        norm += n;
        smooth |= s;
        init = TRUE;
    }
}

```

```

}

// Retrieves a normal if the smoothing groups overlap or there is
// only one in the list
Point3 &VNormal::GetNormal(DWORD s) {
    if (smooth&s || !next) return norm;
    else return next->GetNormal(s);
}

// Normalize each normal in the list
void VNormal::Normalize() {
    VNormal *ptr = next, *prev = this;
    while (ptr) {
        if (ptr->smooth&smooth) {
            norm += ptr->norm;
            prev->next = ptr->next;
            delete ptr;
            ptr = prev->next;
        }
        else {
            prev = ptr;
            ptr = ptr->next;
        }
    }
    norm = ::Normalize(norm);
    if (next) next->Normalize();
}

```

The method **ComputeVertexNormals()** shown below is a demonstration method that uses the **VNormal** class above. The first thing done is to create a table of the vertex normals. Note that since the **Tab** class does not do any initialization (it only allocates the memory), the code loops through each normal and call the constructor to perform the initialization. Then it goes through each face , calculates the surface normal, and adds it into each of the three vertex normals for the face using **AddNormal()**. When all the faces have been processed, it goes through each of the vertex normals and normalizes them. In the code below, the vertex normals are displayed using **DebugPrint()** to the

output window. If there is more than one normal at a vertex, each one is displayed (the **DisplayVertexNormal()** method recursively calls itself to display each one).

```
// Compute the face and vertex normals
void Utility::ComputeVertexNormals(Mesh *mesh) {
    Face *face;
    Point3 *verts;
    Point3 v0, v1, v2;
    Tab<VNormal> vnorms;
    Tab<Point3> fnorms;

    face = mesh->faces;
    verts = mesh->verts;
    vnorms.SetCount(mesh->getNumVerts());
    fnorms.SetCount(mesh->getNumFaces());

    // Compute face and vertex surface normals
    for (int i = 0; i < mesh->getNumVerts(); i++) {
        vnorms[i] = VNormal();
    }
    for (i = 0; i < mesh->getNumFaces(); i++, face++) {
        // Calculate the surface normal
        v0 = verts[face->v[0]];
        v1 = verts[face->v[1]];
        v2 = verts[face->v[2]];
        fnorms[i] = (v1-v0)^(v2-v1);
        for (int j=0; j<3; j++) {
            vnorms[face->v[j]].AddNormal(fnorms[i],face->smGroup);
        }
        fnorms[i] = Normalize(fnorms[i]);
    }
    for (i=0; i < mesh->getNumVerts(); i++) {
        vnorms[i].Normalize();
    }

    // Display the normals in the debug window of the VC++ IDE
    DebugPrint("\n\nVertex Normals ---");
```

```
for (i = 0; i < vnorms.Count(); i++) {  
    DisplayVertexNormal(vnorms.Addr(i), i, 0);  
}  
DebugPrint("\n\n");  
}
```

```
void Utility::DisplayVertexNormal(VNormal *vn, int i, int n) {  
    DebugPrint("\nVertex %d Normal %d=(%.1f, %.1f, %.1f)",  
        i, n, vn->norm.x, vn->norm.y, vn->norm.z);  
    if (vn->next) DisplayVertexNormal(vn->next, i, n+1);  
}
```

## Other Techniques for Computing Vertex Normals

This next section discusses two other techniques that may be used in computing vertex normals. The first technique weights the normals by the vertex angle on each face. The second weights the normal using the area of each face.

### Weighting by Face Angle

To understand why using a weighted normal approach might be used consider the following example:

Create a default Box and convert it to an Editable Mesh. Go into SubObject mode, and find vertex 1 (generally found at the front lower left corner). Display the edges, and go into object properties to unclick "edges only" (so you see the hidden edges). You'll see that vertex 1 is used by bottom faces 1 and 2. (In the SDK, these are vertex 0 and faces 0 and 1.) It's also used by front faces 5 and 6, and left face 11. (4,5,10.)

If we simply averaged the normals of all incident faces, we'd get:

Normalize  $(2*(0,0,-1) + 2*(0,-1,0) + (-1,0,0))$ , which is  $(1/3, 2/3, 2/3)$ .

Because this arrangement of diagonals is haphazard and (inherently) asymmetric, we'd get vertex normals on a box pointing odd, uncoordinated directions.

If we instead weight the normals by the vertex angle on each face, we get:

Normalize  $(\pi/2*(0,0,-1) + \pi/2*(0,-1,0) + \pi/2*(-1,0,0)) = (1,1,1)/\text{Sqrt}(3)$

This is more natural for the user. Each vertex normal points away from all three sides symmetrically. (The individual front and bottom triangles may have varying angles, but the pairs of them always add up to  $\pi/2$ .)

This seems like the right approach in general -- when you divide a face, for example, you don't wind up changing the vertex normal when the surface hasn't essentially changed. If you change vertex angles by dragging neighboring vertices, the normal changes in a natural fashion.

You can compute the vertex angle by using dot products as shown below:

**// Corner is 0, 1, or 2 -- which corner do we want the angle of?**

**float FindVertexAngle(Mesh \*mesh, int face, int corner) {**

**int cnext = (corner+1)%3;**

**int cprev = (corner+2)%3;**

**DWORD \*vv = mesh->faces[face];**

```

// Get edge vectors:
Point3 A = mesh->verts[vv[cnext]] - mesh->verts[vv[corner]];
Point3 B = mesh->verts[vv[corner]] - mesh->verts[vv[cprev]];

// Normalize the edge-vectors, but return 0 if either has 0 length.
float len = Length(A);
if (!len) return len;
A = A/len;
len = Length(B);
if (!len) return len;
B = B/len;

// The dot product gives the cosine of the angle:
float dp = DotProd (A,B);
if (dp>1) dp=1.0f; // shouldn't happen, but might
if (dp<-1) dp=-1.0f; // shouldn't happen, but might
return acos(dp);
}

```

To be efficient when computing all normals, you may want to cache the normalized edge directions (A & B). You can index these by an adjacent edge list for the mesh, where edir[i] is the unit vector pointing from vertex ae->edges[i].v[0] to ae->edges[i].v[1] (AdjEdgeList \*ae).

### Weighting by Face Area

Another possibility is to weight the normals by the area of each face. The area of a face is very easy to compute, it's just half the length of the normal cross product:

```

void GetAreaAndNormal (Mesh *mesh, int face, Point3 & N, float
area) {
    DWORD *vv = mesh->faces[face].v;
    Point3 A = mesh->verts[vv[1]] - mesh->verts[vv[0]];
    Point3 B = mesh->verts[vv[2]] - mesh->verts[vv[0]];
    N = A^B;
    area = Length (N) / 2.0f;
    Normalize (N);
}

```

}

This works using any two edges for A and B.

To weight face normals by area, you can just use the  $N=A \wedge B$  vector directly.

Weighting by area gives an interesting result, but perhaps not as satisfactory as weighting by face angle.

## Custom User Interface Controls

See Also: [Class Interface](#), [Class ICustomControl](#), [Class ICustEdit](#), [Class ISpinnerControl](#), [Class ISliderControl](#), [Class ICustImage](#), [Class ICustStatus](#), [Class IColorSwatch](#), [Class ICustButton](#), [Class ICurveCtl](#), [Class ICustToolbar](#), [Class IRollupWindow](#), [Class IOffScreenBuf](#), [Class TCBGraphParams](#), [Class IDADWindow](#).

## Overview

A set of custom controls are available for use in the user interface design of 3ds max plug-ins. The 3ds max program itself makes extensive use of these custom controls. These controls provide an important element of consistency between the plug-in and the system, making new plug-ins appear fully integrated with MAX. The use of these controls by the majority of developers will provide a level of familiarity for users who often work with many different plug-ins.

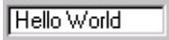
This section takes a quick look at all the control types. It then discusses two major aspects of using MAX's custom controls. The first is how the controls are added to a rollup page in the command panel and presented to the user. The second is how the user's operation of these controls is processed by the plug-in.

The sample program **CUSTCTRL.CPP** demonstrates each of the custom controls which may be used in the command column. To run this program, copy the file **\MAXSDK\PLUGIN\CUSTCTRL.DLU** into your stdplugs directory. The program is implemented as a utility plug-in. You can run the program by choosing the Custom Control selection in the Utility branch under How To. This section makes reference to portions of this plug-in.

You can also look at the custom controls' header file in **\MAXSDK\INCLUDE\CUSTCONT.H**.

## Available Custom Controls

### Custom Edit Controls



This control is a simple text input control. The user may type any string into the field and the plug-in is notified when the user presses the ENTER key. There are also methods to parse and return integer and floating point values entered in the control. If the edit control is to be used with numeric values, it is typically used in conjunction with a custom spinner control.

## Spinner Controls

 (Spinner control)

 (Edit control and Spinner control)

The spinner control is used (usually in conjunction with the custom edit control) to provide input of values limited to a fixed type. For example, the control may be limited to the input of only positive integers. The input options are integer, float, universe (world space units), positive integer, positive float, positive universe, and time. This control allows the user to increment or decrement a value by clicking on the up or down arrows. The user may also click and drag on the arrows to interactively adjust the value. The Ctrl key may be held to accelerate the value changing speed, while the Alt key may be held to decrease the value changing speed. The user may also right click on the arrows to reset the value its default.

## Slider Controls

This control is available in release 3.0 and later only.

The custom slider control is functionality similar to the custom spinner control with some additional functionality. The slider control is used (sometimes in conjunction with the custom edit control) to provide input of values limited to a fixed type.

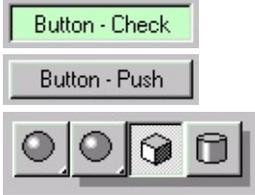


(Slider Control)



('Bracketed' Slider Control)

## Custom Button Controls



Custom Buttons allow the developer to have extra control of the way buttons appear and behave in the dialog box. These custom buttons have the following features:

- The button can be either a check button (which stays pressed in until the user selects it again), or a push button (which pops back out immediately).
- The highlight color of the check button may be specified.
- A button may function as a fly-off. Any number of additional buttons may be specified to fly off. The direction of the fly off may be specified or computed automatically.
- The buttons may be labeled with text or images. Four images may be specified allowing precise control over how the button appears when enabled or disabled and pressed in or released.

## Custom Status Control

(0, 255, 128)

This control provides a recessed area of the dialog which the developer may use as a status prompt area.

## Custom Toolbar Control



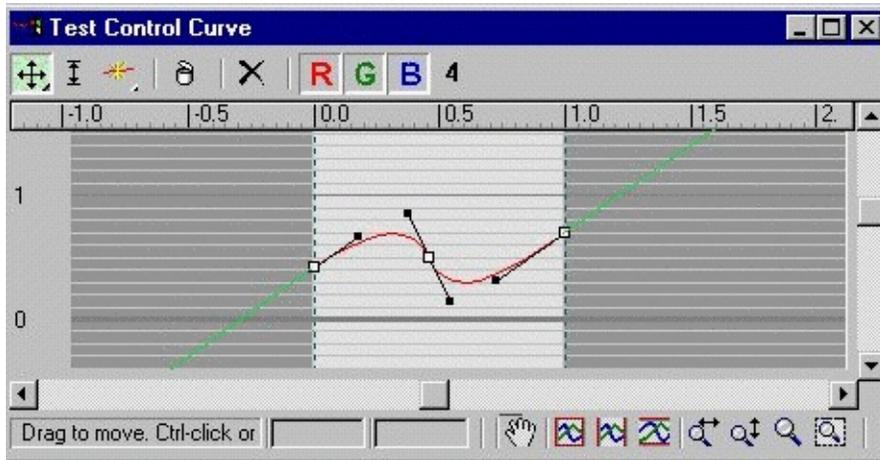
This control allows the creation of toolbars containing buttons (push, check, and fly-offs), status fields, separators (spacers), and other Windows or user defined controls. Note: The standard size for 3ds max toolbar button icons is 16x15.

## Custom Curve Control

This is a spline based control which returns output values from a user adjustable curve. An example of this control in the 3ds max user interface can be seen in the Color Map section of the Output rollup of a 2D Texture map.

Sample code using these APIs is available in

**`\MAXSDK\SAMPLES\UTILITIES\CCUTIL\CCUTIL.CPP.`**



## Custom Image Control



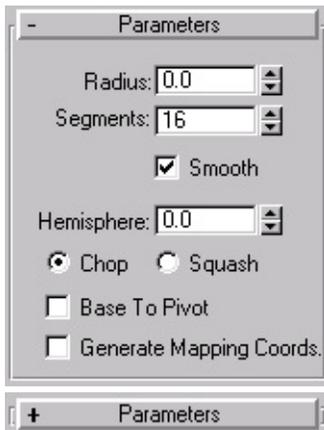
The custom image control provides a recessed area in the dialog to display a bitmap image.

## Color Swatch Control



The Color Swatch control presents the user with the standard 3ds max modeless color selector when the user clicks on the control. The color swatch control displays the currently-selected color and may be continuously updated as the user interactively selects new colors. Color Swatches also handle drag and drop between color swatches.

## Rollup Window Control



This control is used if you are creating a dialog box which will not be used in the command panel. This control adds a container area for rollup pages to be added to the dialog, and provides a scroll bar just like the command panel itself.

Note that this is a special case. Normally, adding rollup pages to the command panel is done using the simple **AddRollupPage()** method of the **Interface** class. This control is only used when you want to have a scrolling region for rollup pages in a dialog box.

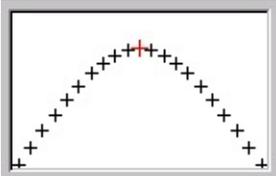
## **Window Thumb Tack Control**

This control installs a thumb tack into a window title bar which allows the user to make the window 'Always On Top'.

## **Off Screen Buffer Control**

This control provides an off-screen buffer which the developer may draw into, then quickly blit onto the actual display for flicker-free image updates.

# TCB Graph



This control displays a tension / continuity / bias graph.

## **Drag and Drop Window Control**

This is a new type of custom control available in 3ds max 2.0 and later. It is used to provide drag and drop to and from things other than Custom Buttons.

## How to a Create a Rollup Page using the Custom Controls

The tool for creating the rollup page portion the user interface of your plug-in is the dialog editor of Visual C++ Developers Studio. You use the dialog editor to create the rollup page, and place and arrange the controls.

The custom controls are positioned in a rollup page like any other control. Each one is created using the Custom Control button. The Custom Control button is the one shown highlighted below.



The control is identified as being custom by filling in the Class field of the Custom Control Properties dialog box with an appropriate string indicating the type of control. To bring up this dialog, double click on the control in the dialog editor window.



These are the values to be used in the Class field of the Custom Control Properties dialog for those custom controls which may appear in a rollup page:

- Custom Edit control - **CustEdit**
- Custom Spinner Control - **SpinnerControl**
- Custom Button control - **CustButton**
- Custom Toolbar control - **CustToolbar**
- Custom Image control - **CustImage**
- Custom Status control - **CustStatus**

Color Swatch control - **ColorSwatch**

Custom Rollup Window - **RollupWindow**

Custom DragAndDrop Window control - **DragDropWindow**

Custom TCB Graph - **TCBGraph**

The other fields in the user control properties dialog are these:

**ID** - This is the resource's identifier. The resource ID is usually a symbol supplied by Visual C++ and defined in the .H file that Visual C++ creates as part of your project. This is where you define the symbolic name for the resource.

**Caption** - This is the text that appears as part of the control to label it. This field is used for Custom Buttons and Status controls.

**Visible** - This determines whether or not the control is visible when the application is first run.

**Disabled** - This determines if the resource is displayed as disabled when the dialog box is created.

**Group** - This specifies the first control of a group of controls in which the user can move from one control to the next by using the arrow keys. All controls in the tab order after the first control with the Group property set to False belong to the same group. The next control in the tab order with Group set to True ends the first group of controls and starts the next group.

**Tabstop** - This specifies that the user can move to this control with the TAB key. This is only appropriate for the edit and button controls.

**Class** - The name of the control's Windows class. This class must be registered before the dialog box containing the control is created. See the class list above.

**Help ID** - Assigns a help ID to the control. The help ID is based on the resource ID. Type: Bool. Default: False.

**Style** - A 32-bit hexadecimal value specifying the control's style, primarily used to edit the lower 16 bits that make up a user control's sub-style.

**ExStyle** - A 32-bit hexadecimal value specifying the control's extended style.

When the rollup page dialog is created, it must be compiled and the resource file it generates must be included in the CPP source file. The default name for this file is **RESOURCE.H**. This may be changed by right clicking on the resource file name from the Resource View list (the top line) and choosing the Resource Includes... option. From this dialog, change the entry under Symbol header file.

Once you create a rollup page layout, there are several ways to present it to the user. The standard way is using a rollup page in the command panel. Typically,

this is called from within **BeginEditParams()** when the user is in the position of editing an items parameters.

There is a method of the interface class to handle adding the rollup. Its called **AddRollupPage()**. The syntax looks like this:

```
hParams = interfacePtr->AddRollupPage(  
    hInstance, // DLL instance handle.  
    MAKEINTRESOURCE(IDD_CUSTCTRL), // ID of the dialog  
    box.  
    DialogProc, // Dialog procedure to process user input.  
    ROLLUP_PAGE_TITLE, // Rollup title text.  
    (LPARAM)this); // Saves the this ptr of the item.
```

The standard width for rollup pages in the command column in the United States is 108 units. When you create dialogs for use in the command panel, always use 108 as the overall width (for use in the United States). Note the width and height are visible in the lower right hand corner of the IDE.

In summary, the custom controls are added to a rollup page layout using the resource editor window of VC++. The Class field of the resource indicates which type of control you are creating. The finished rollup page layout is displayed to the user using the **AddRollupPage()** function.

## How to Process User Input from the Custom Controls

Windows uses a 'Dialog Procedure' created by the developer to handle the users manipulation of the controls in the dialog,. As the user works with the controls, Windows sends messages to the dialog procedure. The developer is responsible for responding to these messages and implementing the logic to process the user input. A full description of this messaging system is beyond the scope of this documentation, but below is a brief overview. If you need more information, see the [Recommended Reading](#) section in the Orientation module.

When you create a dialog box or add a rollup page, you specify the dialog procedure to process the input. The basic structure of this dialog proc is listed below:

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT message,  
WPARAM wParam, LPARAM lParam)  
{  
  switch (message) { // Respond to the message ...  
    case WM_INITDIALOG: // Initialize the Controls here.  
      return TRUE;  
    case WM_DESTROY: // Release the Controls here.  
      return FALSE;  
    case WM_COMMAND: // Various messages come in this way.  
      break;  
    case WM_NOTIFY: // Others this way...  
      break;  
    // Other cases...  
    default:  
      break;  
  }  
  return FALSE;  
}
```

Windows passes in four parameters to the dialog procedure. These are the handle of dialog box, the message, and two parameters which hold message-specific information. Except in response to the **WM\_INITDIALOG** message, the dialog box procedure should return TRUE if it processes the message, and FALSE if it does not.

When the dialog box is initialized, the **WM\_INITDIALOG** message is sent. At this time the custom controls should be initialized. Each control has methods to initialize it. The methods used to initialize the control are discussed in the next section, or you may see the sample program **CUSTCTRL.CPP** for how this is done for each control.

When the user is finished with the dialog box, the **WM\_DESTROY** message is sent. This is where you release the controls. Again, the next section discusses this, or you may take a look at the sample program to see how this is done.

When the user works with any of the custom controls, Windows sends in specific messages to the dialog procedure. For example, when the user changes a spinner control, Windows sends in a **CC\_SPINNER\_CHANGE** message. The developer would add this case to the dialog proc code above and handle the processing required when the spinner changed. The dialog box may have several spinner controls: how does the developer know which spinner changed? The **IParam** and **wParam** arguments to the dialog procedure contain message-specific information. It is here that the ID of the control is provided. (This is the ID entered in the ID field of the user control properties dialog when the control was created). For example, the developer could add a case such as the one below to handle spinner change messages:

```
case CC_SPINNER_CHANGE:
    switch (LOWORD(wParam)) { // Switch on ID
        case IDC_ANGLE_SPINNER: // A specific spinner ID.
            angle = ((ISpinnerControl *)IParam)->GetFVal();
            break;
        case IDC_RADIUS_SPINNER: // A specific spinner ID.
            // Code to handle the Radius spinner...
            break;
    };
    break;
```

This code fragment presents several important concepts. The **LOWORD** macro used above retrieves the low-order word from the given 32-bit value. The spinner control provides the ID of the spinner which changed in the low order word of **wParam**. (Another macro, **HIWORD**, retrieves the high-order word from the given 32-bit value). Splitting the argument into low and high order words allows more information to be packed into the parameter.

A pointer to the spinner control is provided in the **lParam** parameter. This pointer is used to call a method of the spinner to get the new floating point value (**(ISpinnerControl \*)lParam >GetFVal()**).

As another example, let's look at the color change message sent in from the Color Swatch control. When the user is changing the colors in the dialog box, and the developer has asked to be notified on each change, Windows sends the **CC\_COLOR\_CHANGE** message. Again, the developer would add a case to the switch to handle this message. The **LOWORD** of **wParam** contains the ID of the color swatch control. The **HIWORD** of **wParam** contains 0 if the mouse button is down as the user is changing colors, and 1 if the mouse is up. Sometimes, the message sent to the dialog proc is **WM\_COMMAND**, and the programmer must look at both the high and low words of **wParam** to determine the nature of the message and the ID of the control which sent the message. For example, the code below demonstrates how custom button messages may be handled.

```
case WM_COMMAND:
    switch(LOWORD(wParam)) { // Switch on ID
        case IDC_BUTTON: // A specific button's ID.
            switch (HIWORD(wParam)) { // Notification codes
                case BN_BUTTONDOWN: // Button is pressed.
                    break;
                case BN_BUTTONUP: // Button is released.
                    break;
                case BN_RIGHTCLICK: // User right clicked.
                    break;
            };
            break;
        case IDC_FLYOFF: // A specific fly off control ID
            switch (HIWORD(wParam)) { // Notification codes
                case BN_FLYOFF:
                    // This notification code is sent when the
                    // user chooses a new fly off.
                    break;
            };
    };
```

```
break;  
};
```

In summary, when the user works with controls in a dialog, Windows sends messages to the plug-in's dialog proc. The developer responds to these messages to process the user input.

## Overview of Methods of the Custom Control Classes

Before you call any custom control methods, you must initialize the controls by calling the **InitCustomControls()** function (usually from the **DLLMain()** function of the plug-in).

Call **InitCustomControls( hInst )** where **hInst** is the **HINSTANCE** passed into **DLLMain()**.

For each control, there are two functions that you'll always use:

- 1 A function to initialize the pointer to the control so you may call its methods (usually when the dialog is initialized).
- 2 A function to release the control when you are done with it (when the dialog is destroyed).

The sample code below demonstrates this for a spinner control (the other controls are similar):

First, declare a pointer to the control:

```
static ISpinnerControl *radSpin;
```

To initialize the pointer call:

```
radSpin=GetISpinner(GetDlgItem(hDlg, IDC_RAD_SPIN));
```

Each custom control has a Get function to return a handle to the control.

In the example above, two functions are called from a single statement. The **GetDlgItem** function retrieves the handle of a control in the specified dialog box. The parameter **hDlg** is the dialog handle. The **IDC\_RAD\_SPIN** parameter is the ID for the control. This is the name entered in the ID field of the Custom Control Properties dialog box when the control was created.

When you are finished using the control, call:

```
ReleaseISpinner(radSpin);
```

Each control has a Release function to release it.

See [Class ICustomControl](#) for a list of the methods that are available for all the custom controls.

The following functions are not a part of class **ICustomControl** but may also be used in conjunction with the custom controls and UI design in MAX:

This is a bitmap brush where the bitmap is a gray and white checker board.

```
HBRUSH GetLTGrayBrush();
```

```
HBRUSH GetDKGrayBrush();
```

This returns the standard font.

**HFONT GetFixedFont();**

This returns the handle of the hand cursor used for panning.

**HCURSOR GetPanCursor();**

The following section provides links to the custom control classes that document how to use the specific methods of each control.

## Using the Custom Edit Control

See [Class ICustEdit](#).

## Using the Spinner Control

See [Class ISpinnerControl](#).

## Using the Custom Image Control

See [Class ICustImage](#).

## Using the Custom Status Control

See [Class ICustStatus](#).

## Using the Color Swatch Control:

See [Class IColorSwatch](#).

## Using the Custom Button Control

See [Class ICustButton](#).

## Using the Custom Toolbar Control

See [Class ICustToolbar](#).

## Using the Rollup Window Control

See [Class IRollupWindow](#).

## Using the Off Screen Buffer Control

See [Class IOffScreenBuf](#).

## Using the TCB Graph Control

See [Class TCBGraphParams](#).

## Using the Thumb Tack Control

This control installs a thumb tack into a window title bar which allows the user to make the window 'Always On Top'. This control has two functions.

### **void InstallThumbTack(HWND hwnd)**

This function installs a thumb tack in the title bar of a window. NOTE: The window class for the window should have 4 extra bytes in the window structure for **SetWindowLong()**.

### **void RemoveThumbTack(HWND hwnd)**

This function is used to remove the thumb tack from the window title bar.

## Custom Node Properties and App Data

See Also: [Class Interface](#), [Class INode](#), [Class ClassDesc](#), [AppData](#), [AppDataChunk](#).

## **Overview**

A plug-in developer may need to hang arbitrary data off nodes in the scene. 3ds max provides two ways to accomplish this, AppData and User Properties. This section discusses both these approaches. Also discussed is the way a developer can save data associated with their plug-in class to the 3ds max file.

## AppData

AppData is application-specific data that may be attached to any **Animatable** in the scene. With these APIs any 3ds max object (controller, object, node, modifier, material, etc.) can have custom data attached by other objects. These chunks are saved in the .MAX file and can be accessed through the object they are attached to.

The methods used to create AppData are from class **Animatable**. The data is accessed using three owner identifiers: The SuperClassID and the Class\_ID of the owner, and a sub-index.

```
void AddAppDataChunk(Class_ID cid, SClass_ID sid,  
    DWORD sbid, DWORD len, void *d);
```

This method is used to add an AppDataChunk to this Animatable. The chunk is identified using the Class\_ID, and SuperClassID of the owner, and an ID for sub-chunks.

```
AppDataChunk *GetAppDataChunk(Class_ID cid, SClass_ID sid,  
    DWORD sbid);
```

This method is used to retrieve a pointer to an AppDataChunk.

```
BOOL RemoveAppDataChunk(Class_ID cid, SClass_ID sid,  
    DWORD sbid);
```

This method is used to delete an AppDataChunk.

Sample code using these APIs can be found in

**\MAXSDK\SAMPLES\UTILITIES\APPDATA.CPP.**

Developers who have 3D Studio / DOS IPAS AppData associated with nodes being imported into 3ds max have APIs available in 3ds max to help process this imported data. AppData was used by IPAS plug-ins to store special data with nodes. This was used for example by IK and spline patches. These APIs allow a 3ds max plug-in to request incoming app data from an IPAS plug-in and use it. Plug-ins can register a callback that gets called when app data is loaded. The callback can then interpret the data and create a new object that will be used instead of the usual triangle mesh. For more details see [Class ObjectDataReaderCallback](#).

Note: Although it is possible to hang AppData off the scene or the root node, that data won't be saved in the 3ds max file since the scene and the root node are not actually saved.

## Custom Node Properties

Another type of data that may be attached to a node is referred to as a Custom Node Property. This data can be ASCII text, an integer value, a float value, or a Boolean value. These values are stored and retrieved using a 'key' string. This is a string of any length -- the only limitation is the key string must not contain spaces, tabs or the equal sign (=) character.

To get and set specific data values developers may use the following methods of `INode`. These get and set values for ASCII strings, ints, floats, and booleans:

**`GetUserPropString()`, `SetUserPropString()`**

**`GetUserPropInt()`, `SetUserPropInt()`**

**`GetUserPropFloat()`, `SetUserPropFloat()`**

**`GetUserPropBool()`, `SetUserPropBool()`**

An end user may enter this data as well (in the 3ds max user interface it is referred to as a "User Defined Property"). Using the Object Properties dialog a user may enter text in the following format:

**PropertyName=Property Value**

The property name cannot have spaces within it, although a user can use spaces (and any other characters) anywhere outside the property name. The property name must appear at the start of the line (although it can be indented with spaces or tabs) and there can be only one property per line.

A developer may access the entire user-defined property text buffer if they wish to parse it themselves. The methods to do this are **`GetUserPropBuffer()`** and **`SetUserPropBuffer()`**.

For reference information on these methods, see [Class INode](#).

## Saving Class Data

The following three methods of [class ClassDesc](#) may be used to save data associated with a class in a 3ds max file. This is appropriate for preference data and other similar information associated with a class. If you want to save data associated with the class have **NeedsToSave()** return TRUE and implement the **Save()** and **Load()** methods. See the Advanced Topics section on [Loading and Saving](#) for more details on the way data is written to and read from 3ds max files.

### **virtual BOOL NeedsToSave();**

Returns TRUE if there is data associated with the class that needs to be saved in the 3ds max file. If this is so, implement the **Save()** and **Load()** methods below. If there is no class data to save, return FALSE.

### **virtual IOResult Save(ISave \*s);**

If **NeedsToSave()** returns TRUE then this method should be implemented to save the data associated with the class.

### **virtual IOResult Load(ILoad \*l);**

If **NeedsToSave()** returns TRUE then this method should be implemented to load the data associated with the class.

Note: One peculiarity with the loading and saving of the class data is that if the user does a File New or File Reset then the class data is not cleared.

# Debugging

See Also: [Profiling Plug-In Performance](#).

This section presents information on Debugging. This include an introduction to using the VC++ debugger, a useful API for printing debug messages, and information about the [Sparks Developer Program](#).

## Using the Visual C++ IDE to Debug MAX Plug-Ins

### Overview

Developers may use the VC++ IDE to debug 3ds max plug-ins. This provides source code level debugging capabilities while execution is taking place inside the context of the plug-in. One can do things such as set breakpoints (specific lines in the source code where execution stops temporarily so the developer can examine variables, review the call stack, etc.) and establish variables to 'watch'. These variables appear in a separate window and display their current contents as they change.

### Setting Up for Debugging

Developer must use the plug-in project's Hybrid configuration for debugging. The Release build does not contain any source level information and thus cannot be used. For registered developers using the special Debug SDK the Debug configuration should be used (in the description below Hybrid is always referred to however).

To begin, make sure you've compiled the Hybrid configuration so you have the debug information available. You can set the Hybrid configuration as the current one by choosing Build.../Set Active Configuration.... Then choose Build/Rebuild All. This will create a DLL and associated files appropriate for use while debugging.

Next, VC++ needs to know where the 3ds max executable is. You can set this through the IDE by choosing Settings... from the Project pulldown menu. Then go to the Debug Tab, and with the Hybrid configuration selected (highlighted in the 'Settings for:' list), enter the full pathname to 3DSMAX.EXE. For example, under the 'Executable for debug session:' edit field enter 'd:\3dsmax\3dsmax.exe'. In the 'Working directory' field enter 'd:\3dsmax'.

### Executing 3ds max and Your Plug-In

There are several ways to cause execution to halt at a specific spot in your code. One is to set a breakpoint at the spot. This is done by placing the cursor on the line where you want execution to stop and pressing F9. Then, to run the debugger simply press F5. This will launch 3ds max and get things going. 3ds max will run and load your plug-in as usual upon startup. Begin execution of your plug-in as you normally would inside 3ds max (for example, if your plug-in is a modifier apply it to an object). When execution of your plug-in's code reaches the breakpoint, control will return to VC++ and you may use its tool to

examine the state of things (see the section below for more details).

A second way to reach a certain line in your code is to use 'Run to Cursor' option. This is chosen by placing the cursor on the desired line of your source file and selecting 'Build/Start Debug/Run to Cursor' (or by pressing Ctrl+F10). This will begin execution and stop when the point in your source code where the cursor sits.

Once execution is stopped, you can begin again by pressing F5, or pressing F11 to step to the next line of source or into the next function or method. Note that if you attempt to step into MAX's code (not your own) you'll wind up with a window showing the disassembly of the source (since no debugging information is available for MAX's internal code.) If this happens you can simply close this window and set another breakpoint inside your own code.

Note: Before you begin debugging it is often helpful to maximize your (main) source code window. This prevents other tiled windows from overlapping it so you won't have to scroll to see the currently executing line.

### **Examining Variables and the Call Stack**

A Variables Windows is available to examine variables within the program's current context. This window also has a dropdown list for the call stack that shows the hierarchy of functions that are pending completion. This is handy in the case of a program crash since it can often be used to show the sequence of calls made up to the point of the crash.

Also available is a Watch Window. This will display the contents of any variable and updates automatically as the variable changes.

For additional details on these and other debugging tools see the VC++ IDE online help.

### **Summary**

Developers may use the VC++ debugger to help develop 3ds max plug-ins. This section has presented an overview of getting started. Of particular importance is the use of the plug-in project's Hybrid configuration so source code information is available. See the section [Creating a New Plug-In Project](#) for details on setting up such a configuration.

## The DebugPrint() Function

The function **DebugPrint()** may be used to send information to the Debug window in the Developer Studio IDE. This is handy for outputting debug information, as it only appears while debugging. This window is scrollable so a developer can print a lot of information to it, then go back to look it over. The format string works like the standard C **printf()** function.

```
void DebugPrint(const TCHAR *format, ...);
```

## Additional Debugging Tools Available to Sparks Developers

A Debug SDK is available to [registered developers](#). This special version of the SDK provides additional internal source code to 3ds max and is useful to help developers debug their applications deep inside MAX. Also, registered developers are provided with a special build of 3ds max compiled in Debug mode. Using this debug build, registered developers can step into parts of the core of 3ds max as they are debugging.

This special version of 3ds max has a purple animate button to distinguish it from the regular Release build of MAX. There is also a Debug menu available on the toolbar. This menu provides options to review the 3ds max command stack to see what is the current command mode and what modes have been previously pushed.

Also provided is a menu item 'Pipeline...' Each line in the dialog displayed by this command represents the state of the pipeline cache after a particular modifier or derived object.

**TM** refers to the Transform Matrix channel: the number in parenthesis is a zero(0) or one(1) indicating the channel is invalid (0) or valid (1).

**MT** refers to the Material channel, followed by its validity in parenthesis.

**OB** refers to the Object, and is followed by a list in parenthesis of all the object channels. If the channel is listed, it is valid, otherwise only "--" will appear for that channel.

**TO** is the topology channel.

**GE** is the geometry channel.

**TX** is the texture coordinate channel.

**MT** is the sub-object material channel.

**SE** is the sub-object selection channel.

**SU** is the sub-object selection state.

**DI** is the display attributes channel.

An asterisk (\*) following any of these channels indicates that the channel is "locked", meaning that is a shallow copy of a channel that is owned by a cache "upstream" in the pipeline.

The numbers following "t=" and "g=" are the memory addresses of the topology and geometry channel data, respectively, which are useful for debugging

problems related to the shallow copying.

## Deferred Loading of Plug-ins

See Also: [Class DataClassDesc](#), [Class ClassEntry](#).

## **Overview**

It is possible to indicate to 3ds max that certain plug-ins should not be loaded at startup. Each deferred plug-in will be loaded when and if needed, which will happen when one or more of the 3ds max classes implemented by the plug-in is actually created. It might also happen when detailed information about the plug-in is requested. The principal benefit of deferral is that the Windows resources used by the plug-in will not be consumed unless there is some specific use or interrogation of the plug-in. Time taken at startup to load the plug-in, assign its virtual memory, etc. is also saved, or at least deferred.

Deferral is governed by entries in the Windows Registry. These entries contain critical information about each plug-in to be deferred, and about each 3ds max class implemented by each such plug-in. (See details below.) This information must agree with the corresponding values returned by API calls to the plug-in itself. Disagreement can seriously impair the integrity of 3ds max itself.

Typically the Registry entries are created by an installer program or by 3ds max itself if the plug-in supports AutoDefer, but in special cases they could be created by run-time code in a plug-in, or by manual use of a Registry editor.

## AutoDefer

If a plug-in supports deferred loading it can let 3ds max automatically register it. This is done by implementing a new function exported from the DLL itself. This function is called **CanAutoDefer()** and is an addition to the four other required DLL exported functions: **LibNumberClasses()**, **LibVersion()**, **LibDescription()** and **LibClassDesc()**. Example:

```
// Let the plug-in register itself for deferred loading
__declspec( dllexport ) ULONG CanAutoDefer()
{
    return 1;
}
```

You will also need to add this line to the \*.DEF file. Example:

```
CanAutoDefer @5
```

Before adding this function to a plug-in you have to make sure that the plug-in actually supports deferral.

When 3ds max starts up it loads all the plug-in that are not listed as deferred in the registry. If a plug-in is not found in the registry, 3ds max will ask the plug-in if it can be deferred by checking the implementation of **CanAutoDefer()**. If the function is defined, and it returns a non-zero value, then 3ds max continues to load the plug-in and queries it for all the values needed to store in the registry. 3ds max will then write the needed registry information so that next time it starts up, the plug-in will be deferred, i.e. it will not be loaded until it is needed.

When a plug-in is registered by AutoDefer a timestamp is added to the registry. During start-up, if the timestamp of the DLL doesn't match the one in the registry, then the registry information about that plug-in is deleted, the plug-in is loaded, and the plug-in is registered again.

The format of the timestamp is taken from the **lpLastWriteTime** member of the Win32 function **GetFileTime()**. The timestamp is divided into two entries, **TimeStampHigh** and **TimeStampLow**, referencing the **dwLowDateTime** and **dwHighDateTime** members of the **FILETIME** structure.

## Registry Key Structure

The following chain of Registry keys establishes the base for all plug-in descriptions:

**\\HKEY\_LOCAL\_MACHINE\SOFTWARE\Autodesk\3ds  
max\4.0\Plug-ins**

and

**\\HKEY\_LOCAL\_MACHINE\SOFTWARE\Autodesk\3ds  
max\4.0\Plug-ins\Discreet**

Under the 'Plug-ins' key, there can be one key for each supplier of plug-ins; for the internal 3ds max plug-ins, this key is 'Discreet'. Under the key for a supplier there will be one key per deferrable plug-in, with a name that is distinct from all other plug-in key names from that supplier.

This 'plug-in key' has certain mandatory values, and under it there will be one key per 3ds max class implemented by the plug-in. The name of each class key must be distinct, at least within the given plug-in. Each such 'class key' has a set of values, describing certain important properties of the class.

Plug-in Key Values(all must be supplied)

Value Type Example

DLLFile REG\_SZ vrmlimp.dli

LibDescription REG\_SZ VRML Scene Importer

LibVersion REG\_DWORD 0xbb80600

TimeStampHigh REG\_DWORD 0x1be7679 (optional, only supplied for AutoDefer plug-ins)

TimeStampLow REG\_DWORD 0x2d8172f2 (optional, only supplied for AutoDefer plug-ins)

Notes: The DLLFile value gives the name of the DLL file that this key describes; case is disregarded. The name must be unique, and must not match two different files, even if they are in different directories.

Class Key Values

Value Type Example Required for:

ClassIDA REG\_DWORD 0x35da0f0a All

ClassIDB REG\_DWORD 0x10ce4af8 All

ClassName REG\_SZ VrmlImp All

Category REG\_SZ SceneImport All  
SuperClassID REG\_DWORD 0xa10 All  
IsPublic REG\_DWORD 1 All  
OKToCreate REG\_DWORD 1 All  
ShortDesc REG\_SZ VRML Importers, exporters, bitmap loaders  
SupportsOptions REG\_DWORD 0 Exporters  
LongDesc REG\_SZ BMP Image File Bitmap loaders  
Capability REG\_DWORD 0 Bitmap loaders  
ExtCount REG\_DWORD 2 Importers, exporters, bitmap loaders  
Ext REG\_SZ WRL;WRZ Importers, exporters, bitmap loaders  
InputTypeA REG\_DWORD 0x10 Modifiers  
InputTypeB REG\_DWORD 0x00 Modifiers  
InternalName REG\_SZ BitmapTex All plug-ins using  
ParamBlock2/ClassDesc2

Notes: The IsPublic and OKToCreate values should be 0 (no) or 1 (yes). The Ext value consists of upper-case file extensions (without the leading dot); if there are more than one, they are separated by semicolons, with no spaces before or after. The number of extensions given in the Ext value must agree with the ExtCount value. The InternalName value should not be defined if it returns NULL.

## **Deferrable Plug-ins**

To be eligible for deferred loading, a plug-in must meet all of the following criteria.

- All 3ds max classes that it implements must be subclasses of superclasses in the list below.
- All required values must be supplied, and must be constant. Values must not depend on run-time circumstances.
- If values depend on build configurations (e.g. Debug vs. Release), results may be undefined.
- The plug-in must not implement any keyboard accelerators.

Supported Superclasses are: Scene Import, Scene Export, Utility, Bitmap Loader, Helper, Geometric Object, Material/Texture, Image Filter, OSM (Modifiers), WSM/WSMObjects (Space Warps)

A plug-in that does not meet these requirements must not be described for deferral in the Registry.

## **DLL, Library Functions, and Class Descriptors**

See Also: [Class ClassDesc](#), [Generate a Class\\_ID](#), [List of ClassIDs](#), [List of SuperClassIDs](#).

## Overview

The following DLL and Library functions must always be implemented by 3ds max plug-in developers. They are called by 3ds max or Windows at load time and provide information about the DLL being loaded and the plug-in classes provided by DLL. These functions are:

**DllMain(HINSTANCE hinstDLL,ULONG fdwReason,LPVOID  
lpvReserved)**

**LibNumberClasses()**

**LibClassDesc(i)**

**LibDescription()**

**LibVersion()**

This section presents information about each of these functions as well as the details of Class Descriptors. Class Descriptors provide information about plug-in classes and are used by developers in implementing the **LibClassDesc()** function.

## The DllMain() Function

3ds max plug-ins are implemented as DLLs. DLL stands for Dynamic Link Library. DLLs are object code libraries that let multiple programs share code, data, and resources. When developers compile and link their plug-in code, the output is a DLL. The **DllMain()** function is implemented by the developer and called by Windows at startup when the plug-in DLL is loaded.

### DLLMain()

This function is called by Windows when the DLL is loaded (as well as other times, for instance during rendering). Most plug-ins simply make two calls inside this function. These are to initialize the 3ds max custom user interface controls, and to initialize the Win95 common controls. The sample code below, which is typical for most plug-in types, shows how this is done. Developers can see the VC++ help for more details on the **DllMain()** function.

Note that the calls to initialize the controls are made only once. This is done by setting a global variable to TRUE when the controls have been initialized the first time, and checked each time the function is called.

As note above, this function may also be called many times during time critical operations like rendering. Therefore developers need to be careful what they do inside this function. In the code below, note how after the DLL is loaded the first time only a few statements are executed. This function should return TRUE.

```
int controlsInit = FALSE;  
BOOL WINAPI DllMain(HINSTANCE hinstDLL,ULONG  
fdwReason,LPVOID lpvReserved)  
{  
// Hang on to this DLL's instance handle.  
hInstance = hinstDLL;  
if (! controlsInit) {  
    controlsInit = TRUE;  
  
// Initialize MAX's custom controls  
InitCustomControls(hInstance);  
  
// Initialize Win95 controls  
InitCommonControls();  
}
```

```
return(TRUE);  
}
```

## The Library Functions

These functions provide access to, and information about, the classes present inside the plug-in's DLL.

### LibNumberClasses()

When 3ds max is first started, it looks for DLLs to load. When it finds one, it needs a way to determine the number of plug-in classes inside the DLL. The programmer provides this information to 3ds max by defining the

**LibNumberClasses()** function. For example:

```
__declspec(dllexport) int LibNumberClasses() { return 1; }
```

The developer should return the number of plug-in classes inside the DLL.

### LibClassDesc(i)

The plug-in must provide the system with a way to retrieve the Class Descriptors defined by the plug-in. Class descriptors provide the system with information about the plug-in classes in the DLL. The function **LibClassDesc(i)** allows the system to access the class descriptors. A DLL may have several class descriptors, one for each plug-in class. The function should return a pointer to the 'i-th' class descriptor. For example:

```
__declspec(dllexport) ClassDesc *LibClassDesc(int i) {  
switch(i) {  
case 0: return &MeltCD;  
case 1: return &CrumpleCD;  
default: return 0;  
}  
}
```

This example returns a pointer to the Melt plug-in class descriptor, or the Crumple plug-in class descriptor. See the section at the end of this topic for a discussion of the class descriptor returned by **LibClassDesc(i)**.

### LibDescription()

When a 3ds max file is loaded that contains an entity (procedural object, modifier, controller, etc.) that the system does not have access to (i.e. the DLL is not available), a message is presented to the user. The system requires that each DLL return a text string to present to the user if it is unavailable.

As an example, say a user has a Melt modifier. He applies the melt to an object in the scene, and saves the file. He then give this file to a friend who does not have the Melt DLL. When the friend loads the file, the system will put up a message indicating an entity exists in the file that relies upon a DLL that cannot be found. This message could be something like: "Melt Modifier. To obtain a copy, call 1-800-PLUG-INS". To provide this string to the system, the DLL must implement the function **LibDescription()**. This function returns a string to be presented if the DLL is not found. This string is also presented in the dialog presented to the 3ds max user via the File / Summary Info / Plug-In Info... command. Once a plug-in from the DLL has been used in the scene, the system stores the string in the 3ds max file (since it must be presented when the DLL is unavailable).

Note that the scene is still loaded even if the DLL is not present. 3ds max preserves any entities for which the DLL is not found. In this way, if the file is modified and saved, then the user obtains the DLL and loads the modified file, the entity is still present and linked in to the scene. Entities which are loaded without access to their DLL are referred to as orphaned entities.

The orphaned entity will be loaded by a generic 'stand-in' for its SuperClass. The stand in will display a minimal scene representation. For instance, if the entity is a modifier, it will display its name in the modifier list, but no parameters will be presented. Missing object-types have dummy-type representations in the scene. They can be moved, rotated, scaled, linked, grouped, deleted: anything that just involves the node. Missing controllers only provide constant default values, which are not adjustable.

Also see the Advanced Topics section on [Read Only Plug-Ins](#). By allowing your plug-in to function in read-only mode, users can freely distribute the DLL, but unless they are authorized to run based on the ID of a specific hardware lock they will have restricted use until the user purchases their own copy. Below is a sample implementation of this function.

```
__declspec( dllexport ) const TCHAR *LibDescription() {  
    return _T("Melt Modifier. Call 1-800-PLUG-INS to obtain a  
    copy");  
}
```

## **LibVersion()**

Developers must implement a function that allows the system to deal with

obsolete versions of 3ds max plug-in DLLs. Because the 3ds max architecture supports such a close relationship to its plug-ins, the system may at some point need to prevent older plug-in DLLs from being loaded. To allow 3ds max to accomplish this, the DLL must implement a function called **LibVersion()**. This function simply returns a pre-defined constant indicating the version of the system under which it was compiled. Future versions of 3ds max may update this constant, yet the older DLLs will always return the previous value. With this function the system can check if an obsolete DLL is being loaded, and if so put up a message.

```
__declspec( dllexport ) ULONG LibVersion() { return  
VERSION_3DSMAX; }
```

Note: The meaning of **VERSION\_3DSMAX** has been enhanced in R2 to include information about the current 3ds max release as well as the 3ds max API release number and the 3ds max SDK release number.

The upper word now contains the 3ds max release number multiplied by 1000, and the lower word is API number and SDK revision within the API version. 3ds max will only load DLL's with the API number that is current.

The version number is composed as follows:

```
#define VERSION_3DSMAX ((MAX_RELEASE<<16)+  
(MAX_API_NUM<<8)+MAX_SDK_REV)
```

Each part is describes below:

**MAX\_RELEASE**

This is the 3ds max release number multiplied by 1000.

**MAX\_API\_NUM**

This is the 3ds max API number. When a change in the API requires DLLs to be recompiled, this number is incremented and **MAX\_SDK\_REV** is set to 0. This will make all DLLs with the former **MAX\_API\_NUM** unloadable.

**MAX\_SDK\_REV**

This denotes the revision of the SDK for a given API. This is incremented when the SDK functionality changes in some significant way (for instance a new **GetProperty()** query response is added), but the headers have not been changed.

Note: A developer may access this value using the global function:

**Function:**

## **DWORD Get3DSMAXVersion();**

### **Remarks:**

Returns the state of the **VERSION\_3DSMAX** #define from **\MAXSDK\INCLUDE\PLUGAPI.H** when the running version of 3ds max was compiled.

The following macros may be used for extracting parts of **VERSION\_3DSMAX**:

```
#define GET_MAX_RELEASE(x) ((x>>16)&0xffff)
```

```
#define GET_MAX_API_NUM(x) ((x>>8)&0xff)
```

```
#define GET_MAX_SDK_REV(x) (x&0xff)
```

```
#define GET_MAX_SDK_NUMREV(x) (x&0xffff)
```

There is also a function you can call to find out which application your plug-in is running on -- either 3ds max or VIZ.

### **Function:**

## **APPLICATION\_ID GetAppID();**

### **Remarks:**

This function is available in release 2.5 and later only.

This function returns the ApplicationID, either VIZ or MAX. If a plug-in is designed to work only in one product, then it could call this function inside **ClassDesc::IsPublic()** to switch between exposing the plug-in or not.

For example:

```
IsPublic() { return(GetAppID()==kAPP_VIZ?1:0); } // VIZ only
```

### **Return Value:**

One of the following enum values:

**kAPP\_NONE**

**kAPP\_MAX**

**kAPP\_VIZ**

## Library Function Summary

The plug-in must implement these five functions: **DllMain()**, **LibNumberClasses()**, **LibClassDesc(i)**, **LibDescription()**, **LibVersion()**. These functions allow the system to determine information about the plug-ins contained in the DLL.

The section below provides information about the Class Descriptors which must be returned by **LibClassDesc(i)**.

## Class Descriptors

Class descriptors provide the system with information about the plug-in classes in the DLL. A method of the class descriptor is also responsible for allocating new instances of the plug-in class. The developer creates a class descriptor by deriving a class from **ClassDesc** and implementing several of its methods. Below is a sample class descriptor and the declaration of a single static instance.

```
class MeltClassDesc : public ClassDesc {  
public:  
int IsPublic() { return TRUE; }  
void * Create(BOOL loading=FALSE) { return new MeltMod(); }  
const TCHAR * ClassName() { return _T("Melt"); }  
SClass_ID SuperClassID() { return OSM_CLASS_ID; }  
Class_ID ClassID() { return Class_ID(0xA1C8E1D1,  
0xE7AA2BE5); }  
const TCHAR* Category() { return _T(""); }  
};  
static MeltClassDesc MeltCD;
```

The following six methods of the class descriptor are described below:

**IsPublic()**, **Create()**, **ClassName()**, **SuperClassID()**, **ClassID()** and **Category()**. Developers wishing to review the entire set of methods should see [Class ClassDesc](#).

### IsPublic()

This method returns a Boolean value. If the plug-in can be picked and assigned by the user, as is usually the case, return TRUE. Certain plug-ins may be used privately by other plug-ins implemented in the same DLL and should not appear in lists for the user to choose from. These plug-ins would return FALSE.

### Create(BOOL loading=FALSE)

3ds max calls this method when it needs a pointer to a new instance of the plug-in class. For example, if 3ds max is loading a file from disk containing a previously used plug-in (procedural object, modifier, controller, etc.), it will call the plug-in's **Create()** method. The plug-in responds by allocating a new instance of its plug-in class. As the sample implementation above shows, simply use the **new** operator.

The optional parameter passed to **Create()** is a flag indicating if the class being created is going to be loaded from a disk file. If the flag is TRUE, the plug-in may not have to perform any initialization of the object because the loading process will take care of it. See the Advanced Topics section on [Loading and Saving](#) for more information.

When the system needs to delete an instance of a plug-in class, it usually calls a method named **DeleteThis()**. Plug-in developers must implement this method as well. Since a developer uses the **new** operator to allocate memory, he or she should use the **delete** operator to de-allocate it. For instance, the developer would implement **DeleteThis()** as follows:

```
void DeleteThis() { delete this; }
```

See the Advanced Topics section on [Memory Allocation](#) for additional details.

### **ClassName()**

This method returns the name of the class. This name appears in the button for the plug-in in the 3ds max user interface.

### **SuperClassID()**

This method returns a system-defined constant describing the class that this plug-in class was derived from. For example, the Bend modifier returns **OSM\_CLASS\_ID**. This super class ID is used by all object space modifiers. Some other example super class IDs are: **CAMERA\_CLASS\_ID**, **LIGHT\_CLASS\_ID**, **SHAPE\_CLASS\_ID**, **HELPER\_CLASS\_ID**, and **SYSTEM\_CLASS\_ID**. See [List of Super Class IDs](#) for the entire list of available super class IDs.

### **ClassID()**

This method must return the **unique** ID for the plug-in object. A program is provided with the SDK to generate these ClassIDs. It is VERY important you use this program to create the ClassIDs for your plug-ins. **If you use one of the source code examples to create your plug-in, you MUST change the existing Class\_ID**. If you don't, you'll get a conflict. If two ClassIDs conflict, the system will only load the first one it finds (and will post a message when it attempts to load the second one noting that there is a Class\_ID conflict).

A Class\_ID consists of two unsigned 32-bit quantities. The constructor assigns a value to each of these, for example **Class\_ID(0xA1C864D1,**

**0xE7AA2BE5**). See [Class Class\\_ID](#) for reference information.

Note that the sample code plug-ins used in 3ds max use 0 as the second 32-bit quantity of the Class\_ID. **Only the built-in classes (those that ship with MAX) should have the second 32 bits equal to 0.** All plug-in developers should use **both** 32 bit quantities. Again, make sure you use the program provided to create your ClassIDs. This will ensure no conflicts arise between plug-ins. To generate a random Class\_ID and optionally copy it to the clipboard, click [Generate a Class\\_ID](#).

## Category()

The category is selected in the bottom-most drop down list in the create branch of the command panel. If this is set to be an existing category (i.e. "Standard Primitives", "Particle Systems", etc.) then the plug-in will appear in that category. Developers should NOT add to the categories provided by 3ds max (see the note below). If the category doesn't yet exist then it is created. If the plug-in does not need to appear in the list, it may simply return a null string as in `_T("")`. **Category()** is also used for modifiers to classify them in the button sets dialog.

**Important Note: The 3ds max architecture has a limit of 12 plug-ins per category in the Create branch. To prevent a problem with too many plug-ins per category, developers should ALWAYS create a new Category for their plug-ins, rather than using one of the existing ones used by the standard 3ds max plug-ins. Note that versions of 3ds max prior to release 1.2 would crash if there were more than 12 buttons per category.**

# Extending Render To Texture to Support New Materials

## Overview

The render to Texture feature of 3ds max r5 provides the ability to render out various elements from a standard render for example light maps and specular maps. It works in conjunction with the new Unwrap modifier, which can now flatten texture coordinates. Please see the user guide for a more detailed description.

This new feature now creates certain maps that do not fit into any particular material map slot for example, a light map. However 3<sup>rd</sup> party materials may exist that support extended features. **BakeTextureMappings.ini** file allows developers to add support for new materials. This provides the mapping between the different baked elements and the map slots in the material.

An example INI file can be seen below.

**[DeclareFileMappings]**

**DeclareFileMapping1=SaveAll**

**DeclareFileMapping2=CompleteOnly**

**DeclareFileMapping3=**

**DeclareFileMapping4=**

**DeclareFileMapping5=**

**DeclareFileMapping6=**

**DeclareFileMapping7=**

**DeclareFileMapping8=**

**DeclareFileMapping9=**

**DeclareFileMapping10=**

**[CompleteOnly]**

**shaderName = blinn**  
**ambientMap=**  
**diffuseMap = CompleteMap**  
**specularMap=**  
**diffuseLevelMap=**  
**specularLevelMap=**  
**glossinessMap=**  
**selfIllumMap=**  
**opacityMap=**  
**filterMap=**  
**bumpMap=**  
**reflectionMap=**  
**refractionMap=**  
**displacementMap=**  
**diffuseRoughnessMap=**  
**anisotropyMap=**  
**orientationMap=**  
**glossinessMap2=**  
**anisotropyMap2=**  
**orientationMap2=**

**[SaveAll]**

**shaderName= oren-nayar-blinn**  
**ambientMap= CompleteMap**  
**diffuseMap= DiffuseMap**  
**diffuseLevelMap= ShadowsMap**  
**specularLevelMap=SpecularMap**  
**bumpMap= NormalsMap**  
**opacityMap= AlphaMap**  
**selfIllumMap= LightingMap**

## **displacementMap= BlendMap**

The first section of the file is used to populate the drop down list in the RTT dialog box. For each entry there must be a subsequent entry defining the mapping. In the save all section all the supported baked elements are mapped to a standard material. This provides a method of viewing all the maps created, not necessarily to produce the correct rendering

The names, on the left hand side are the supported Maxscript parseable names for the parameters that have been defined in the ParamBlockDesc2 of the material. The name on the right is the actual baked element name used in the RTT. This mapping and naming needs to be accurate as it can cause a run time error in the RTT macro script

The Render To Texture script has been written to use the new ViewportManager to handle new Viewport Shaders, Internally it supports the MetalBump and LightMap shaders. The script can be found at **UI\Macroscripts\Macro\_BakeTextures.mcr**. In the function called **UpdateMaterial**, there is a section that provides additional ViewportShader support. The script lists all available shaders, so all that is needed is for the developer to supply the correct setup for the shader. Examples can be found in the function **setupLightMapShader**.

Script lists all available shaders, so all that is needed is for the developer to supply the correct setup for the shader. Examples can be found in the function **setupLightMapShader**

## Foreground / Background Planes.

See Also: [Class ChangeForegroundCallback](#), [Class IScene](#), [Class ViewExp](#), [Class GraphicsWindow](#), [Class CommandMode](#).

## Overview

In order to allow 3ds max to redraw the viewports as quickly as possible, an option is provided in the 3ds max user interface to allow the user to use Dual Planes. If dual plane mode is enabled, there is an extra buffer in memory that holds what is referred to as The Background. The background contains an entire image buffer and Z buffer (if Z buffering is turned on). Consider the following example to understand how the background buffer is used to speed up screen redraws:

Say there is a scene with two items: a small sphere that the user is interactively moving around, and a 500,000 polygon model that is not changing at all -- simply a static model. The large model could be rendered **once** into the background plane. The sphere that is changing is flagged as being in the foreground. The next time 3ds max needs to redraw the screen, instead of re-rendering both models, it can copy the background image (which has the large model already rendered) into the actual image buffer and only render those nodes that are flagged as being in the foreground (the moving sphere) on top of this background image. This provides the same visual result as rendering both models but is much quicker.

The key to making this system work is to figure out what items need to be in the foreground and what items may be put in the background. The system does this in almost all cases (the plug-in developer does not normally need to worry about this). If the user is moving a selected item in the scene, then the selected items (and all their dependents) go in the foreground. All other items may go in the background. For example, if the user selects a parent node to move, the children nodes must have to go in the foreground as well since they will move along with the parent. However everything else may go to the background. When the user plays back an animation any items that are animated go into the foreground and non-animated items may go in the background. When the user is in the Modify branch, any items that are dependent on the item being edited are put in the foreground, the others go into the background. Also, when a camera is being moved, all objects in the camera's viewport are tagged as being in the foreground, since they all must be redrawn.

## Methods

For specialized plug-ins, APIs are provided to allow the plug-in to control which nodes are put in the foreground when these plug-ins have created their own command mode. There is a method of the **CommandMode** class named **ChangeFGProc()** that can be used to specify a custom callback to process nodes that should be placed in the foreground. For example, if a utility plug-in needs to specify an additional set of nodes to go into the foreground plane along with the one it is currently working on, it could derive a class from **ChangeForegroundCallback** and implement the **callback()** method. In this method it could flag any nodes required (using a method of **INode** named **FlagForeground()**). In almost all cases, however, plug-ins that use a custom command mode can simply specify one of the two predefined values in their implementation of **ChangeFGProc()**. These are **CHANGE\_FG\_SELECTED** and **CHANGE\_FG\_ANIMATED**. See [Class CommandMode](#) for more details.

As a final note, if the user has turned off the Use Dual Plane toggle, there is still a notion of foreground/background objects -- only with dual planes off, some background objects end up being rendered. 3ds max uses a simple bounding box check: If a background object's bounding box intersects the foreground object's bounding box (at either the old or new position), then that background object is re-rendered. Otherwise, it isn't redrawn.

To see this dramatically, consider the example presented above: a small sphere and a large 500K polygon object. If the object is off to the right of the viewport, and the sphere is to the left (all by itself) then the sphere can be moved very quickly. But once it gets near enough to the large object that the bounding boxes intersect, interactivity drops dramatically because the large object must now be redrawn.

# Function Publishing System

[Class FPInterface](#), [Class FPMixinInterface](#), [Class FPInterfaceDesc](#), [Class Interface\\_ID](#), [Class FPFunctionDef](#), [Class FPPropDef](#), [Class FPActionDef](#), [Class FPPParamDef](#), [Class FPPParams](#), [Class FPEnum](#), [Class FPInterfaceCallback](#), [Class FPValue](#), [Class FPPParamOptions](#), [Class FPValidator](#), [Class ClassDesc2](#), [Class ActionTable](#).

## Abstract

### What is Function Publishing?

The Function Publishing System, new to 3ds max R4, is a new system that allows plugins to publish their major functions and operations in such a way that code outside the plugin can discover and make enquiries about these functions and is thus able to call them through a common calling mechanism. The whole system is very similar to Window's COM and OLE Automation systems and share many similar concepts in the architecture. However, the Function Publishing System is not based on COM and OLE but instead is a custom architecture more suited and optimized for MAX. The Function Publishing API serves a number of purposes which allow 3<sup>rd</sup> party developers to open up important portions of their plugins for use by external sources, allowing for users to extend and control these directly. Some of the purposes of the API are:

- • Modularizing plugin code into various "engines" that are able to supply services to other parts of 3ds max and other 3<sup>rd</sup> party plugins and can be delivered to the user through various different user-interfaces.
- • Providing automatic scriptability by exposing the published functions directly to MAXScript.
- • Providing alternate means of invoking plugin functions in the UI, such as via the new manipulator system, scripted menus, quad menus, hot keys, macroScripts, toolbar buttons, etc.
- • Allowing the MAXScript Macro Recorder to automatically generate calls to the published functions, in the event that these are invoked by the expanded ParamMap2 system or other UI mechanisms such as hot keys or

menu items.

- • Facilitate automatic generation of COM interfaces and OLE Type Libraries in such a way that external COM clients can invoke the published functions in the plugin code.

### **What would a plugin publish?**

The various kinds of functions published by a plugin usually fall into the following categories. You can, however, publish anything you want.

- • Important algorithms in the plugin, for example, the Edit Mesh modifier might publish its face extrude and mesh attach functions, or the flex modifier its soft-body dynamics algorithms. In these cases, the functions would be parameterized in the most general way, independent of any current scene state or UI mode in MAX, for example, the face extrude might take a Mesh, a set of faces and a distance.
- • Functions that enquire about or affect the state of one of the plugin's objects in the scene. Usually, these are unnecessary if the plugin stores its state as parameters in ParamBlock2s, which are already accessible externally, but in cases where this is not done or certain kinds of state are not cleanly accessible via the ParamBlock2 system, extra functions may be published by the plugin. These usually take an instance of one of the plugin's objects as one of their parameters, and should be independent of any UI mode, for example, they should not require that the object being manipulated be the current focus in the Command Panel. Prior to R4, plugins that wished to provide scripter access to internal functions would use the MAXScript SDK to provide scripter wrapper functions. In R4, the recommended and much simpler technique, is to use the Function Publishing system, as this not only provides automatic scriptability, but is a general mechanism for any external system to control and use the plugin.
- • UI action functions. These basically provide a programmatic way of "pressing" buttons and keys in the UI for a plugin and are specifically meant to be UI modal. They take no parameters, since these are defined by the current state in the UI. For example, the vertex delete action function for an Editable Mesh object would operate on the current vertex selection

for the current object in the Modify panel. These are not unlike the keyboard ShortcutTables that plugins could publish in R3, but by publishing them as action functions, any external system can effectively control the UI of a plugin. As well as being automatically exposed in the scripter like other published functions, Action functions are automatically entered into the new ActionTable system. This is a generalization of the R3 ShortcutTable system and basically holds all the commands and actions that may be bound to hotkeys or added to menus or put in buttons on a toolbar using the R4 CUI system, so that by publishing your action functions using the FnPub system, you are making them automatically available for binding to hotkeys and placing in menus and toolbar buttons. Action functions are treated specially in the FnPub system, and have extra descriptor data for things like menu item text, tooltip text, enable predicates, etc.

## Introduction to Function Publishing

### The Function Publishing API

The Function Publishing API consists of two main components, a function descriptor system and a function calling mechanism. The descriptor system is used by the publishing component to declare the functions it is publishing and provide necessary descriptive data and may be used by a component's client to enquire about published functions. The calling system is used by a component's client to invoke one of the published functions.

### Interfaces

Functions are published in one or more **Interfaces** by your plugin. If you have a large number of functions to be published, you might organize them functionally into different interfaces to make them more manageable for the user. For example, EditMesh might publish vertex functions, edge functions and face functions in separate interfaces. Action functions must be published in their own (set of) interface(s). Each interface is represented by an instance of a class derived from the base class **FPInterface**. You normally create these instances as static objects using the FPInterface constructor, in much the same way as ParamBlock2 descriptors are created. Each interface contains a list of the functions it publishes and the parameters they take, along with type and name

and other descriptive info for all these things. All the interfaces for a particular plugin class are kept in its ClassDesc object. An external system can find out about the interfaces you publish by calling various query methods on ClassDesc that access these interface definition objects. As well as these enquiry or 'reflection' methods, an FPIInterface also has the calling methods for actually invoking a particular function in the interface, so that if something has hold of one of your interfaces, it can call any of its published functions.

Additionally, a **Mixin** interface is provided which can be multiply-inherited by a plugin's class and returned via its implementation of the above new **GetInterface(Interface\_ID)** method, in the way that most existing object-based interfaces are now implemented.

The FPIInterface class is defined in the header file **maxsdk\include\iFnPub.h**, along with all the other FnPub classes & macros described in these notes.

### **Direct and Indirect Calling**

The FnPub system lets you set up interfaces so that functions in them can be called directly, as virtual member functions of the interface object, or indirectly via a dispatching method that takes a runtime function ID and a table of parameters. This is roughly equivalent to the dual vtable and IDispatch interface schemes in COM. Typically, you provide the virtual interface in a public header file so that it can be compiled against by other plugins and they can call the virtual functions on an interface object directly. The indirect call mechanism is used by MAXScript and other external systems that look for the published interface functions at runtime using the interface metadata.

### **Interface and Function IDs**

Each interface is uniquely identified by an **Interface\_ID**, which is a new class in the R4 SDK. This class is structurally very similar to Class\_ID, containing two randomly-chosen longwords to provide a unique global ID. It is defined in **maxsdk\include\maxtypes.h**

Each function in an interface is identified by an integer ID of type **FunctionID**. This is similar to the BlockID and ParamID's used in the paramblock2 system. This ID is used in the dispatch-based calling mechanism to identify the function

to call.

## Interface Organization

The map is bounded by **BEGIN\_FUNCTION\_MAP** and **END\_FUNCTION\_MAP** and contains one entry for each function in the interface. The map entry macros used come from a set named according to the number of arguments and whether the function is void. In this case, **FN\_2** is used for a 2 argument function returning a value and **VFN\_3** is used for a void function taking 3 arguments. The **FN\_2** macro, for example, takes the function ID, the return type, the virtual interface method name, and the types of the two arguments. **FN\_VA**, **VFN\_VA**, **FNT\_VA**, etc., **FUNCTION\_MAP** entry macro variants have same args as **\_0** macros and specify that the function takes a variable number of arguments (passed directly in **FPParams** instances). There are more details on these macros in the ref section below. There is a separate declaration macro, **DECLARE\_DESCRIPTOR(<class>)** that must be specified in the interface descriptor class. In the implementation class for a static or action or core interface, you provide a single descriptor declarator with the **DECLARE\_DESCRIPTOR()** macro, giving the current class as the macro parameter and a function map using **BEGIN\_FUNCTION\_MAP** and **END\_FUNCTION\_MAP** macros as before. The **DECLARE\_DESCRIPTOR(<class>)** is only required for **FPStaticInterface** subclasses; **FPMixinInterface** subclasses should only have the **FUNCTION\_MAP/END\_FUNCTION\_MAP** map table.

Within your plugin, each interface is usually organized into 3 separate sections of code:

- • The public virtual interface along with the `Interface_ID` and function ID definitions in a public header file.
- • the implementation interface, which inherits from the virtual interface, and contains both the function implementations and a function dispatch map used by the indirect calling methods. The `FnPub` system provides a set of macros to ease the definition of these maps.

- the interface descriptor, usually a static instance of the implementation interface. The constructor for this uses the same varargs technique used by the ParamBlockDesc2 constructor, enabling descriptive info for all the functions in the interface to be supplied in one constructor call.

Here's an (imaginary & simplified) example of an interface on EditMesh called 'FaceOps' with two functions, delete and extrude:

### 1. The public interface (in a public header file)

```
#include "iFnPub.h"
#define EM_FO_INTERFACE Interface_ID(0x434455, 0x65654)
#define GetEMFaceOpsInterface(cd) \
    (EMFaceOps *)(cd)->GetFPInterface(EM_FO_INTERFACE)

enum { em_delete, em_extrude, };

class EMFaceOps : public FPStaticInterface
{
public:
    virtual int Delete (Mesh* m, BitArray* faces)=0;
    virtual void Extrude(Mesh* m, BitArray* faces, float amt)=0;
};
```

This defines the Interface\_ID for the 'FaceOps' interface and provides a helper macro for getting the interface pointer from the plugin's ClassDesc. This is followed by an enum for the two function IDs and then the virtual interface class itself in which each function is declared as a pure virtual function. The return and parameter types of these functions are restricted to a fixed set, defined below. Fixing this set is necessary so that systems that access the interface metadata or use the dispatch form of calling have a known set of types to deal with. External code that wishes to call one of these functions, and has access to this public header can do so as in the following example:

```
EMFaceOps* efi = GetEMFaceOpsInterface(edmeshCD);
```

```
...
```

```
efi->Extrude(mesh, faces, 10.0);
```

To call this function indirectly, you call the `Invoke()` method on the interface, giving it the function ID and an **FPParams** object containing all the parameters, as in the following example:

```
FPParams p (3, TYPE_MESH, mesh,  
            TYPE_BITARRAY, faces,  
            TYPE_FLOAT, 10.0);
```

```
FPValue result;
```

```
FPInterface efi = edmeshCD->GetFPInterface("FaceOps");
```

```
efi->Invoke(em_extrude, result, &p);
```

```
x = result.i;
```

Any function result is passed back in an **FPValue** instance, given as one of the args to `Invoke`, which is a type-tagged variant structure that can hold any one of the FnPub supported types. The **FPParams** class contains a convenience `varargs` constructor for building parameter sets as shown. An **FPParams** instance contains a **Tab<>** of **FPValue** instances, each holding a parameter. This example demonstrates how to look up an interface by name in a `ClassDesc`. The `Invoke()` function has several overloads, for calling functions with and without results and parameters.

## **2. The implementation class (in the .cpp implementation file)**

```
class EMFaceOpsImp : public EMFaceOps  
{  
    DECLARE_DESCRIPTOR(EMFaceOpsImps)  
  
    BEGIN_FUNCTION_MAP
```

```

    FN_2(em_delete, TYPE_INT, Delete, TYPE_MESH,
        TYPE_BITARRAY)
    VFN_3(em_extrude, Extrude, TYPE_MESH, TYPE_BITARRAY,
        TYPE_FLOAT)
    END_FUNCTION_MAP

```

```

int Delete(Mesh* m, BitArray* faces)
{
    //... do the delete
    return face_count;
}
void Extrude(Mesh* m, BitArray* faces, float amt)
{
    //... do the extrude
}
};

```

The interface implementation class specializes the virtual interface class defined in the public header. It can contain any implementation-required data members and utility methods, but must at least contain implementations for each of the virtual methods defined in the virtual interface class.

The key component shown above is the **function map** which generates the indirect call dispatcher used by the Invoke() method. This dispatcher unbundles the parameters from the indirect call parameter structure, forwards them to the correct implementation method and bundles the return value into an **FPValue**. It is specified here using the map macros that come with the FnPub system (in **iFnPub.h**), in a manner somewhat similar to the message maps in MFC. You can implement this dispatcher by hand, but it is advised that you use the map macros.

The map is bounded by **BEGIN\_FUNCTION\_MAP** and **END\_FUNCTION\_MAP** and contains one entry for each function in the interface. The map entry macros used come from a set named according to the

number of arguments and whether the function is void. In this case, **FN\_2** is used for a 2 argument function returning a value and **VFN\_3** is used for a void function taking 3 arguments. The **FN\_2** macro, for example, takes the function ID, the return type, the virtual interface method name, and the types of the two arguments. There are more details on these macros in the ref section below. There is a separate declaration macro,

**DECLARE\_DESCRIPTOR(<class>)** that must be specified in the interface descriptor class. In the implementation class for a static or action or core interface, you provide a single descriptor declarator with the **DECLARE\_DESCRIPTOR()** macro, giving the current class as the macro parameter and a function map using **BEGIN\_FUNCTION\_MAP** and **END\_FUNCTION\_MAP** macros as before. The **DECLARE\_DESCRIPTOR(<class>)** is only required for **FPStaticInterface** subclasses; **FPMixinInterface** subclasses should only have the **FUNCTION\_MAP/END\_FUNCTION\_MAP** map table.

### 3. The interface definition (in the .cpp implementation file)

```
static EMFaceOpsImp emfi (  
    EM_FO_INTERFACE, _T("FaceOps"), IDS_EMFO,  
&edmeshCD, 0,  
    em_delete, _T("delete"), IDS_DELETE, TYPE_INT, 0, 2,  
    _T("mesh"), IDS_MESH, TYPE_MESH,  
    _T("faces"), IDS_FACES, TYPE_BITARRAY,  
    em_extrude, _T("extrude"), IDS_EXTRUDE, TYPE_VOID, 0, 3,  
    _T("mesh"), IDS_MESH, TYPE_MESH,  
    _T("faces"), IDS_FACES, TYPE_BITARRAY,  
    _T("amount"), IDS_AMOUNT, TYPE_FLOAT,  
    end  
);
```

A distinguished instance of the interface implementation class is constructed and registered with the owning ClassDesc. This is the instance given out when the CD is asked for an interface via its **GetFPInterface()** method. The

instance has data members that contain all the metadata for the interface. You normally build this instance as a static in the implementing .cpp file, in a manner similar to ParmBlock2 descriptor instances.

In the example above, the instance 'emii' is statically declared using the FPIInterface's varargs constructor. The initial arguments to this constructor provide the **Interface\_ID**, the internal fixed name, localizable descriptor string resID, owning ClassDesc and flags bits. This is followed by a list of function descriptors which themselves each have lists of parameter descriptors. The **em\_delete** function description, for example, consists of a fixed name, a localizable description string resID, a return type, flag bits and a count of the number of parameters. For each parameter, there is a line giving parameter name, description string resID and type.

### **FnPub Object Based Mixin Interfaces**

A variant of the FPIInterface, known as a Mixin interface, is provided and can be multiply-inherited by a plugin's class and returned via its implementation of the above new **GetInterface(Interface\_ID)** method, in the way that most existing object-based interfaces are now implemented. Here's an example setup. First, the public header that would be used by an SDK-level client of the interface:

```
// interface ID  
#define FOO_INTERFACE Interface_ID(0x342323, 0x55664)  
  
#define GetFooInterface(obj) \  
    ((FooInterface*)obj->GetInterface(FOO_INTERFACE))  
  
// function IDs  
enum { foo_move, foo_setRadius, };  
  
// mixin interface  
class FooInterface : public FPMixinInterface  
{
```

```

BEGIN_FUNCTION_MAP
    VFN_1(foo_move, Move, TYPE_POINT3);
    VFN_1(foo_setRadius, SetRadius, TYPE_FLOAT);
END_FUNCTION_MAP

```

```

FPInterfaceDesc* GetDesc();

virtual void SetRadius(float radius)=0;
virtual void Move(Point3 p)=0;
};

```

This is much the same as existing stand-alone FnPub interfaces, except that the function map is put into the virtual interface class, rather than the implementing interface class. Then, in the plugin class, you inherit the mixin interface (you are "mixing" it into the class). You provide implementations for the interface's virtual methods in the main class and an implementation of **GetInterface()** that returns the object cast to the interface, exactly as you would have done for old-style object-based interfaces:

```

class MyObject : public SimpleObject2, public FooInterface
{
public:
    ...
    void SetRadius(float radius);
    void Move(Point3 p);
    ...
    FPInterface* GetInterface(Interface_ID id) {
        if (id == FOO_INTERFACE)
            return (FooInterface*)this;
        else
            return SimpleObject2::GetInterface(id);
    }
}

```

```
...  
}
```

Note that the **GetInterface()** method needs to cast the 'this' to the mixin interface class so the correct vtable is used (this was the case with old-style object interfaces, as well). It also calls **SimpleObject2::GetInterface()** if the id doesn't match so that other base class interfaces and stand-alone interfaces, if any, are made available to the caller. Finally, you provide a descriptor for the interface, as with stand-alone interfaces, but in this case, you must make it an instance of **FPInterface**, not **FooInterface** or **MyObject**, and must specify the **FN\_MIXIN** flag to denote the interface as mixin:

```
static FPInterfaceDesc foo_mixininterface(FOO_INTERFACE,  
    _T("foo"), 0,  
    &myObjDesc, FP_MIXIN,  
    foo_move, _T("move"), 0, TYPE_VOID, 0, 1,  
    _T("vector"), 0, TYPE_POINT3,  
    foo_setRadius, _T("setRadius"), 0, TYPE_VOID, 0, 1,  
    _T("radius"), 0, TYPE_FLOAT,  
    end  
);  
  
FPInterfaceDesc* FooInterface::GetDesc()  
{  
    return &foo_mixininterface;  
}
```

This static instance provides the interface metadata and is recorded in the **ClassDesc**. To access the interface metadata, you must get this instance directly from the **ClassDesc** via **ClassDesc::GetInterface()**, and not via **Animatable::GetInterface()** on an object.

All this is pretty-much the same work you would do to provide an **Animatable::GetInterface()** interface currently in MAX. The extra stuff is

the **FUNCTION\_MAP** and the interface descriptor. The SDK-level clients use the interface in exactly the same way, but now the scripter and other external systems can find and use these interfaces automatically at runtime.

Mixin object-based interfaces are accessible in the scripter in similar way to stand-alone published interfaces. In particular, the functions are available in a struct function package which is itself accessed as a property on the plugin class object. For example, if the above example plugin class is named 'MyObject' in the scripter, the functions in its 'foo' mixin interface would be accessed as:

**MyObject.foo.move**

**MyObject.foo.setRadius**

The functions are effectively 'generic' functions that require an instance of the plugin as the first argument, resulting in:

**MyObject.foo.move \$baz [10,10,10]**

**MyObject.foo.setRadius \$bar 123.4**

### **Publishing Mixin Interface on Arbitrary Classes**

You can publish FnPub mixin interfaces on any class, not just Animatable subclasses. To do this, you inherit from, **IObject**, a new virtual base class in the FnPub system. This provides an API for querying and iterating FPMixinInterfaces that the class wishes to publish, similar to **Animatable::GetInterface()**. The API that the publishing class must implement is as follows:

```
class IObject  
{  
public:  
    // object/class name  
    virtual TCHAR* GetName()=0;
```

```

// iterate over all interfaces...
virtual int NumInterfaces()=0;
virtual FPInterface* GetInterface(int i)=0;

// get ID'd interface
virtual FPInterface* GetInterface(Interface_ID id)=0;

// IObject ref management (can be implemented by
dynamically-
// allocated IObjects for ref-count based lifetime control)
virtual void Acquire() { };
virtual void Release() { };
};

```

There is a corresponding new ParamType2 type code, TYPE\_IOBJECT, that allows instances of these classes to be passed and returned in **FPInterface** methods, providing a simple form of user-defined type, in the sense that these instance collections are passed as interfaces rather than pointers (similar to COM). MAXScript has been extended to provide wrapper value classes for IObjects and so this mechanism provides a light-weight alternative to the MAXScript SDK facilities for adding new wrapper value classes to the scripter.

MAXScript also calls the **Acquire()** and **Release()** methods on IObjects as it creates and collects these wrappers, so that IObject objects can keep track of MAXScript's extant references to them. For example:

```

class IFoo1 : public FPMixinInterface
{
    virtual void Frabulate(Point3 p)=0;
    ...
};

class IFoo2 : public FPMixinInterface

```

```

{
    ...
};

class Foo : public IObject, public IFoo1, public IFoo2
{
    // Foo methods
    ...
// IObject methods
    TCHAR* GetName() { return _T("Foo"); }
    int NumInterfaces() { return 2; }
    FPIInterface* GetInterface(int i)
    {
        if (i == 0) return (IFoo1*)this;
        if (i == 1) return (IFoo2*)this;
    }
    FPIInterface* GetInterface(Interface_ID id)
    {
        if (id == FOO_INTERFACE_1) return (IFoo1*)this;
        if (id == FOO_INTERFACE_2) return (IFoo2*)this;
    }
// IFoo1 methods
    void Frabulate(Point3 p) { ... }
    ...
// IFoo2 methods
    ...
};

```

Instances of Foo can be passed as parameters and results in FnPub interface descriptors and function maps by declaring them as **TYPE\_IOBJECT** values, and they will get cast to **IObject\*** automatically. For example, in a plugin that

uses the Foo class, a method in one of its FPInterfaces might want to return a Foo instance. The method might be defined as:

```
Foo* GetFoo(INode* object)  
{  
    Foo* x = new Foo (...);  
    ...  
    return x;  
}
```

Since Foo is not a supported base ParamType2 type but is derived from **IObject**, the return value of the above method can be declared using **TYPE\_IOBJECT** in the **FUNCTION\_MAP** as:

```
FN_1(my_getFoo, TYPE_IOBJECT, GetFoo, TYPE_INODE);
```

and in the descriptor constructor as:

```
my_getFoo, _T("getFoo"), 0, TYPE_IOBJECT, 0, 1,  
_T("object"), 0, TYPE_INODE,
```

In MAXScript, calling getFoo() would return an **IObject** value that has two interface properties, ifoo1 and ifoo2, and these in turn would expose their methods as properties:

```
f = getFoo $  
f.ifoo1.frabulate [10,0,0] // call the IFoo1 frabulate method
```

### **Passing FPInterfaces as Parameters & Results**

FPInterfaces themselves can now be passed directly as parameters and results via the new type, **TYPE\_INTERFACE**. These turn up in MAXScript as instances of a new value class, **FPInterface**, with all the interface's methods accessible as properties.

In cases where you have a class publishing a single mixin interface, it is possible to pass instances directly as **TYPE\_INTERFACE** and let the `FPInterface*` type-cast implied by **TYPE\_INTERFACE** extract the mixin interface. This is a useful simple alternative to the **IObject** scheme for publishing a mixin interface on non-Animatables, since you don't actually need to implement the **IObject** protocol in these cases. If the Foo example above had a single mixin, it would publish it in a similar way to the following:

```
class IFoo : public FPMixinInterface
{
    virtual void Frabulate(Point3 p)=0;
    ...
};

class Foo : public OtherBaseClass, public IFoo
{
    // Foo methods
    ...
    // IFoo methods
    void Frabulate(Point3 p) { ... }
    ...
};
```

The same `Foo*`-returning **GetFoo()** method in the previous example would then be declared using **TYPE\_INTERFACE** in the **FUNCTION\_MAP** as:

```
FN_1(my_getFoo, TYPE_INTERFACE, GetFoo, TYPE_INODE);
```

and in the descriptor constructor as:

```
my_getFoo, _T("getFoo"), 0, TYPE_INTERFACE, 0, 1,  
_T("object"), 0, TYPE_INODE,
```

In this case, calling **getFoo()** in MAXScript would return an **FPInterface** value that exposes the **IFoo** methods directly as properties.

```
f = getFoo $  
f.frabulate [10,0,0] // call the Foo1 frabulate method
```

Note again this scheme won't work for class publishing multiple multiple mixins since implied **FPInterface\*** cast becomes ambiguous - you must use the **IObject** mechanism in this case.

### **FP\_CORE Interfaces**

There is an interface descriptor flag, **FP\_CORE**, which must be specified on Core interface descriptors. Static Core interface descriptors are now automatically registered with the Core, so you do not need to explicitly call **RegisterCOREInterface()** on them. The code examples below include a Core interface using this flag. The following example taken from the DragAndDrop manager interface.

#### **1. The public interface definition (in a public header file)**

```
class IDragAndDropMgr : public FPStaticInterface  
{  
public:  
    virtual void EnableDandD(BOOL flag)=0;  
    virtual BOOL IsEnabled()=0;  
    virtual BOOL EnableDandD(HWND hwnd, BOOL flag,  
        DragAndDropHandler* handler = NULL)=0;  
    virtual BOOL DropPackage(HWND hwnd, POINT& point,  
        URLTab& package)=0;  
    virtual BOOL DownloadPackage(URLTab& package,  
        TCHAR* directory,
```

```
    HWND hwnd = NULL)=0;  
    virtual TCHAR* GetStdDownloadDirectory()=0;  
};
```

## 2. The implementation class (in the .cpp implementation file)

Note in this example, a couple of the interface methods use a new type, URLTab, which is not a ParamType2 type. This is a Tab<TCHAR\*> specialization and so provides overloads that take Tab<TCHAR\*>s and convert them to URLTabs, specifically for dispatch-based calls.

```
class DragAndDropMgr : IDragAndDropMgr  
{  
    ...  
public:  
    void EnableDandD(BOOL flag) { global_enable = flag; }  
    BOOL IsEnabled() { return global_enable; }  
    BOOL EnableDandD(HWND hwnd, BOOL flag,  
        DragAndDropHandler* handler = NULL);  
    BOOL DropPackage(HWND hwnd, POINT& point, URLTab&  
package);  
    BOOL DropFiles(HWND hwnd, HDROP hDrop);  
    BOOL DownloadPackage(URLTab& package, TCHAR*  
directory,  
        HWND hwnd = NULL);  
    TCHAR* GetStdDownloadDirectory();  
  
    // variants for FnPub interface that take Tab<TCHAR*>s  
    BOOL DropPackage(HWND hwnd, POINT& point,  
Tab<TCHAR*>& package);  
    BOOL DownloadPackage(Tab<TCHAR*>& package,  
TCHAR* directory);
```

```

DECLARE_DESCRIPTOR(DragAndDropMgr)

// dispatch map
BEGIN_FUNCTION_MAP
    VFN_1(dndmgr_globalEnableDnD, EnableDandD,
TYPE_BOOL);
        FN_0(dndmgr_isEnabled, TYPE_BOOL, IsEnabled);
        FN_2(dndmgr_enableDandD, TYPE_BOOL,
EnableDandD,
            TYPE_HWND, TYPE_BOOL);
        FN_3(dndmgr_dropPackage, TYPE_BOOL,
DropPackage,
            TYPE_HWND, TYPE_POINT_BR,
            TYPE_STRING_TAB_BR);
        FN_2(dndmgr_downloadPackage, TYPE_BOOL,
DownloadPackage,
            TYPE_STRING_TAB_BR, TYPE_STRING);
        FN_0(dndmgr_downloadDirectory, TYPE_STRING,
            GetStdDownloadDirectory);
END_FUNCTION_MAP
    ...
};

```

### 3. The descriptor, note the FP\_CORE flag.

FP\_CORE descriptors are automatically registered with **RegisterCOREInterface()**.

```

DragAndDropMgr dragAndDropMgr(DND_MGR_INTERFACE,
_T("dragAndDrop"),
    IDS_DND_INTERFACE, NULL, FP_CORE,
    dndmgr_globalEnableDnD, _T("globalEnableDragAndDrop"),

```

```

0,
    TYPE_BOOL, 0, 1, _T("onOff"), 0, TYPE_BOOL,
    dndmgr_isEnabled, _T("isEnabled"), 0, TYPE_BOOL, 0, 0,
    dndmgr_enableDandD, _T("enableDragAndDrop"), 0,
TYPE_BOOL, 0, 2,
    _T("window"), 0, TYPE_HWND, _T("onOff"), 0,
    TYPE_BOOL,
    dndmgr_dropPackage, _T("dropPackage"), 0, TYPE_BOOL, 0,
3,
    _T("window"), 0, TYPE_HWND, _T("mousePoint"), 0,
    TYPE_POINT_BR, _T("files"), 0,
    TYPE_STRING_TAB_BR,
    dndmgr_downloadPackage, _T("downloadPackage"), 0,
TYPE_BOOL, 0,
    2, _T("files"), 0, TYPE_STRING_TAB_BR,
    _T("directory"), 0,
    TYPE_STRING,
    dndmgr_downloadDirectory, _T("geStdtdownloadDirectory"), 0,
    TYPE_STRING, 0, 0,
end

```

## Action Interfaces

A special kind of interface is the Action Interface. These interfaces only contain UI Action Functions that provide a programmatic way of "pressing" buttons and keys in the UI for a plugin. As mentioned in the opening section of this doc, these action functions have certain special characteristics and possess additional descriptive metadata, relative to the functions we've described so far. Here's an annotated Action Interface example for the EdMesh system we looked at above.

### 1. The public interface (in a public header file)

```
#include "iFnPub.h"
```

```

#define EM_ACT_INTERFACE Interface_ID(0x65678, 0x123)
#define GetEMAActionsInterface(cd) \
    (EMAActions*)(cd)->GetFPInterface(EM_ACT_INTERFACE)

enum {
    ema_create, ema_create_enabled, ema_create_checked,
    ema_delete, ema_delete_enabled,
};

class EMAActions: public FPInterface
{
public:
    virtual FPStatus Create()=0;
    virtual FPStatus Delete()=0;
    virtual BOOL IsCreateEnabled()=0;
    virtual BOOL IsCreateChecked()=0;
    virtual BOOL IsDeleteEnabled()=0;
};

```

As with normal function interfaces, we we have an **Interface\_ID** defined and interface accessor helpers. There is also a function ID enum, but in this case there are more IDs than published actions. Each action function can have associated with it up to 3 predicate functions that users of the interface can call to determine status for the action. These are the **isEnabled**, **isChecked**, and **isVisible** predicates and they are used mostly by the UI elements that can be bound to an interface action to determine things like whether a button or menu item should greyed or a checkbutton should be highlighted or whether the action should be added to an about-to-be-displayed pop-up menu. The **isEnabled** & **isVisible** predicates are optional and default to always TRUE. The **isChecked** predicate is for actions that toggle a state, for example starting & stopping a mouse command mode. This predicate is also optional and defaults to FALSE, but it is required if the action is bound to **TYPE\_CHECKBUTTON** via the ParamMap2 system.

Declarations for action functions along with any associated predicates are supplied in the virtual interface class. Action functions always take zero arguments and return an **FPStatus** result which indicates whether the action was successfully performed or not (success if **FPS\_OK**, but it may return values such as **FPS\_ACTION\_DISABLED** if the action was not enabled at the time). Predicate functions always take zero arguments and return a **BOOL**.

Here are some action function call examples:

```
EMActions* emai = GetEMActionsInterface(edmeshCD);
```

```
...
```

```
if (emai->IsCreateEnabled()) // direct calls  
    emai->Create();
```

Indirect Calls:

```
if (emai->IsEnabled(ema_delete))  
    emai->Invoke(ema_delete);
```

This indirect example uses one of the predicate helper functions to call the **isEnabled** predicate in the interface for a given function, specified by FunctionID.

## **2. The implementation class (in the .cpp implementation file)**

```
class EMActionsImp : public EMActions  
{  
public:  
    BOOL creating;  
  
DECLARE_DESCRIPTOR(EMActionsImp)  
BEGIN_FUNCTION_MAP
```

```
FN_ACT(ema_create, Create)  
    FN_PRED(ema_create_enabled, IsCreateEnabled)  
    FN_PRED(ema_create_checked, IsCreateChecked)  
FN_ACT(ema_delete, Delete)  
FN_PRED(ema_delete_enabled, IsDeleteEnabled)  
END_FUNCTION_MAP
```

```
void Init()
```

```
{  
    // initialize any state data  
    creating = FALSE;  
}
```

```
    FPStatus Create()
```

```
{  
    //... start or stop the create mode  
    //...  
    creating = !creating; // toggle creating state  
    //...  
    return FPS_OK;  
}
```

```
BOOL IsCreateEnabled()
```

```
{  
    //... determine if create enable, perhaps  
    // vertex subobjlevel test, etc.  
}
```

```
BOOL IsCreateChecked()
```

```
{
```

```

    return creating; // the current create state
}
//...
};

```

The action interface implementation class contains implementations for the actions and predicates, as required by the virtual interface. It also contains a **FUNCTION\_MAP**, but in this case the actions and predicates entries in the map are defined with their own special FN\_ macros, **FN\_ACT** for actions and **FN\_PRED** for predicates.

The Create action in this example corresponds to a vertex create mode start button and so it provides an **isChecked** predicate to tell what state the action is in, in this case checked => in create mode, not checked => not in create mode. The implementation interface contains a data element 'creating' to help keep track of this state. Note that it is initialized in a method called **Init()**. Since the varargs constructor is the normal way the interface instance is created, it guarantees to call this (virtual) init method for you; alternatively, you could make it a static data member and statically initialize it.

### 3. The interface definition (in the .cpp implementation file)

```

static EMActionsImp emai (
    EM_ACT_INTERFACE, _T("Actions"), IDS_EMAI,
    &edmeshCD, FP_ACTIONS,
    ema_create, _T("create"), IDS_CREATE, 0,
    f_predicates, ema_create_enabled, ema_create_checked,
    FP_NO_FUNCTION,
    f_category, _T("creation"), 0,
    f_iconRes, IDI_CREATE_ICON,
    f_toolTip, IDS_VERTEX_CREATE_BY_CLICKIN,
    f_ui, em_params, 0, TYPE_CHECKBUTTON, IDC_CREATE,
    GREEN_WASH,

```

```

    end,
    ema_delete, _T("delete"), IDS_DELETE, 0,
        f_isEnabled, ema_delete_enabled,
        f_ui, em_params, 0, BUTTON, IDC_DELETE,
    end,
end
);

```

The constructor for an Action Interface follows a slightly different syntax to a normal function interface. The header parameters are the same (**Interface\_ID**, internal name, descriptor, CD, flags), the **FP\_ACTIONS** flag must be specified to mark this as an action interface.

Each action function is specified by header parameters giving **FunctionID**, internal name, descriptor **resID** and flag bits. This is followed by one or more tagged action function options, in the same manner as parameter options in a **ParamBlockDesc2** constructor. Each option defines a separate item of metadata for the function. In the case of **ema\_create**, the following are specified:

- • 2 of the 3 possible predicates via their FunctionIDs,
- • a category string, used by the UI customize dialogs to arrange actions into category groups
- • a resID for an Icon resource that will be made available for use in toolbar buttons and other UI locations
- • a toolTip string resID
- • a UI specification that allows the parammap2 system to automatically connect a button to this action. This is described in more detail in the next section.

The possible options include:

**f\_category** category name, as internal TCHAR\* and localizable string resID,

defaults to interface name

**f\_predicates** supply 3 functionIDs for **isEnabled**, **isChecked**, **isVisible**

predicates

**f\_isEnabled** **isEnabled** predicate functionID

**f\_isChecked** **isChecked** predicate functionID

**f\_isVisible** **isVisible** predicate functionID

**f\_iconRes** icon as resource ID

**f\_icon** icon as UI .bmp filename, index pair, as per CUI icon specifications

**f\_buttonText** button text string resID, defaults to function description

**f\_toolTip** tooltip string resID, defaults to function description

**f\_menuText** menu item text string resID, defaults to buttonText or function

description

**f\_ui** UI spec if paramMap2-implemented UI (pmap blockID, mapID, control

type, button or checkbutton resID, hilight col if chkbtn)

**f\_shortcut** default keyboard shortcuts for action functions

**f\_keyArgDefault** marks a parameter as optional and supplies a default value. Optional parameters must come after the positional parameters. Example:

```
Meshop::buildMapFaces,  
  _T("buildMapFaces"), 0, TYPE_VOID, 0, 3,  
    _T("source"), 0, TYPE_FPVALUE_BR,  
      _T("keep"), 0, TYPE_BOOL,  
f_keyArgDefault, FALSE,  
    _T("channel"), 0, TYPE_INT,  
f_keyArgDefault, 0,
```

**f\_inOut** specifies whether **\_BR** parameter is just for input, just for output or both via **FPP\_OUT\_PARAM**, **FPP\_IN\_PARAM**, or **FPP\_IN\_OUT\_PARAM** argument. Default is **FP\_IN\_OUT\_PARAM**.

**f\_macroEmitter** Supply a pointer to **FPMacroEmitter** subclass instance to customize macro-recorder emission. The **Emit()** method is called on this instance with interface and function def pointers for the particular action function to be emitted.

The ActionTable system 'table ID' DWORD for Action interfaces is generated automatically from the Interface\_ID for the Action interface (by xor-ing the two DWORDS in the interface ID). If this generates an ID that clashes with built-in table IDS (in the range 0-64), an error message is generated and you should choose a more random Interface\_ID.

You define the UI context for the interface as a whole, rather than for each individual action function. The context ID should be specified immediately after the flag word in an Action interface descriptor constructor. For example:

```
static IKChainActionsImp sIKChainActionImp(  
    IKCHAIN_FP_INTERFACE_ID, _T("Action"),  
    IDS_IKCHAIN_ACT,  
    &theIKChainControlDesc, FP_ACTIONS,  
    kActionMainUIContext,  
    IKChainActionsImp::kSnap, _T("snap"),  
    IDS_IKCHAIN_SNAP, 0,  
    ...
```

The constant '**kActionMainUIContext**' is one of the built-in ActionTable system contexts that you can use (defined in **maxsdk\include\ActionTable.h**), and is most likely the one to be used. If you create an entirely new context and context ID, you will need to register and control its activation yourself (see the **IActionManager** in **maxsdk\include\ActionTable.h** for details).

The automatically-generated ActionTable for the Action interface can be accessed via a new FPInterface method: **virtual ActionTable\***

### **GetActionTable();**

By default, the action table is activated immediately and stays active throughout the 3ds max session. You can dynamically control this activation via a new FPInterface method: **virtual void EnableActions(BOOL onOff);**

This enables or disables the entire set of actions in the interface. You might do this if the actions are only to be active during certain times in the running of MAX. Usually, this control is achieved via ActionTable contexts.

### **ParamMap2 Buttons**

The **f\_ui** option for an Action Function can be used to make the **ParamMap2** system connect the action function to a button in a ParamMap2-mediated rollup in your plugin's UI. This means you have to be using the ParamBlock2 system in the interface's plugin (ie, same **ClassDesc2**) and have at least one paramblock containing a paramMap whose rollup contains the button dialog item.

In the **f\_ui** option, you first specify the BlockID of the paramblock containing this paramMap and then MapID of the map within that block (single map blocks use MapID 0). This is followed by a control type code, and its required parameters, as listed below:

#### **TYPE\_BUTTON, <dlg\_item\_ID>**

a standard push button. The dialog item must be a 3ds max **CustButton** custom control.

#### **TYPE\_CHECKBUTTON, <dlg\_item\_ID>, <highlight\_color>**

a standard check button. Again, the dialog item must be a 3ds max **CustButton** custom control, the 2<sup>nd</sup> parameters is the button highlight color as a **COLORREF** word, either one of the predefined colors in **maxsdk\include\custcont.h**, or an RGB value using the system **RGB()** macro.

This is basically all you need to do. Whenever the specified map opens its rollup, say as part of a **BeginEditParams**, it will automatically look for any Action functions associated via an **f\_ui** and will call that action function whenever the button is clicked. If an **isEnabled** predicate is supplied, the **ParamMap2** will call it at various display update times to determine whether to enable or disable (gray out) the button. For CheckButtons, an **isChecked** predicate *must* be supplied and will be called at similar times to update the pressed or not-pressed state of the button display.

## The FPInterface Class Hierarchy

The **FPInterface** class hierarchy is such that static and mixin interfaces and interface descriptors each have their own types.

**FPInterface** The base class for all interfaces, prime client type for using interfaces

**FPInterfaceDesc** Contains the interface metadata

**FPStaticInterface** Use as the base class for defining static or core virtual interface classes.

**FPMixinInterface** Use as the base class for defining object-based mixin interface

classes, use **FPInterfaceDesc** for mixin interface descriptors

The **FPInterface** class continues to provide all the original method and predicate invocation functions required by an SDK-level client of the interface. A pure virtual method, **GetDesc()**, should be used to get the **FPInterfaceDesc** instance for the interface. The metadata definition and accessing methods and the metadata data members now all live in the **FPInterfaceDesc**.

All the original interface query methods in the API (such as **Animatable::GetInterface(id)**, **ClassDesc2::GetInterface(id)**, **GetCOREInterface(id)**) still return **FPInterface** pointers, so you would use

**GetDesc()** in constructs like the following to get at an interface's metadata:

```
FPInterface* fpi = GetCOREInterface(FOO_INTERFACE);  
TCHAR* iname = fpi->GetDesc()->internal_name.data();
```

The **FPInterface::GetDesc()** method has a default implementation in **FPInterfaceDesc** (and so **FPStaticInterface**), simply returning 'this', since instances of these classes contain the metadata). **FPMixinInterface** subclasses provide implementations of **GetDesc()**, returning their associated **FPInterfaceDesc** instance.

## Parameter Validation

An interface descriptor can now contain validation information for individual parameters, so that clients such as MAXScript can validate values given as parameters to **FPInterface** calls, prior to making the call. The validation info can be in the form of a range of values for int and float types, or more generally, a validator object that is called to validate a parameter value.

MAXScript now applies the validations if supplied and generates descriptive runtime errors if the validation fails.

The validation info is specified in the **FPInterface** descriptor in optional tagged entries following the parameters to be validated. The two possible tags are **f\_range** and **f\_validator**. Here's an example from a possible mixin interface to Cylinder:

```
static FPInterfaceDesc cylfpi (  
    CYL_INTERFACE, _T("cylMixin"), 0, &cylinderDesc,  
    FP_MIXIN,  
    ...  
    cyl_setRadius, _T("setRadius"), 0, TYPE_VOID, 0, 1,  
        _T("radius"), 0, TYPE_FLOAT, f_range, 0.0, 10000.0,  
    cyl_setDirection, _T("setDirection"), 0, TYPE_VOID, 0, 1,  
        _T("vector"), 0, TYPE_POINT3, f_validator,  
        &cylValidator,  
)
```

```
...
end
);
```

In this above example, the "radius" parameter is defined to have a range 0.0 to 10000.0. An **f\_range** spec can only be used for int (**TYPE\_INT**, **TYPE\_TIMEVALUE**, **TYPE\_RADIOBTN\_INDEX**, **TYPE\_INDEX**, **TYPE\_ENUM**) and float parameters (**TYPE\_FLOAT**, **TYPE\_ANGLE**, **TYPE\_PCNT\_FRAC**, **TYPE\_WORLD**, **TYPE\_COLOR\_CHANNEL**) and is given as a pair of low and high range values. The values must be floating point or integer as needed by the **TYPE\_xxx** code, you cannot specify integer range values for float types and vice versa, hence the 0.0 and 10000.0 in the example above. MAXScript checks parameter values against these supplied ranges and will generate a descriptive error message for out-of-range values.

The "vector" parameter in the above example has a validator object specified. This must be a pointer to an instance of a class derived from the new class, **FPValidator**, defined in **iFnPub.h**. This is a virtual base class, containing a single method, **Validate()**, that is called to validate a prospective value for a parameter. You would typically subclass **FPValidator** in your code and provide an implementation of **Validate()** to do the validation. Here is the **FPValidator** virtual base class:

```
class FPValidator
{
public:
    // validate val for the given param in function in interface
    virtual bool Validate(FPInterface* fpi, FunctionID fid,
        int paramNum, FPValue& val, TSTR& msg)=0;
};
```

The **Validate()** function is called with interface, function-within-interface and parameter-within-function identifiers and an **FPValue** to validate. It can optionally install an error message string in the 'msg' TSTR& parameter for the

user of the validator to display. If there are many parameters to validate this way, you can choose to provide a separate subclass for each parameter or a single subclass and switch on the parameter identification supplied.

For the Cylinder example above, the "vector" parameter might be required to be given as a unit vector, so the validator might check for this as follows:

```
class CylValidator : public FPValidator
{
    bool Validate(FPInterface* fpi, FunctionID fid, int paramNum,
               FPValue& val, TSTR& msg)
    {
        if (fabs(Length(*val.p) - 1.0) > 1e-6)
        {
            msg = "Direction vector must be unit length."
            return false;
        }
        else
            return true;
    }
}
```

Note that the type is already checked by the caller, since it has type info for the parameter, so you don't also need to check the **FPValue** type.

Note, also, that the **FPValue& val** argument given to **Validate()** is a reference to the actual parameter value to be given to the called function, so it is also possible to massage the value rather than reporting an error. For example, the above sample **Validate()** method could normalize the vector and always return true. The type must not be changed in this process, only the value can be adjusted.

A singleton instance of this **FPValidator** subclass would be created, typically as a

static instance, to be given in the **f\_validator** specification: **static CylValidator cylValidator;**

If you need to get at the validation metadata, it is available in the new 'options' data member in an **FPParamDef** instance for a parameter. If non-NULL, this points to a **FPParamOptions** instance which contains the range or validator information for the parameter. See class **FPParamOptions** in **iFnPub.h** for details.

## Exception Handling

FnPub interface functions can now report fatal error conditions to callers by using C++ exception-handling. There is a new exception base class, **MAXException**, defined in **iFnPub.h** that can be thrown directly, or subclassed as needed for error grouping. The class is defined as follows:

```
class MAXException  
{  
public:  
    TSTR message;  
    int error_code;  
    MAXException(TCHAR* msg, int code=0) : message(msg),  
        error_code(code) { }  
};
```

It contains a message buffer and an optional error code. You would signal an error using the **MAXException()** constructor and the C++ throw statement, as in the following example:

```
...  
if (discrim < 0.0) // oh-oh, not good  
throw MAXException ("Unable to find root.", -23);  
...
```

This signals a fatal error with the message and code shown. If the error occurs

during a call to the function by MAXScript code, it will be trapped by MAXScript and the error message will be displayed and the running script will be terminated (but 3ds max will continue running). If the error occurs during a C++-level call, typically the outer 3ds max error catcher will catch and report the error and then exit MAX, or clients of the interface can install their own catch code.

## Property accessors

It is possible to define selected methods in an **FPInterface** as 'property accessor' methods so that dispatch-based clients of the interface may present these methods as properties instead of functions. MAXScript, for example, now presents property accessors defined in this way as simple dot-notation properties of the interface. These accessors remain as ordinary methods in the

C++-level interface for direct C++ clients but are marked as accessors in the descriptor and function maps. A 'property' is typically defined by a pair of accessor methods, one for getting and one for setting, but you can also define read-only methods that have a single getter method. As an example, here's a rework of the above Cylinder interface in which the radius and direction are made properties.

```
class CylInterface : public FPMixinInterface
{
    ...
    virtual void RemoveCaps()=0;
    virtual void AddBend(float offset, float angle, float radius)=0;
    ...
    virtual float GetRadius()=0;
    virtual void SetRadius(float radius)=0;
    virtual Point3 GetDirection()=0;
    virtual void SetDirection(Point3 dir)=0;
    ...
    BEGIN_FUNCTION_MAP
```

```

...
    VFN_0(cyl_removeCaps, RemoveCaps);
    VFN_3(cyl_addBend, AddBend, TYPE_FLOAT,
TYPE_ANGLE,
    TYPE_FLOAT);
...
    PROP_FNS(cyl_getRadius, GetRadius, cyl_setRadius,
SetRadius,
    TYPE_FLOAT);
    PROP_FNS(cyl_getDir, GetDirection, cy_setDir,
SetDirection,
    TYPE_POINT3_BV);
...
END_FUNCTION_MAP
FPInterfaceDesc* GetDesc();
};

```

Each property as a getter and setter virtual method. Their signatures must conform to the convention shown, namely, the getter takes no parameters and the setter one. The getter returns the same type as the setter and the setter type is void. There are also variants that take an explicit time as described below.

Properties have single entries in the function map, using one of the property-related entry macros. In this case, **PROP\_FNS()** takes a getter ID, getter function, setter ID, setter function and finally the property type code. There is also a **RO\_PROP\_FN()** macro for read-only properties that takes a getter ID, getter FN and prop type. Finally, there are variants of **PROP\_FNS** and **RO\_PROP\_FN** that indicate the accessors take an explicit **TimeValue** argument, presumably associated with animatable properties and indicating that the property access should be at the given time. These macros are **PROP\_TFNS** and **RO\_PROP\_TFN** and takes the same arguments as their corresponding base macros. If the above example properties were time-sensitive, the definitions would be as follows:

```

class CylInterface : public FPMixinInterface
{
    ...
    virtual void RemoveCaps()=0;
    virtual void AddBend(float offset, float angle, float radius)=0;
    ...
    virtual float GetRadius(TimeValue t)=0;
    virtual void SetRadius(float radius, TimeValue t)=0;
    virtual Point3 GetDirection(TimeValue t)=0;
    virtual void SetDirection(Point3 dir, TimeValue t)=0;
    ...
    BEGIN_FUNCTION_MAP
        ...
        VFN_0(cyl_removeCaps, RemoveCaps);
        VFN_3(cyl_addBend, AddBend, TYPE_FLOAT,
TYPE_ANGLE,
        TYPE_FLOAT);
        ...
        PROP_TFNS(cyl_getRadius, GetRadius, cyl_setRadius,
SetRadius,
        TYPE_FLOAT);
        PROP_TFNS(cyl_getDir, GetDirection, cy_setDir,
SetDirection,
        TYPE_POINT3_BV);
        ...
    END_FUNCTION_MAP

    FPIInterfaceDesc* GetDesc();
};

```

Recall that there are time-sensitive variants for the **FN\_0**, **FN\_1**, etc., macros,

namely **TFN\_0**, **TFN\_1**, etc. Functions specified by such macros in the function map have an implicit **TimeValue** last parameter. Note that in all the time-sensitive variants, the last **TimeValue** parameter is not defined as an explicit parameter in the **FPInterface** descriptor entry for that function. Such time-sensitive properties and functions are handled specially in MAXScript. It supplies the current MAXScript time, as defined by the current 'at time' context for this implicit parameter, making these functions and properties behave consistently with the rest of MAXScript.

Properties are defined in a special section in the **FPInterface** descriptor constructor following the function definitions, headed by the special tag 'properties'. They are also entered in the function map using new property-specific **FUNCTION\_MAP** macros. Here's an example descriptor fragment:

```
static FPInterfaceDesc cylfpi (  
    CYL_INTERFACE, _T("cylMixin"), 0, &cylinderDesc,  
    FP_MIXIN,  
    ...  
    cyl_removeCaps, _T("removeCaps"), 0, TYPE_VOID, 0, 0,  
    cyl_addBend, _T("addBend"), 0, TYPE_VOID, 0, 3,  
        _T("offset"), 0, TYPE_FLOAT,  
        _T("angle"), 0, TYPE_ANGLE,  
        _T("radius"), 0, TYPE_FLOAT,  
    properties,  
    cyl_getRadius, cyl_setRadius, _T("radius"), IDS_RADIUS,  
        TYPE_FLOAT, f_range, 0.0, 10000.0,  
    cyl_getDirection, cyl_setDirection, _T("direction"), IDS_DIR,  
        TYPE_POINT3_BV,  
    ...  
    end  
);
```

The 'properties' section follows the function definitions. Each property has a

single entry defining the function IDs for the getter and setter functions, a fixed internal property name, a descriptor string resource ID and the property type. If the property is read-only and there is no setter function, specify **FP\_NO\_FUNCTION** for the setter ID. Each property definition can optionally be followed by parameter validation options, as described in the Parameter Validation section above, and these apply to the parameter given to the setter function during property assignment. In this case, the radius is range checked.

If you need to get at it, the property metadata is accessible in the 'props' data member in an **FPInterfaceDesc**. It is a **Tab<>** of pointers to **FPPropDef** class instances, which contain individual property metadata. See the **FPPropDef** class definition in **iFnPub.h** for details.

MAXScript now exposes properties defined in this way as direct dot-notation properties on the interface. So, were before the Get/SetRadius in the example cylinder mixin would have been accessed as functions:

```
r = $cyl01.cylMixin.getRadius()  
$cyl01.cylMixin.setRadius 23
```

you now can use:

```
r = $cyl01.cylMixin.radius  
$cyl01.cylMixin.radius = 23
```

This is true for all the types of FP interfaces that turn up in MAXScript, static, mixin and core. As a further optimization, MAXScript now effectively promotes all interface methods and properties to the level of the interface, so if individual methods and properties have unique names within all the interfaces of an object or class, you can elide the interface name. The above examples could now be written:

```
r = $cyl01.getRadius()  
$cyl01. setRadius 23
```

and:

```
r = $cyl01.radius  
$cyl01. radius = 23
```

If there is a naming conflict, you can always include the interface name level to resolve this.

## **Symbolic Enums**

One or more symbolic enums, similar to C++ enums, can now be added to an FPIInterface's metadata, and individual int parameters and/or results for functions in that interface can be defined as **TYPE\_ENUM** and associated with one of the enum lists. This allows metadata clients to support symbolic encodings for these parameters and results, which MAXScript now does.

Enums are defined in the FPIInterface descriptor following the function and property definitions as sets of string/code pairs. Each enum list is identified by a unique integer, similar to function IDs, which is used to associated a **TYPE\_ENUM** parameter or result with its enum. IDs for these would normally be defined somewhere near the function IDs for an interface. For example:

```
// function IDs  
enum { bmm_getWidth, bmm_getHeight, bmm_getType,  
bmm_copyImage, ...};  
// enum IDs  
enum { bmm_type, bmm_copy_quality, ...};
```

might be some of the IDs for a possible bitmap manager interface. The two

enums provide symbolic codes for the bitmap type and copyImage quality defines in the "bitmap.h" SDK header, such as **BMM\_PALETTED**, **BMM\_TRUE\_32**, **COPY\_IMAGE\_RESIZE\_LO\_QUALITY**, etc. In the descriptor for the interface, any enum lists follow the function and property definitions. They are introduced by the special tag, 'enums', as in the following example:

```
static FPInterfaceDesc bmmfpi (  
    BMM_INTERFACE, _T("bmm"), IDS_BMMI, NULL,  
    FP_CORE,  
    ...  
    bmm_copyImage, _T("copyImage"), ...  
        _T("copyType"), IDS_COPYTYPE, TYPE_ENUM,  
    bmm_copy_quality,  
    ...  
    properties,  
        geo_getType, geo_setType, _T("type"), 0, TYPE_ENUM,  
    bmm_type,  
        enums,  
        bmm_type, 7,  
            "lineArt", BMM_LINE_ART,  
            "paletted", BMM_PALETTED,  
            "gray8", BMM_GRAY_8,  
            "gray16", BMM_GRAY_16 ,  
            "true16", BMM_TRUE_16,  
            "true32", BMM_TRUE_32,  
            "true24", BMM_TRUE_64,  
        bmm_copy_quality, 4,  
            "crop", COPY_IMAGE_CROP,  
            "resizeLo", COPY_IMAGE_RESIZE_LO_QUALITY,  
            "resizeHi", COPY_IMAGE_RESIZE_HI_QUALITY,
```

```
        "useCustom", COPY_IMAGE_USE_CUSTOM,  
    end  
);
```

In the above example, the enums are listed following the function & property definitions. They are introduced by the 'enums' tag and consist of an enum ID followed by a count of items, followed by that many string and code pairs. By attaching them to the interface like this, any number of functions and properties in the interface can use them.

The above example also has function and property definitions showing the use of `TYPE_ENUM`. The `copyImage` function takes a `copyType` parameter which uses the `bmm_copy_quality_enum` and the type property uses the `bmm_type_enum`. In all situations where `TYPE_xxx` types can be supplied in a descriptor, including the new property definitions, `TYPE_ENUM` can be used to indicate an int by-value type. `TYPE_ENUM`'s must always be followed by an enum ID. This is the only case in which the type is specified as a pair of values. `TYPE_ENUM` parameters and results show up in MAXScript as # names. For example, if a bmm interface was in the variable 'bm1' and the bitmap type was `BMM_GRAY_16`,

```
bm1.type  
-> #gray16
```

```
bm1.type = #true32 -- set it to #true24 (code is BMM_TRUE_24)  
bm2 = bm1.copyImage #resizeHi
```

the integer `TYPE_ENUM` codes are translated back-and-forth to symbolic # names by MAXScript using the definitions in the `FPInterface` descriptor's enums. If you need to access the enum metadata in an `FPInterfaceDesc`, it is available in the 'enumerations' data member. This is a `Tab<>` of pointers to `FPEnum` class instances which themselves contain a `Tab<>` of name, code pairs. See class `FPEnum` in `iFnPub.h` for details.

## Additional ParamType2 codes for Function Publishing and MAXScript

**TYPE\_FPVALUE** // FPValue\*, variant value  
**TYPE\_VALUE** // MAXScript Value\*

**TYPE\_FPVALUE** can be used to pass a variant data type, containing one of the types of data that **FPValue** can hold. This allows a function to be defined that can accept many types of values for a **TYPE\_FPVALUE** parameter, the function can determine the type from the 'type' field in the **FPValue** passed. MAXScript supports **TYPE\_PFVALUE** parameters and return values and will convert back-and-forth between the MAXScript types that correspond to the various **FPValue** types. **TYPE\_FPVALUE** also supports **TYPE\_FPVAUE\_BV** and **TYPE\_FPVALUE\_BR** variants. MAXScript arrays will attempt to convert themselves into **TYPE\_XXX\_TAB FPValues** if the array elements are all of the same type, for example all integers generate a **TYPE\_INT\_TAB**, all scene nodes generate a **TYPE\_INODE\_TAB**, etc. If the array contains mixed value types or types not in the following list, a **TYPE\_FPVALUE\_TAB** will be generated, containing an **FPValue\*** variant value for each element in the array. Supported homogeneous types are:

**TYPE\_INT\_TAB**  
**TYPE\_FLOAT\_TAB**  
**TYPE\_TIMEVALUE\_TAB**  
**TYPE\_STRING\_TAB**  
**TYPE\_NAME\_TAB**  
**TYPE\_POINT3\_TAB**  
**TYPE\_POINT2\_TAB**  
**TYPE\_INODE\_TAB**  
**TYPE\_REFTARG\_TAB** (all other ReferenceTarget\*s, modifiers, mtl, ctrls, etc.)

**TYPE\_VALUE** is basically a fallback to be used in situations where MAXScript values of types not covered by **FPValue** types need to be passed

into a published function. The called function is responsible for using the MAXScript SDK to convert to and from the **Value\*** in **TYPE\_VALUE** values.

## Parameter/Result Types

More parameter/result types have been added and the type system has been generalized to allow you to specify pass-by-reference, pass-by-pointer and pass-by-value variations of the appropriate base types. This permits a wider range of interface method signatures in FnPub interfaces, particularly those with '&' reference types typically used for passing back values via parameters. The scripter has been upgraded to support all these new types and pass-by options (see also the MAXScript section here for details).

The type codes, defined in the **ParamType2** enum, are arranged in sections with various suffixes signifying the sections. The main section gives the so-called 'base types', such as **TYPE\_INT**, **TYPE\_FLOAT**, and the other sections are variations derived from the base types. The variations are:

<**base\_type**>\_BV base type passed by value

<**base\_type**>\_BR base type passed by reference, used with & ref params & results

<**base\_type**>\_BP base type passed by pointer, used with \* pointer params& results

<**base\_type**>\_TAB a Tab<> of the base type

<**base\_type**>\_TAB\_BV a Tab<> of base type, the Tab<> is passed by value

<**base\_type**>\_TAB\_BR a Tab<> of base type, the Tab<> is passed by reference

Since some base types are naturally passed by value or pointer, and the base **Tab<>** type is passed by pointer, so not all possible combinations are actually made available. For example, there is no **TYPE\_INT\_BV**, since **TYPE\_INT** is already passed by value, and there is no **TYPE\_STRING\_BP** since **TYPE\_STRING** is already passed by pointer.

## Added base types

**TYPE\_POINT** a Win32 POINT struct

**TYPE\_TSTR** a 3ds max SDK TSTR class

**TYPE\_IOBJECT** a new FnPub system IObject

**TYPE\_INTERFACE** an FPIInterface

**TYPE\_HWND** a Win32 HWND handle

**TYPE\_NAME** a variant of TYPE\_STRING, meant to be interpreted as an interned symbol or

name in the client (eg, MAXScript represents these Name instances,  
as in #foo, etc.)

All the new base types are passed naturally as pointers, so there are **\_BV** and **\_BR** variants for all of them (except for **TYPE\_HWND**). There are also **\_TAB**, **\_TAB\_BR**, **\_TAB\_BV** variants for them all. For reference, all the currently available types are in the ParamType2 enumeration in **/include/iparmb2.h**. Some base types are naturally passed by value, others by pointer. Basically, all the int and float-derived types are passed naturally by-value and all the rest by pointer. The by-value base types are:

ints:

**TYPE\_INT**

**TYPE\_BOOL**

**TYPE\_TIMEVALUE,**

**TYPE\_RADIOBTN\_INDEX**

**TYPE\_INDEX**

floats:

**TYPE\_FLOAT**

**TYPE\_ANGLE**

**TYPE\_PCNT\_FRAC**  
**TYPE\_WORLD**  
**TYPE\_COLOR\_CHANNEL**

The **FPValue** class has a union containing fields for all the base and table types and show the natural passing mode for each. Note that the **\_BP** by-pointer variants of the base types are passed as iptr & fptr pointer fields in the **PValue** union. For the types that are passed naturally by pointer, the **\_BV** variants cause local copies to be made and owned by the **FPValue** carrying them. The **FPValue** destructor will free memory taken by these copies. Also, the **\_BR** variants of naturally-pointer types need to be supplied as pointers to the **FPValue** (or **FPParams**) constructors and are dereferenced at parameter delivery time. So, for example, the following **FPInterface** method:

```
Tab<int> Foo(INode* object, Tab<Point3*>& points);
```

would be declared in the **FUNCTION\_MAP** as

```
FN_2(my_foo, TYPE_INT_TAB_BV, Foo, TYPE_INODE,  
TYPE_POINT3_TAB_BR);
```

and in the descriptor constructor as

```
my_foo, _T("foo"), 0, TYPE_INT_TAB_BV, 0, 2,  
_T("object"), 0, TYPE_INODE,  
_T("points"), 0, TYPE_POINT3_TAB_BR,
```

## **Supported Types**

The type codes you can use to specify function return types or argument types are defined by the **ParamType2** enumeration in `\include\paramtype.h`. Several of the type codes map to the same underlying C++ type, for example there are five float types. In most cases, the alternate codes imply different UI scaling Dimensions that are honored by systems like MAXScript. For example, using

**TYPE\_ANGLE** will cause MAXScript to convert back and forth between radians internally and degrees to the user, so you should use the most specific type code.

As you can see from the list below, all the values are pointer-sized or smaller, so they fit in a single pointer-sized union in an **FPValue**. This means any object larger than a pointer is normally passed by reference. To support passing and returning by-value, a variant set of types is provided that take local copies of the object; they have an **\_BV** suffix in the type code name. This local copy is freed in the destructor of **FPValue**, so **FPValues** of these types effectively have the semantics of pass-by-value. Typically, you would use these by-value types for returning values that have been created locally in the called function, Point3s, strings, etc. All **Tab<>** types take a local copy of the table. All the reftarg types (Mtl\*, INode\*, Texmap\*) are passed only as pointers, no attempt is made to keep local references in the **FPValue** or **FPParam** instances.

#### **Built in data types (enum ParamType)**

**TYPE\_USER**

#### **Built in data types (enum ParamType2)**

**TYPE\_INDEX**

**TYPE\_MATRIX3**

**TYPE\_PBLOCK2**

The following are only for published function parameter types, not pblock2 parameter types.

**TYPE\_ENUM**

**TYPE\_VOID**

**TYPE\_INTERVAL**

**TYPE\_ANGAXIS**

**TYPE\_QUAT**  
**TYPE\_RAY**  
**TYPE\_POINT2**  
**TYPE\_BITARRAY**  
**TYPE\_CLASS**  
**TYPE\_MESH**  
**TYPE\_OBJECT**  
**TYPE\_CONTROL**  
**TYPE\_POINT**  
**TYPE\_TSTR**  
**TYPE\_IOBJECT**  
**TYPE\_INTERFACE**  
**TYPE\_HWND**  
**TYPE\_NAME**  
**TYPE\_COLOR**  
**TYPE\_FPVALUE**  
**TYPE\_VALUE**  
**TYPE\_DWORD**  
**TYPE\_bool**

The following are tables of the above data types (in the same order as base types).

**TYPE\_INDEX\_TAB**  
**TYPE\_MATRIX3\_TAB**  
**TYPE\_PBLOCK2\_TAB**

The following are only for published function parameter types, not pblock2 parameter types.

**TYPE\_ENUM\_TAB, TYPE\_VOID\_TAB,**

**TYPE\_INTERVAL\_TAB, TYPE\_ANGAXIS\_TAB,  
TYPE\_QUAT\_TAB, TYPE\_RAY\_TAB, TYPE\_POINT2\_TAB,  
TYPE\_BITARRAY\_TAB, TYPE\_CLASS\_TAB,  
TYPE\_MESH\_TAB, TYPE\_OBJECT\_TAB,  
TYPE\_CONTROL\_TAB, TYPE\_POINT\_TAB,  
TYPE\_TSTR\_TAB, TYPE\_IOBJECT\_TAB,  
TYPE\_INTERFACE\_TAB, TYPE\_HWND\_TAB,  
TYPE\_NAME\_TAB, TYPE\_COLOR\_TAB,  
TYPE\_FPVALUE\_TAB, TYPE\_VALUE\_TAB,  
TYPE\_DWORD\_TAB, TYPE\_bool\_TAB**

The following pass by-ref types, implies & parameters, int& & float& are passed via .ptr fields, only for FnPub use. These are defined as **TYPE\_xxx + TYPE\_BY\_REF.**

**TYPE\_FLOAT\_BR, TYPE\_INT\_BR, TYPE\_BOOL\_BR,  
TYPE\_ANGLE\_BR, TYPE\_PCNT\_FRAC\_BR,  
TYPE\_WORLD\_BR, TYPE\_COLOR\_CHANNEL\_BR,  
TYPE\_TIMEVALUE\_BR, TYPE\_RADIOBTN\_INDEX\_BR,  
TYPE\_INDEX\_BR, TYPE\_RGBA\_BR, TYPE\_BITMAP\_BR,  
TYPE\_POINT3\_BR, TYPE\_HSV\_BR, TYPE\_REFTARG\_BR,  
TYPE\_MATRIX3\_BR, TYPE\_ENUM\_BR,  
TYPE\_INTERVAL\_BR, TYPE\_ANGAXIS\_BR,  
TYPE\_QUAT\_BR, TYPE\_RAY\_BR, TYPE\_POINT2\_BR,  
TYPE\_BITARRAY\_BR, TYPE\_MESH\_BR, TYPE\_POINT\_BR,  
TYPE\_TSTR\_BR, TYPE\_COLOR\_BR, TYPE\_FPVALUE\_BR,  
TYPE\_DWORD\_BR, TYPE\_bool\_BR**

The following pass by-ref Tab<> types, implies & parameters, int& & float& are passed via .ptr fields, only for FnPub use. These are defined as **TYPE\_xxx + TYPE\_TAB + TYPE\_BY\_REF.**

**TYPE\_FLOAT\_TAB\_BR, TYPE\_INT\_TAB\_BR,  
TYPE\_RGBA\_TAB\_BR, TYPE\_POINT3\_TAB\_BR,**

**TYPE\_BOOL\_TAB\_BR, TYPE\_ANGLE\_TAB\_BR,  
TYPE\_PCNT\_FRAC\_TAB\_BR, TYPE\_WORLD\_TAB\_BR,  
TYPE\_STRING\_TAB\_BR, TYPE\_FILENAME\_TAB\_BR,  
TYPE\_HSV\_TAB\_BR, TYPE\_COLOR\_CHANNEL\_TAB\_BR,  
TYPE\_TIMEVALUE\_TAB\_BR,  
TYPE\_RADIOBTN\_INDEX\_TAB\_BR, TYPE\_MTL\_TAB\_BR,  
TYPE\_TEXMAP\_TAB\_BR, TYPE\_BITMAP\_TAB\_BR,  
TYPE\_INODE\_TAB\_BR, TYPE\_REFTARG\_TAB\_BR,  
TYPE\_INDEX\_TAB\_BR, TYPE\_MATRIX3\_TAB\_BR,  
TYPE\_TSTR\_TAB\_BR, TYPE\_ENUM\_TAB\_BR,  
TYPE\_INTERVAL\_TAB\_BR, TYPE\_ANGAXIS\_TAB\_BR,  
TYPE\_QUAT\_TAB\_BR, TYPE\_RAY\_TAB\_BR,  
TYPE\_POINT2\_TAB\_BR, TYPE\_BITARRAY\_TAB\_BR,  
TYPE\_CLASS\_TAB\_BR, TYPE\_MESH\_TAB\_BR,  
TYPE\_OBJECT\_TAB\_BR, TYPE\_CONTROL\_TAB\_BR,  
TYPE\_POINT\_TAB\_BR, TYPE\_IOBJECT\_TAB\_BR,  
TYPE\_INTERFACE\_TAB\_BR, TYPE\_HWND\_TAB\_BR,  
TYPE\_NAME\_TAB\_BR, TYPE\_COLOR\_TAB\_BR,  
TYPE\_FPVALUE\_TAB\_BR, TYPE\_VALUE\_TAB\_BR,  
TYPE\_DWORD\_TAB\_BR, TYPE\_bool\_TAB\_BR**

The following pass by-value types, implies dereferencing the (meaningful) pointer-based values, only for FnPub use. These are defined as **TYPE\_XXX + TYPE\_BY\_VAL**.

**TYPE\_RGBA\_BV, TYPE\_POINT3\_BV, TYPE\_HSV\_BV,  
TYPE\_INTERVAL\_BV, TYPE\_BITMAP\_BV,  
TYPE\_MATRIX3\_BV, TYPE\_ANGAXIS\_BV, TYPE\_QUAT\_BV,  
TYPE\_RAY\_BV, TYPE\_POINT2\_BV, TYPE\_BITARRAY\_BV,  
TYPE\_MESH\_BV, TYPE\_POINT\_BV, TYPE\_TSTR\_BV,  
TYPE\_COLOR\_BV, TYPE\_FPVALUE\_BV, TYPE\_CLASS\_BV**

The following pass by-value Tab<> types, implies dereferencing the (meaningful) pointer-based values, only for FnPub use. These are defined as

**TYPE\_xxx + TYPE\_TAB + TYPE\_BY+VAL.**

**TYPE\_FLOAT\_TAB\_BV, TYPE\_INT\_TAB\_BV,  
TYPE\_RGBA\_TAB\_BV, TYPE\_POINT3\_TAB\_BV,  
TYPE\_BOOL\_TAB\_BV, TYPE\_ANGLE\_TAB\_BV,  
TYPE\_PCNT\_FRAC\_TAB\_BV, TYPE\_WORLD\_TAB\_BV,  
TYPE\_STRING\_TAB\_BV, TYPE\_FILENAME\_TAB\_BV,  
TYPE\_HSV\_TAB\_BV, TYPE\_COLOR\_CHANNEL\_TAB\_BV,  
TYPE\_TIMEVALUE\_TAB\_BV,  
TYPE\_RADIOBTN\_INDEX\_TAB\_BV, TYPE\_MTL\_TAB\_BV,  
TYPE\_TEXMAP\_TAB\_BV, TYPE\_BITMAP\_TAB\_BV,  
TYPE\_INODE\_TAB\_BV, TYPE\_REFTARG\_TAB\_BV,  
TYPE\_INDEX\_TAB\_BV, TYPE\_MATRIX3\_TAB\_BV,  
TYPE\_PBLOCK2\_TAB\_BV, TYPE\_VOID\_TAB\_BV,  
TYPE\_TSTR\_TAB\_BV, TYPE\_ENUM\_TAB\_BV,  
TYPE\_INTERVAL\_TAB\_BV, TYPE\_ANGAXIS\_TAB\_BV,  
TYPE\_QUAT\_TAB\_BV, TYPE\_RAY\_TAB\_BV,  
TYPE\_POINT2\_TAB\_BV, TYPE\_BITARRAY\_TAB\_BV,  
TYPE\_CLASS\_TAB\_BV, TYPE\_MESH\_TAB\_BV,  
TYPE\_OBJECT\_TAB\_BV, TYPE\_CONTROL\_TAB\_BV,  
TYPE\_POINT\_TAB\_BV, TYPE\_IOBJECT\_TAB\_BV,  
TYPE\_INTERFACE\_TAB\_BV, TYPE\_HWND\_TAB\_BV,  
TYPE\_NAME\_TAB\_BV, TYPE\_COLOR\_TAB\_BV,  
TYPE\_FPVALUE\_TAB\_BV, TYPE\_VALUE\_TAB\_BV,  
TYPE\_DWORD\_TAB\_BV, TYPE\_bool\_TAB\_BV**

The following pass by-pointer types for int & float types, implies \* parameters, int\* & float\* are passed via .ptr fields, only for FnPub use. These are defined as **TYPE\_xxx + TYPE\_BY\_PTR.**

**TYPE\_FLOAT\_BP, TYPE\_INT\_BP, TYPE\_BOOL\_BP,  
TYPE\_ANGLE\_BP, TYPE\_PCNT\_FRAC\_BP,  
TYPE\_WORLD\_BP, TYPE\_COLOR\_CHANNEL\_BP,  
TYPE\_TIMEVALUE\_BP, TYPE\_RADIOBTN\_INDEX\_BP,**

**TYPE\_INDEX\_BP, TYPE\_ENUM\_BP, TYPE\_DWORD\_BP,  
TYPE\_bool\_BP**

There are no specific by-pointer Tab<> types, all Tab<> types are by-pointer by default.

**TYPE\_MAX\_TYPE**

### **Published Functions and MAXScript**

MAXScript automatically provides access to all functions published by a plugin via the FnPub system. The current scheme is experimental and may change somewhat. Each plugin class appears in MAXScript as a 3ds max class object, that can be used to construct instances of the plugin, do class tests, etc. If a plugin publishes interfaces, they are visible in MAXScript as properties on this class object. The internal name for the interface is used as the property name. All the functions in the interface are accessible as named properties on the interface. So, if the above example interfaces were published by EditMesh, the following script frags would work:

**EditMesh.faceOps.extrude \$foo.mesh #{1,2,3} 10**

calls the Extrude function in the FaceOps interface on \$foo's mesh, faces 1, 2 and 3, amount 10 units.

**EditMesh.actions**

retrieves and displays the action functions. Each interface is stored as a struct definition in the class object.

**EditMesh.actions.create ()**

starts (or stops) the create mode. This would (should!) have the side-effect of highlighting/unhighlighting the Create button in the EditMesh rollups. Calls to Action functions in MAXScript return **true** if the function returns **FPS\_OK** and **false** otherwise.

**if EditMesh.actions.create.isChecked() then ...**

The predicate functions for an Action Function are available as properties on the action function object itself, as shown. You can determine if a predicate is supplied by asking:

**if EditMesh.actions.create.isChecked != undefined**

## **Extending Animatable::GetInterface()**

The following have been added to class Animatable:

**virtual FPInterface\* GetInterface(Interface\_ID id);**

Any future object-based interfaces should be allocated unique Interface\_IDs (you can use Gencid.exe for this) and made available through this call.

The default implementation of GetInterface(Interface\_ID) looks up a standalone interface of the given ID on the object's ClassDesc. This gives access to standalone FnPub interfaces via any of a plugin's objects, without having to dig around for the ClassDesc, so you should fall back to calling the default implementation if you don't recognize an ID in your implementation of GetInterface(Interface\_ID).

## **Global Functions related to Function Publishing**

### **Prototype:**

**void RegisterCOREInterface(FPInterface\* fpi);**

### **Remarks:**

This function is available in release 4.0 and later only.

This function registers an interface object. Creating Core interfaces is done in the normal FnPub manner, via public virtual interface headers and implementation interfaces through the FPInterface constructor as above, but by specifying NULL for the ClassDesc pointer. The core interfaces published this way are automatically available in the scripter, with each interface visible as a struct function package of the same name as the internal name of the interface.

### **Parameters:**

**FPInterface\* fpi**

The pointer to the function publishing interface class.

### **Prototype:**

**FPInterface\* GetCOREInterface(Interface\_ID id);**

### **Remarks:**

This function is available in release 4.0 and later only.

This function locates a Core interface object by its unique interface ID.

**Parameters:**

**Interface\_ID id**

The unique interface ID of an interface object.

**Return Value:**

A pointer to the Core interface object associated with the specified interface ID.

**Prototype:**

**int NumCOREInterfaces();**

**Remarks:**

This function is available in release 4.0 and later only.

This function returns the number of available Core interfaces.

**Prototype:**

**FPInterface\* GetCOREInterface(int i);**

**Remarks:**

This function is available in release 4.0 and later only.

This function returns a Core interface object by its index.

**Parameters:**

**int i**

The index of the Core interface object you wish to retrieve.

**Return Value:**

A pointer to the I-th Core interface object.

**Prototype:**

**FPInterface\* GetCOREInterfaceAt(int i);**

**Remarks:**

This function is available in release 4.0 and later only.

This function returns a Core interface object by its index.

**Parameters:**

**int i**

The index of the Core interface object you wish to retrieve.

**Return Value:**

A pointer to the I-th Core interface object.

**Prototype:**

```
FPInterface *GetInterface(SClass_ID super, Class_ID cls,  
Interface_ID id);
```

**Remarks:**

This function is available in release 4.0 and later only.

This function is a global helper function that finds the ID'd interface for the given plugin class and superclass ID's and saves client code from having to dig through the ClassDir to find ClassDesc's.

**Parameters:**

**SClass\_ID super**

The superclass ID of the plugin.

**Class\_ID cls**

The class ID of the plugin.

**Interface\_ID id**

The unique ID of the interface object to retrieve.

**Return Value:**

A pointer to the Core interface object associated with the specified class and superclass ID's.

## Geometry Pipeline System

See Also: [Class ModContext](#), [Class ObjectState](#), [Class LocalModData](#), [Class IDerivedObject](#), [Class Object](#), [Class INode](#), [List of Channel Bits](#), [Class GeomPipelineEnumProc](#).

## **Sub-Topics**

The following hyperlinks jump to the beginning of sub-topics within this section:

[Overview](#)

[Pipeline Details](#)

[Instanced Modifiers](#)

[Space Warps in the Pipeline](#)

[ObjectState Details](#)

[Developer Access to the Pipeline](#)

[Channel Details](#)

[Data Flow in the Pipeline](#)

[The Pipeline and the INode TM Methods](#)

[A Note about Caching](#)

[Modifier Stack Branching](#)

[Objects Flowing Through the Pipeline](#)

## Overview

An understanding of the geometric pipeline system is important for developers creating plug-ins that deal with geometry in the scene. This section describes the pipeline system. A pipeline is the system used by 3ds max that allows a node in the scene to be altered, perhaps repeatedly, through the application of modifiers. At the beginning of a pipeline is the **Base Object**. This is a procedural object or just a simple mesh. At the end of a pipeline is the world space state of the object. This world space state is what appears in the 3D viewports and is rendered.

For the system to evaluate the state of the object at the end of the pipeline, it must apply each modification along the way, from beginning to end. As an example, say a user creates a procedural cylinder in the scene, applies a Bend modifier to it, and then applies a Taper modifier to it. As the system evaluates this pipeline, it starts with the state of the cylinder object. As this object state moves along the pipeline, it encounters the Bend modifier. The system asks the Bend modifier to apply its deformation to the object state. The result of this operation is passed as the source into the Taper modifier. The Taper then applies its deformation. The result of this operation is passed to the system which translates the result into world space, and the state of the node in the scene is complete.

To maximize the speed that the system can evaluate the state of a node, the system maintains a **World Space Cache** for each node in the scene. This world space cache is the result of the node's pipeline. It reflects the state of the object in world space after everything has been applied to it. Along with the cache, the system maintains a **Validity Interval**. The validity interval indicates the period of time over which the cache accurately reflects the state of the node.

Whenever a node needs to perform an operation, such as display itself at a certain time, the system checks the validity interval at that time to see if the cache is valid. If it is, the operation is performed using the cached representation. If it is not, the pipeline is evaluated and the cache is made valid at that time. The validity interval is also updated to reflect the new cache. The operation is then performed.

As an additional mechanism to speed up processing, a pipeline is broken up into **Channels**. Channels allow modifiers to only alter certain portions of the object. For example, there is a separate channel for geometry (i.e. the vertices of the object). If a modifier only affects the vertices and nothing else, the system has considerably less work to do than if it had to reevaluate the entire state of the

object including its face structure, UV coordinates, material assignments, etc. For example, the Bend modifier only affects the geometry channel; all the other channels do not require evaluation.

There are separate channels for geometry (vertices), topology (face or polygon structures), texture vertices (UV coordinates), sub-object selection, level of selection, and display control. These separate channels allow the cache system of 3ds max to be more sophisticated. Instead of just caching one global state for the object, it can cache separate portions of it based on the channels.

## Pipeline Details

This section discusses the details of the 3ds max pipeline architecture. The concepts of Base Objects, Derived Objects, and World Space Derived Objects are presented. The ModContext, ModApp, and ObjectState are also explained. The diagram below (Figure 1) shows the simple pipeline of a Cylinder in the scene. Each node in the scene has a reference to an object. In this case the node's object reference points directly to the **Base Object**. It is the flow of these object references that represent the pipeline. In this case it's the flow from the node's object reference to the procedural cylinder object that stores the creation parameters.

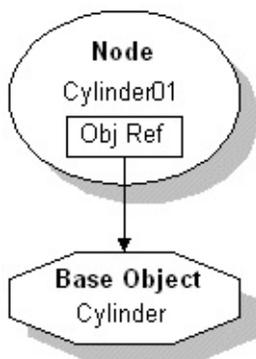


Figure 1.

In 3ds max objects can be instanced. This means that more than one node can point to the same object reference. Figure 2 below shows two pipelines. In this case there is a Cylinder and an instanced copy of it. Note how the object reference of the instanced copy points to the same Base Object as the original. If the creation parameters of the Cylinder are changed, both nodes will change in the scene since they both point at the same Base Object which stores the creation parameters.

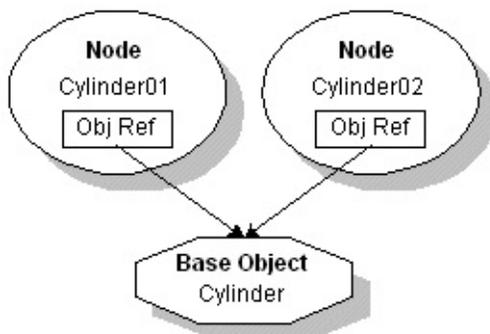


Figure 2.

3ds max supports the application of one or more modifiers to alter the pipeline in

some way. Figure 3 below shows the pipeline resulting from applying a Bend object space modifier to a Cylinder. When a modifier is first applied, a new **Derived Object** is inserted into the node's pipeline.

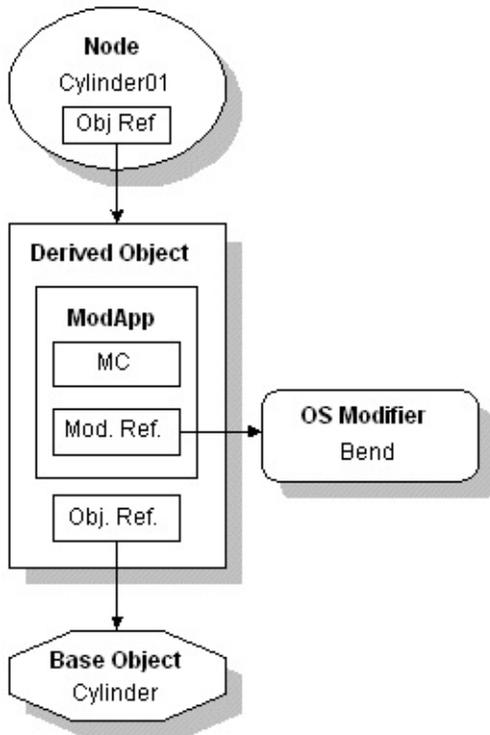


Figure 3.

This arrangement would appear in MAX's modifier stack as follows:

**Bend**

-----

**Cylinder**

This indicates that the Base Object of the pipeline is the Cylinder. The ----- indicates the start of a Derived Object. The Bend is the modifier referenced by this Derived Object.

A Derived Object consists of one or more applications of modifiers followed by a reference to another object. This other object may be a Base Object or another Derived Object. In Figure 3 above, the Derived Object has a single application of a modifier. This application of a modifier is referred to as a **ModApp**. The ModApp primarily consists of a reference to a modifier -- in this case the Bend. Note that the modifier does not sit within the pipeline, but is rather referenced by the ModApp within the pipeline.

In addition to the modifier reference, the ModApp contains an instance of the

class **ModContext**. In Figure 3 and those that follow, the ModContext is represented by a box labeled 'MC'. The ModContext stores information about the space the modifier was applied in, and allows a modifier to store data that it needs for its operation.

Specifically, the ModContext stores three items shown in Figure 4 and described below:

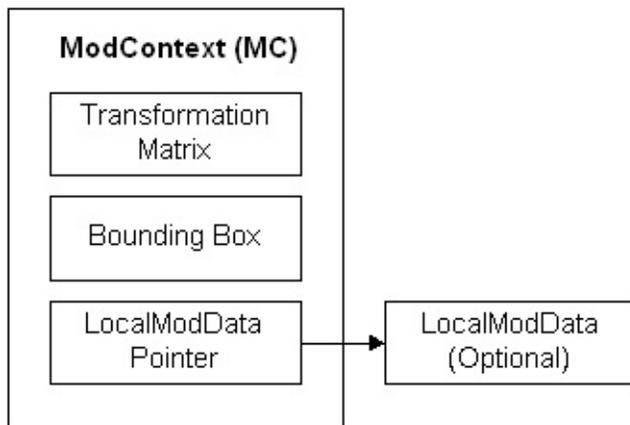


Figure 4.

- The Transformation Matrix. This matrix represents the space the modifier was applied in. The modifier plug-in uses this matrix when it deforms an object. The plug-in modifier first transforms the points with this matrix. Next it applies its own deformation. Then it transforms the points back through the inverse of this transformation matrix.
- The Bounding Box of the Deformation. This represents the scale of the modifier. For a single object it is the bounding box of the object. If the modifier is being applied to a sub-object selection it represents the bounding box of the sub-object selection. If the modifier is being applied to a selection set of objects (and the user interface 'Use Pivot Points' checkbox is off), then this is the bounding box of the entire selection set. For a selection set of objects the bounding box is constant. In the case of a single object, the bounding box is not constant. For example, if the user applies a 90 degree bend to a cylinder, then changes the height of the cylinder, one would want the cylinder to still be bent 90 degrees. If the bounding box did not adapt, the cylinder would appear to move through the bend causing the bend angle to be incorrect.
- A pointer to an instance of a class derived from LocalModData. This is the part of the ModContext that the plug-in developer controls. It is the place where a modifier may store application-specific data. The LocalModData

class has two methods the derived class must implement. One is a Clone procedure so the system can copy a ModContext. The second is a virtual destructor so the derived class can be properly deleted.

More than one modifier may be applied to an object. Figure 5 below shows the previous bent cylinder with an additional Taper modifier applied.

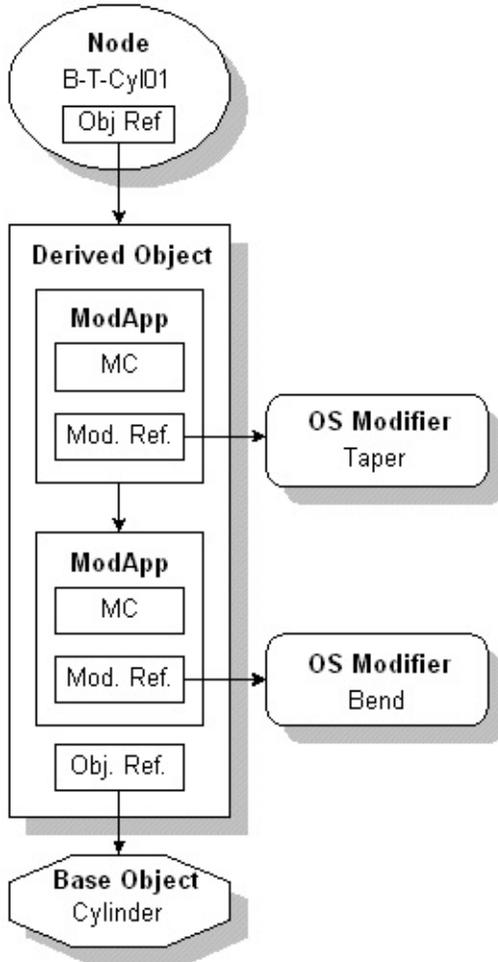


Figure 5.

In this case a new ModApp is inserted into the existing Derived Object. The modifier reference of the new ModApp points to the Taper modifier.

3ds max also allows Reference Copies of items in the scene. Note: the use of the term 'Reference' here is from the 3ds max user interface definition of reference -- not the C++ reference or the 3ds max dependency reference meanings.

When a Reference Copy is made, a new Derived Object is inserted into the item's pipeline. Modifiers applied to the Reference Copy will have their ModApps inserted into the new Derived Object. You can see this graphically illustrated in Figure 6. This diagram shows the result of a bent cylinder being

Reference Copied and having a Taper modifier applied. Note that the object reference of the new Derived Object points to the original Derived Object.

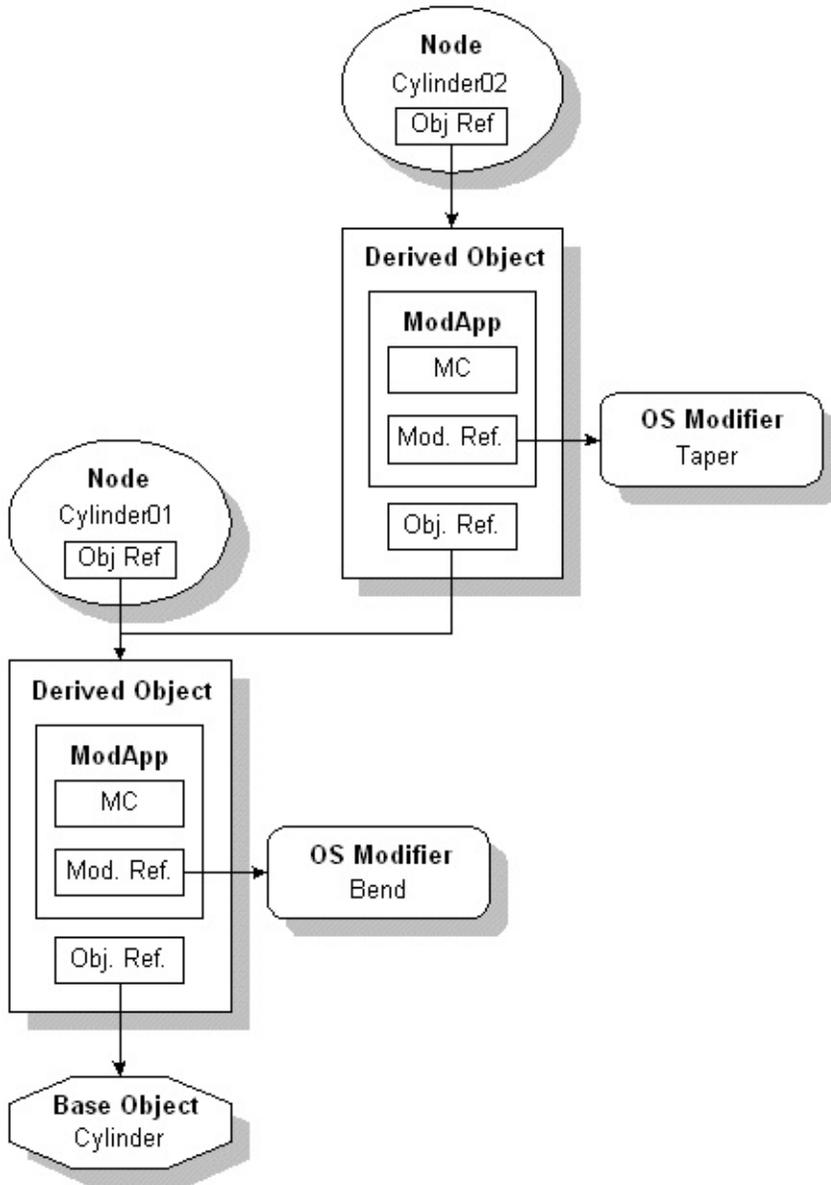


Figure 6.

The modifier stack for Cylinder02 in Figure 6 would appear as:

**Taper**

-----

**Bend**

-----

**Cylinder**

Note the two occurrences of the ----- indicating that two Derived Objects are in use. The original one points to the Bend while the new one points to the Taper.

## Instanced Modifiers

An instance of the **ModApp** class exist in addition to the modifier because the plug-in modifier itself may be instanced (used by several objects). When the user applies a modifier to more than one object, one new instance of the modifier's class is created and shared amongst the objects. Each object that the modifier is applied to gets a new ModApp inserted into its pipeline. These ModApps then reference the same modifier.

In the screen image and diagram below (Figures 7 and 8), the user has selected two independent Cylinders in the scene, checked the 'Use Pivot Points' box, and applied a Bend modifier.

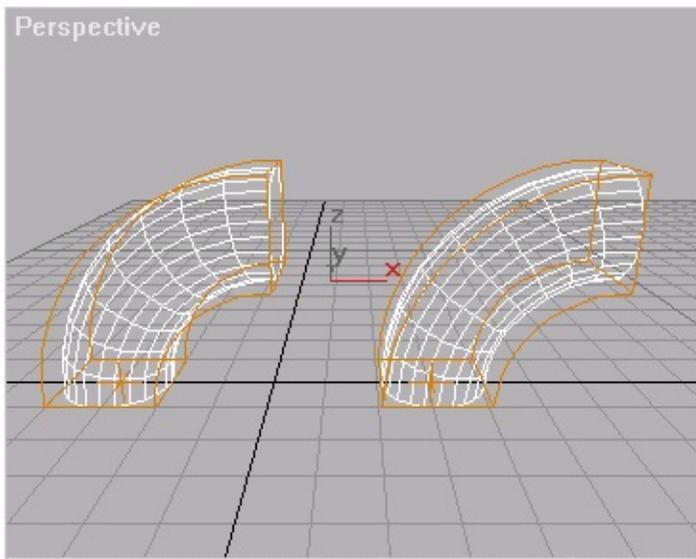


Figure 7.

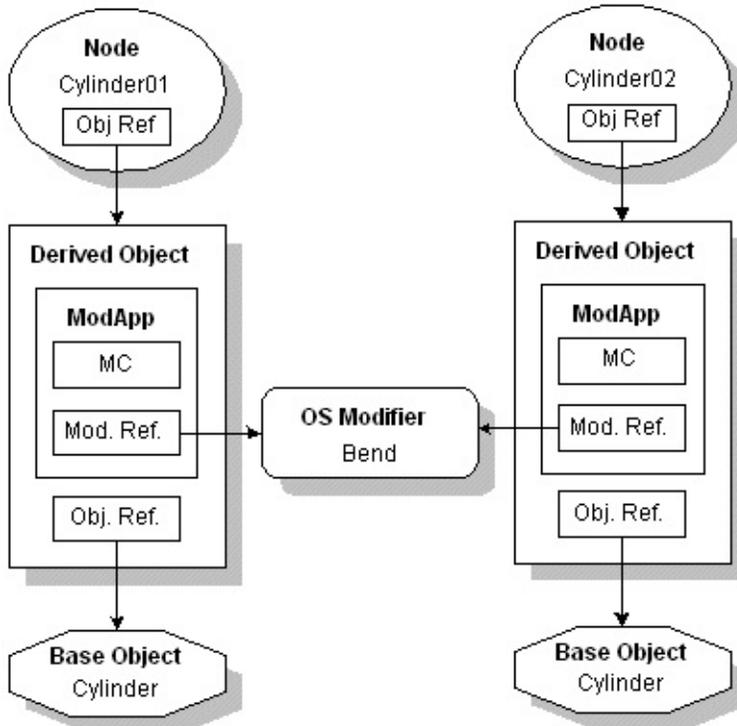


Figure 8.

Note that each cylinder has its own Derived Object. The modifier reference of each ModApp points to the same instanced modifier however. The ModApp also stores the ModContext (labeled 'MC' in the diagrams). One data member of this ModContext is the bounding box of the deformation. Because the 'Use Pivot Points' button was checked at the time the bend was applied, each bounding box stored in the ModContext is the size of each cylinder alone. Additionally, the transformation matrix of each ModContext reflects that the bend is to be applied to each cylinder in its own space. The result is the bend is applied locally and independently to each cylinder.

Contrast this with the following case. In the screen capture of Figure 9 the user applied the Bend modifier to the cylinders as follows: First the independent cylinders were selected. Then the 'Use Pivot Points' button was un-checked. Then the Bend modifier was applied. This means the bend will be applied to the entire selection set as a whole. Note that the cylinders are bent about a common center.

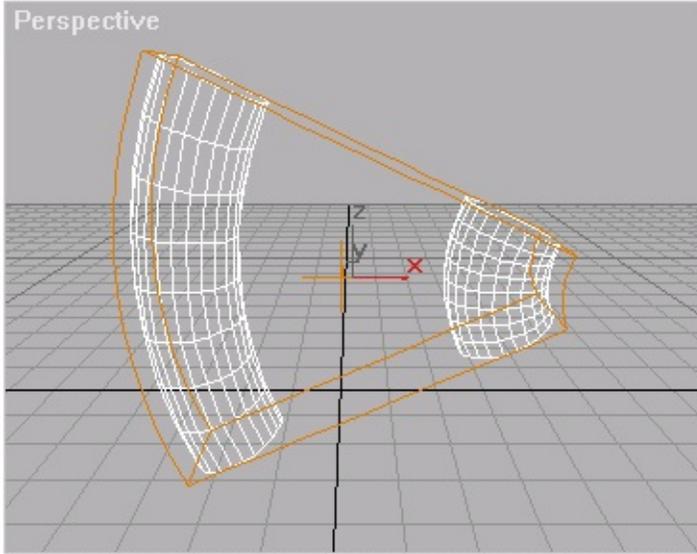


Figure 9.

In this case the bend is applied relative to the bounding box of both cylinders. Thus the bounding box of the ModContexts are the same for each cylinder. Note what happens if one cylinder is moved away from the other in the scene. The Modifier is selected so the gizmo shows the bounding box graphically. The bounding box remains the size of both even when the objects no longer share the same world space relationship.

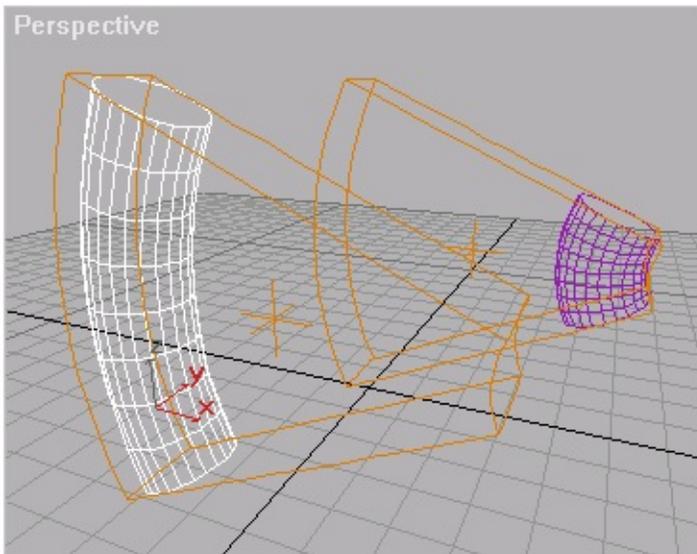


Figure 10.

The separation of the ModApps from the Modifier allow this flexibility. The bounding box stored with the ModContext of the ModApp represents the scale of the application of the modifier. The transformation matrix of the ModContext represents the space the modifier was applied in. The instanced modifier just

uses this information to properly modify each input object.

## Space Warps (World Space Modifiers) in the Pipeline

This section discusses the pipeline of items with World Space Modifiers applied. Below is a diagram of a Cylinder and a Ripple Space Warp before the Cylinder has been bound to the space warp.

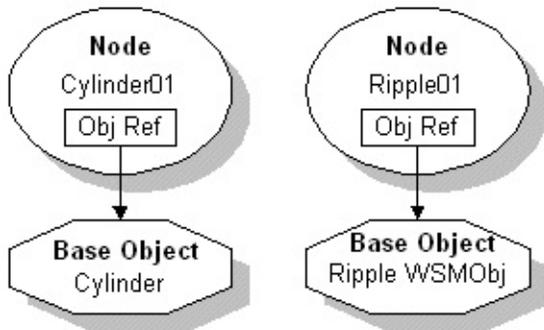


Figure 11.

The following diagram shows the pipeline of the Cylinder after it has been bound to the Ripple Space Warp.

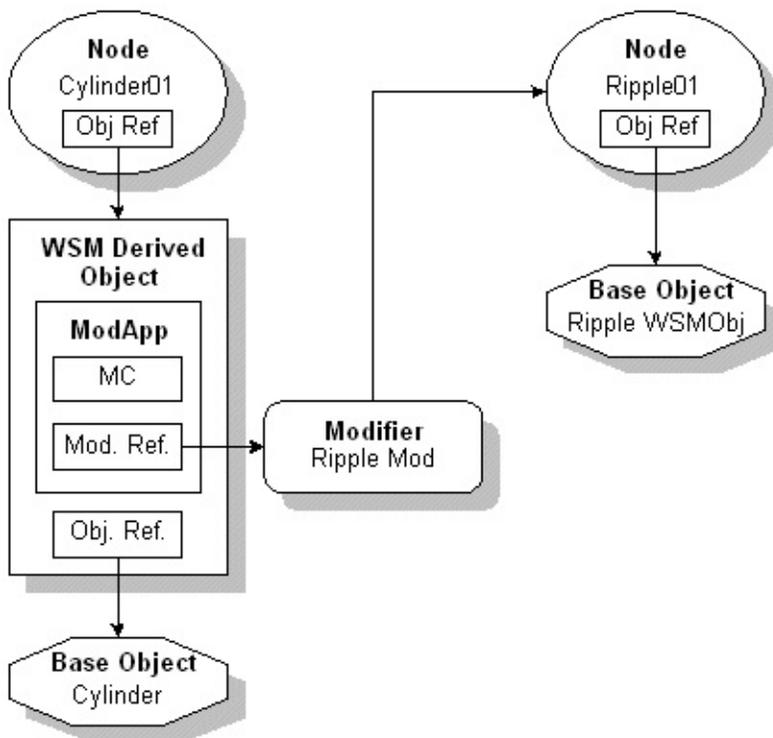


Figure 12.

The modifier stack for this condition would appear as follows:

### **Ripple Binding**

=====

## Cylinder

The Cylinder is the Base Object. The ===== represents the beginning of a WSM Derived Object. The Ripple Binding is the actual application of the world space modifier.

When the Cylinder is bound to the Ripple a new WSM Derived Object is inserted into the Cylinder's pipeline. The WSM Derived Object is similar to the Derived Objects that hold object space modifier ModApps except that these are contained in a specific node (WSM Derived Object is in fact the exact same class as Derived Object except for the **ClassID()**). Since they are associated with a specific node, they cannot be instanced.

The modifier reference of the ModApp points to a newly-created WSM Modifier. This WSM Modifier usually has a references to the node in the scene. In this case it is to the Ripple01 node. This reference is used to retrieve the position of the node (from the node's world space transformation matrix). It uses this matrix to transform the points of the object it is deforming into the space of the WSM object where it actually performs the deformation. For additional information on Space Warp plug-ins see the Advanced Topics section [Space Warp Plug-Ins](#).

## ObjectState Details

The object state is the structure that flows up the pipeline. When the **Object::Eval()** method is called on an object or a derived object, it returns an **ObjectState**. This is passed from one object reference to the next. The **ObjectState** contains these elements:

- A pointer to the object in the pipeline. A modifier will often refer to the object in the pipeline using this pointer. The object pointer is a public data member and is defined as: **Object \*obj;**
- A matrix. If an object cannot convert itself to a deformable type, 3ds max deforms a matrix instead. After the matrix is deformed, it is converted back into an [= 4\) BSPSPopupOnMouseOver\(event\);">orthonormal](#) matrix using an iterative process that 'averages' the axis. All objects are supposed to be able to convert themselves to TriObjects (which are deformable) so in general this is not used. However in the case of cameras and lights it doesn't make sense to convert them to TriObjects so this is how they are deformed. You can see an example of this by binding a camera to a space warp like ripple. The camera will bounce up and down as it is 'deformed' by the space warp.
- Flags for channels that are Boolean type. Developers do not need to be concerned with these flags.
- A material index. This is no longer used.

See [Class ObjectState](#) for details on the methods dealing with the **ObjectState**.

## Developer Access to the Pipeline

This section discusses how a developer can access and work with the results of the pipeline.

There is an API available to retrieve the result of a node's pipeline. This is **INode::EvalWorldState()**.

**Prototype:**

```
virtual const ObjectState& EvalWorldState(TimeValue  
time,BOOL evalHidden=TRUE)=0;
```

This returns the result of the node's pipeline just as it appears in the scene. This may not return an object that a developer has a reference to -- it may just be an object that has flowed down the pipeline. For example, if there is a Cylinder in the scene that has a Bend and Taper applied, **EvalWorldState()** would return an ObjectState containing a **TriObject**. This is the result of the cylinder turning into a **TriObject** and being bent and tapered. See [Class INode](#).

If a developer needs to access the object that the node in the scene references, then the method **INode::GetObjectRef()** should be used instead. See [INode - Object Reference Methods](#).

Class **IDerivedObject** also allows a developer to create derived objects and add and delete modifiers. To access the pipeline of a node in the scene first retrieve the object reference using **INode::GetObjectRef()**. Given this **Object** pointer check its SuperClassID to see if it is **GEN\_DERIVOB\_CLASS\_ID**. If it is, you can cast it to an **IDerivedObject**. See [Class IDerivedObject](#) for more details.

## Channel Details

Channels allow modifiers to only alter certain portions of the object. The pipeline is divided into the following channels:

### [List of Channel Bits](#)

Modifiers have the option of only modifying specific channels. The main purpose of this is to allow MAX's caching system to be more sophisticated. Individual caches can be constructed for different channels at different points along the pipeline. So for example, if the texture coordinate portion of the pipeline changes, and the geometric portion of the pipeline is cached, the geometry portion won't need to be reevaluated. This means that modifiers that only depend on geometry may not need to be reevaluated.

It is up to the object to define the meaning of the channels. Take the **TOPO\_CHANNEL** for example. For a **TriObject** the topology is the face structure, the materials, and the smoothing information. For a **SplineShape** the topology is undefined. This is because a **SplineShape** is essentially just an array of points and has no topology.

## Data Flow in the Pipeline - An Example

This section presents a detailed look at the flow of the pipeline from the Base Object to the resulting World Space Cache. This includes the derived objects of the object space portion of the pipeline, the application of the node's transform controller, and through the world space portion of the pipeline.

Figure 13 below is a diagram showing a procedural Cylinder with a Bend modifier applied, and bound to a Ripple space warp.

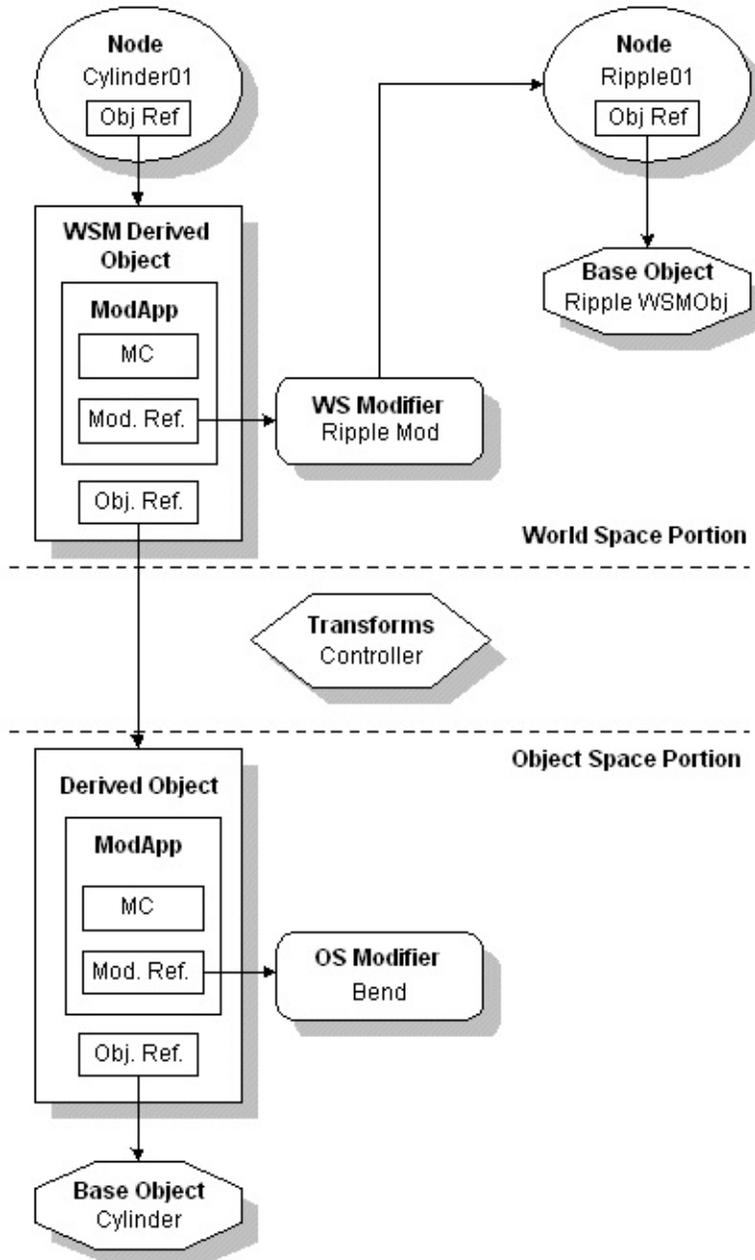


Figure 13.

The data flow in the pipeline follows the Object References. In the example above, the Cylinder01 Node has an Object Reference pointing to the WSM Derived Object. Its Object Reference points to the Derived Object. Its Object Reference points to the Cylinder Base Object. This is the path the data follows as it moves along the pipeline.

The actual object that flows between these references is an ObjectState. This ObjectState is the result of the **Object::Eval()** method being called on the Object Reference. Below is a description of the flow of this ObjectState along the pipeline starting at the procedural cylinder base object.

This pipeline starts at the Base Object -- the procedural Cylinder. The system asks the cylinder to evaluate itself. In the cylinder's implementation of **Object::Eval()**, it simply returns itself ( **return ObjectState(this);** ). It returns this ObjectState to the next Object Reference in the pipeline. This is the Object Reference of the Derived Object.

The Derived Object sends this ObjectState through each of its ModApps. In this example, the only ModApp is for the Bend Modifier. The Bend requires Deformable objects (it indicates this in its implementation of **Modifier::InputType()**). The ModApp handles converting the object to the appropriate type for the Modifier. The cylinder is asked to convert itself to a Deformable object. It does this in its implementation of **Object::ConvertToType()** by creating a new TriObject and setting the TriObject's mesh pointer to point at the cylinder's triangle mesh.

This Deformable TriObject is then sent through the Bend Modifier. The Bend is passed the ModApp's ModContext as an argument to its **Modifier::ModifyObject()** method. The Bend first modifies the points of the TriObject using the transformation matrix of the ModContext. This puts the object into the space the modifier was applied in. Next the Bend applies its deformation to the points (it bends them). Then the Bend modifies the points of the TriObject by the inverse of the ModContext transformation matrix. This restores the object, excepting it now has the bend effect applied.

At this point, the Derived Object returns the result to the next Object Reference in the pipeline. In this case the result is a TriObject that has been bent. It is returned to the Object Reference of a World Space Derived Object.

This juncture between the Derived Object and the World Space Modifier Derived Object is where the pipeline crosses from object space to world space. At this point the result of the node's transform controller, and the object offset

transformation are put into the pipeline. For more details on these various TMs see the Advanced Topics section on [Node and Object Offset Transformations](#).

To understand how the node's transform controller and the object offset transformation are put into the ObjectState we need to take a look at the ObjectState TM. There is a TM that is part of the ObjectState that flows up the pipeline. This TM starts as the identity matrix. The first thing that happens is this TM goes through the object space pipeline. At this time, any modifiers that need to be applied to it are applied. In most cases no modifiers are applied. However, if the object flowing through the pipeline is Deformable but has no points to deform (such as a camera), then modifiers acting on Deformables may be applied to it. This is because cameras are deformable, but don't have 'points' to deform (i.e. there is no 'mesh' associated with a camera). Since there are no points, this matrix is deformed instead.

At the end of the object space portion of the pipeline, out comes the ObjectState and its TM. Usually this TM is the identity, but at times (like with the camera) it is not.

Now the node's transform controller TM must be taken into account. The transform controller's TM is the result of taking the node's parent's TM, and passing it into the transform controller's **Control::GetValue()** method. The controller applies its relative effect to this matrix, and the result is the Node TM. To the Node TM, the object offset transformation is applied. This result is then multiplied by the ObjectState TM that resulted from the object space portion of the pipeline. This resulting matrix is then stored back in the ObjectState TM. Note that this TM is not applied to the object yet -- it is only carried by the ObjectState.

At this point we have the node's transform controller and the object offset transformation stored in the ObjectState TM.

Now comes the world space portion of the pipeline and the world space modifiers -- we've reached the first ModApp of the WSM Derived Object. The first World Space Modifier transforms the object's points by this ObjectState TM. It does this because it does its deformation in world space, and applying the ObjectState TM puts the object into world space. This happens automatically for deformables. If the TM does not get applied to the object (for example for a camera) the TM will continue to get deformed by the world space modifiers. As soon as this TM is applied to the object points, the TM gets set to the identity matrix. At this time, the points of the object have been transformed into world

space.

Next the ObjectState is passed to the Ripple Modifier for it to apply its effect. The Ripple modifier has a reference to the Ripple Object Node in the scene. It uses this reference to get the Object TM of the Ripple Node. It needs this because this is the space it is going to apply the ripple effect in. The Ripple Modifier also uses its reference to the Ripple Node to get the parameters of the Ripple WSM Object.

The Ripple Modifier uses this data, and applies its ripple affect. The result at this point is a TriObject with the Bend and Ripple applied. This ObjectState is returned to the next Object Reference in the pipeline. This is the Object Reference of the node in the scene.

The node maintains an ObjectState that is effectively the world space cache. This world space cache is the storage for the result of the pipeline. Associated with this cache is a validity interval. When the system needs the result of a node's pipeline, it checks to see if the validity interval of the cache is valid. If it is, the cached representation is used. If it is not, the pipeline is evaluated and the cache is made valid. The validity interval is updated, and the cached ObjectState is returned.

## The Pipeline and the INode TM Methods

This section discusses the INode methods **GetObjectTM()**, **GetObjTMBeforeWSM()** and **GetObjTMAfterWSM()** and their relationship to the pipeline.

The **INode::GetObjectTM()** method returns a matrix that is used to transform the points of the object from object space to world space. Let's look at an example of how this method is used. Consider how the cylinder node is drawn in the scene. The cylinder draws itself in its **BaseObject::Display()** method. Into this method is passed an INode pointer. What the implementation of **Display()** does is call **INode::GetObjectTM()**. This method returns the matrix that is used to transform the points of the object from object space to world space. The **Display()** method then takes the matrix returned from **GetObjectTM()** and sets it into the graphics window (using **GraphicsWindow::setTransform()**). In this way, when the object starts drawing points in object space, they will be transformed with this matrix. This puts them into world space as they are drawn.

Below is the code from the **SimpleObject** implementation of **BaseObject::Display()**. This is the code that the cylinder uses to draw itself. Note the **GetObjectTM(t)** and **setTransform(mat)** calls.

```
int SimpleObject::Display(TimeValue t, INode* inode,
    ViewExp *vpt, int flags)
{
    if (!OKtoDisplay(t)) return 0;
    GraphicsWindow *gw = vpt->getGW();
    Matrix3 mat = inode->GetObjectTM(t);
    UpdateMesh(t); // UpdateMesh just calls BuildMesh() if req'd at
    time t.
    gw->setTransform(mat);
    mesh.render(gw, inode->Mtls(),
        (flags&USE_DAMAGE_RECT) ? &vpt->GetDammageRect() :
    NULL,
        COMP_ALL, inode->NumMtls());
    return(0);
}
```

There is a case when world space modifiers are applied to an object where the points of the object may have already been transformed into world space. If a world space modifier has been applied, the points of the object may have already been transformed into world space and then deformed by the world space modifier. With the bent, rippled, cylinder example above, this is exactly what has happened. When the ripple space warp was applied, the points of the object were transformed into world space. In this case, the points should not be transformed into world space again when they are drawn (since they were already by the space warp). The problem is, the object does not know if it has been transformed into world space or not.

The way 3ds max handles this situation is by storing some state information with the node. Before the system calls **Display()**, **HitTest()**, etc. on the object it sets a flag. The flag indicates if the object has already been transformed into world space. The **GetObjectTM()** method looks at this flag to determine the proper matrix to return. In this way, when **GetObjectTM()** is called, it returns the matrix the object needs to be multiplied by in order to get into world space. If the object is already in world space it will return the identity matrix. If it's not in world space, it will return the matrix to get it there. So all any objects need to do in their **Display()** methods is call **GetObjectTM()** and use whatever matrix is returned.

There may be times when a developer needs to access the full object TM regardless of whether the points of the object have been transformed into world space already. For example, if a developer was creating a utility plug-in to align two objects. In this case, the developer would need to get the full object TM including the NodeTM and the object offset transformation. It would not matter if the points of the object had already been transformed into world space, the TM is what matters. In this case **GetObjectTM()** would not work. This is because it returns the identity matrix if the object is already in world space.

To solve this problem 3ds max provides two other **Inode** methods that may be used.

### **GetObjTMBeforeWSM()**

This method explicitly gets the full NodeTM and object-offset transformation affect before the affect of any world space modifiers.

### **GetObjTMAfterWSM()**

This method explicitly gets the full NodeTM and object-offset transformation and world space modifier affect unless the points of the object have already

been transformed into world space in which case it will return the identity matrix.

Using these methods a developer has complete access to any transformation matrix they require. Below is a code example that uses all these methods. This function computes the bounding box of the first object in the current selection set at the current time. It removes anything but scaling from the Object TM. In this way rotation of the node will not affect the bounding box.

To do this we first need to determine if the object is in world space or in object space. Since we are after the object space bounding box (and will later apply scaling) we need to convert the TM back into object space if it is in world space. To check if the object is in world space we call **GetObjTMAfterWSM()**. If this matrix is the identity we know we are in world space. This is because when the points of the object get transformed by the ObjectState TM to put them into world space the ObjectState TM is set to the identity. Therefore if the matrix is the identity we are in world space.

If the object is in world space we need to compute the object space TM. We can do this by taking the inverse of the world space TM.

If the object is not in world space we just need to get its object TM by calling **GetObjectTM()**.

Once we have the object space TM we want to extract just the scaling portion of the matrix. 3ds max provides a set of APIs that make this easy. This is done by calling **decomp\_affine()**. This function decomposes a matrix into its translation, rotation and scaling components. See [Structure AffineParts](#) for more details.

Once we have the scaling portion of the matrix we can get the bounding box and apply the scaling by calling **GetDeformBBox()**.

```
void Utility::ComputeBBox(Interface *ip) {  
    if (ip->GetSelNodeCount()) {  
        INode *node = ip->GetSelNode(0);  
        Box3 box; // The computed box  
        Matrix3 mat; // The Object TM  
        Matrix3 sclMat(1); // This will be used to apply the scaling  
        // Get the result of the pipeline at the current time  
        TimeValue t = ip->GetTime();  
        Object *obj = node->EvalWorldState(t).obj;
```

```

// Determine if the object is in world space or object space
// so we can get the correct TM. We can check this by getting
// the Object TM after the world space modifiers have been
// applied. If the matrix returned is the identity matrix the
// points of the object have been transformed into world space.
if (node->GetObjTMAfterWSM(t).IsIdentity()) {
    // It's in world space, so put it back into object
    // space. We can do this by computing the inverse
    // of the matrix returned before any world space
    // modifiers were applied.
    mat = Inverse(node->GetObjTMBeforeWSM(t));
}
else {
    // It's in object space, get the Object TM.
    mat = node->GetObjectTM(t);
}
// Extract just the scaling part from the TM
AffineParts parts;
decomp_affine(mat, &parts);
ApplyScaling(sclMat, ScaleValue(parts.k*parts.f, parts.u));
// Get the bound box, and affect it by just
// the scaling portion
obj->GetDeformBBox(t, box, &sclMat);
// Show the size and frame number
float sx = box.pmax.x-box.pmin.x;
float sy = box.pmax.y-box.pmin.y;
float sz = box.pmax.z-box.pmin.z;
TSTR title;
title.printf(_T("Result at frame %d"),
    t/GetTicksPerFrame());
TSTR buf;
buf.printf(_T("The size is: (%.1f, %.1f, %.1f)"), sx, sy, sz);
MessageBox(NULL, buf, title,
    MB_ICONINFORMATION|MB_OK);

```

}  
}

## A Note About Caching

The full details of the 3ds max pipeline cache system are beyond what a developer needs to understand to effectively work with the pipeline. There is however one detail that may be useful for a Modifier that is being adjusted interactively.

While a modifier is being edited, in its implementation of **LocalValidity()**, it can return **NEVER**. This forces a cache to be built after the previous modifier. For example, the **SimpleMod** class does this. The pipeline will try to put a cache after something that is relatively constant but before something that is changing a lot. If a modifier for its local validity starts returning **NEVER**, this will cause a cache to be created before it. This is useful if a modifier is being edited interactively. In this way, the system does not have to evaluate the whole pipeline. This can considerably improve the interactivity.

After the modifier is done being edited, if the modifier is not animated, there is no need to have a cache before it. Therefore after the modifier is done being edited, it can stop returning **NEVER** for **LocalValidity()**. Below is a code fragment from the **SimpleMod** implementation of **LocalValidity()** where this is being done:

```
Interval SimpleMod::LocalValidity(TimeValue t)  
{  
    // If we are being edited, return NEVER to forces a cache to  
    // be built after previous modifier.  
    if (TestAFlag(A_MOD_BEING_EDITED))  
        return NEVER;  
    ...  
}
```

## **Modifier Stack Branching**

Compound objects such as the boolean object and the lofter can actually cause the pipeline to branch. See the advanced topics section on [Modifier Stack Branching](#) for the details on the methods a compound object uses to implement branching in the pipeline.

## Objects Flowing through the Pipeline

Certain plug-in objects flow through the pipeline. Examples are both the **TriObject** and the **PatchObject**. Most plug-ins do not however because they convert themselves to **TriObjects** or **PatchObjects** and these objects flow through the pipeline. For developers creating objects that flow through the pipeline there are some pipeline concepts and specific methods that must be understood. These are discussed in this section.

Most of these methods relate to minimizing the amount of overhead present within the system when the object is flowing through the pipeline. To be as efficient as possible the system will try not to create any extra copies of the object flowing down the pipeline. Additionally, the object is broken into channels and the individual channel copying is kept to a minimum.

To accomplish this, the system uses a set of 'locks' that indicate when it is not okay to free memory or modify the objects. The object lock and channel lock methods are implemented and called by the system. The plug-in developer implements methods to maintain validity intervals for the channels, create new copies of channels as needed, and free the memory associated with channels that are no longer needed. Again, the main purpose of this is to minimize the overhead of the object flowing down the pipeline.

Every object has what is referred to as its "shell". The shell has channels within it. Example channels are the geometry channel named **GEOM\_CHANNEL** (typically an array of vertices) or the topology channel named **TOPO\_CHANNEL** (typically an array of faces). Some channels are always present in the shell. For example the **SUBSEL\_TYPE\_CHANNEL**, **SELECT\_CHANNEL** and **DISP\_ATTRIB\_CHANNEL** state channel are each just a single value. These are not allocated or de-allocated dynamically, and are always present. Other channels like the **GEOM\_CHANNEL**, **TOPO\_CHANNEL**, and **TEXMAP\_CHANNEL** are allocated dynamically so the system tries to not have extra copies of them when possible. The following methods from class **Object** deal with the state of the shell. These methods are implemented and called by the system.

### **void LockObject()**

This method locks the object as a whole.

### **void UnlockObject()**

This method unlocks the object as a whole.

## **int IsObjectLocked()**

Returns nonzero if the object is locked; otherwise 0.

If the shell is locked then it should not be deleted by the pipeline. For example, a sphere in the pipeline will be locked. This is because it exists in the scene and thus should not be deleted. If a bend modifier was applied to the sphere, then the sphere would convert itself to a TriObject. This TriObject is flowing down the pipeline and is just a temporary object. This object will be unlocked meaning that it may be deleted. This is true unless a ModApp decides to cache it. In this case it will become locked because the ModApp has taken ownership of it and therefore it should not be deleted. Note that these methods are implemented and called by the system and not the plug-in object. They simply manipulate a private data member inside the **Object** class.

There is a related topic to discuss -- channel locking. This is not the locking of the object as a whole (as described above) but rather the locking of only certain channels within the object. For example, a ModApp may have the geometry channel cached (say it has an array of vertices cached somewhere in the pipeline). When the system is evaluating this pipeline, if it notices a certain channel it requires is cached, instead of evaluating the rest of the pipeline, it will use the cache. It will do what is called a **shallow copy** of the cached channel into the object going down the pipeline. This is just copying the pointer to the cached channel into the object flowing down the pipeline. So essentially there are two TriObjects whose geometry channels are both pointing to the same array of vertices. This reduces the memory overhead required because instead of copying the whole array of vertices there are essentially two meshes that are sharing the same block of memory. Again this is referred to as a shallow copy. The system (MAX) takes care of all of this. It carefully keeps track of who owns what so things don't get deleted twice or get deleted when still being used.

The channel lock methods below deal with this system. These methods are implemented and called by the system and not the plug-in object. They simply manipulate a private data member inside the **Object** class.

## **void LockChannels(ChannelMask channels)**

Locks the specified channels of the object.

## **void UnlockChannels(ChannelMask channels)**

Unlocks the specified channels of the object.

## **ChannelMask GetChannelLocks()**

Returns the locked status of the channels.

### **void SetChannelLocks(ChannelMask channels)**

Sets the locked status of the object's channels.

### **ChannelMask GetChannelLocks(ChannelMask m)**

Returns the locked status of the specified channels.

If a channel is locked, this means that the object does **not** own the channel and it should not free it. If the channel is unlocked, this means the object does own it, and it may free it. A channel that is locked should also not be modified. For example, if a channel that was cached upstream in the pipeline was modified, the cached version would not be correct anymore. The locking of the cached channel prevents this modification from happening.

The **ChannelMask** that appears in the above methods is an unsigned long. Each channel is represented by a bit. Note: Developers must not get confused between channel numbers (**TOPO\_CHAN\_NUM**, **GEOM\_CHAN\_NUM**, etc.) and channel bits (**TOPO\_CHANNEL**, **GEOM\_CHANNEL**, etc.). Some methods refer to the channel by number and some by bit. Developers must not confuse these two as the compiler will not catch this as an error.

The meaning of the channels are defined by the object. The three main channels (those that get allocated dynamically) are the **GEOM\_CHANNEL**, **TOPO\_CHANNEL**, and the **TEXMAP\_CHANNEL**. The other channels are always present. The developer must determine what part of their object falls into the **GEOM\_CHANNEL**, **TOPO\_CHANNEL**, and the **TEXMAP\_CHANNEL**. The TriObject defines the **GEOM\_CHANNEL** to mean the points or vertices of the mesh. The TriObject defines the **TOPO\_CHANNEL** to mean the face or polygon structures. This includes the smoothing groups and materials as well.

It is up to the object flowing down the pipeline to store validity intervals for each channel. Consider a sphere object with an animated radius parameter. The validity interval for the geometry channel for this sphere will be instantaneous (valid for a single **TimeValue**). This is because the radius is animated and is thus always changing. However the topology channel of the sphere is never changing. Therefore the validity interval for the topology channels will be **FOREVER**. See the Advanced Topics section on [Intervals](#) for more details. The methods below are implemented by the developer of the plug-in object unless noted otherwise.

### **virtual Interval ChannelValidity(TimeValue t, int nchan);**

Retrieve the current validity interval for the nchan channel of the object.

**virtual void SetChannelValidity(int nchan, Interval v);**

Sets the validity interval of the specified channel.

**void UpdateValidity(int nchan, Interval v);**

Implemented by the system. This method is called to AND in interval **v** to the specified channel validity.

**virtual void InvalidateChannels(ChannelMask channels);**

This method invalidates the intervals for the given channel mask. This just sets the validity intervals to empty (calling **SetEmpty()** on the interval) .

At certain times the system must ask the plug-in object to prepare certain channels for modification. For example, if a channel is cached upstream in the pipeline, and was modified, the cached version would be modified as well (since they both point to the same memory). This would confuse the system and thus locked channels must not be modified. If the system needs to modify one of the channels it will call the following method implemented by the system.

**void ReadyChannelsForMod(ChannelMask channels);**

This method is used to make the channels specified by the channel mask writable.

As mentioned above, a channel that is locked should not be modified. This method will take the channels that are locked and allocate a new block of memory for them. It will then copy the locked channels into the new block of memory and set the new memory as the current channel. Then it will unlock the channel. It does this by calling **NewAndCopyChannels()** which is implemented by the plug-in and is described below. In this way the system can modify the channels and not affect the cached copy as they no longer point to the same memory.

The following methods also deal with the allocation and copying of object channels. These methods are implemented by the plug-in.

**virtual Object \*MakeShallowCopy(ChannelMask channels);**

This method creates a new shell and then shallow copies in the channels that are specified.

**virtual void ShallowCopy(Object\* fromOb, ChannelMask channels);**

This method is passed the shell, and it copies the specified channels into it.

The shallow copy just copies the pointers (for example, the vertices pointer or

the faces pointer).

**virtual void NewAndCopyChannels(ChannelMask channels);**

This method takes the channels specified and clones them, and makes them read only (by locking them).

**virtual void FreeChannels(ChannelMask channels);**

This method deletes the memory associated with the specified channels and set the intervals associated with the channels to invalid (empty).

The following method is related to caching and shallow copying. This method is only implemented by particle systems. Particle systems bypass a lot of how the pipeline works and so they implement this method to ensure that they are never cached. Particle systems handle their own caching mechanism. All objects other than particle system can use the default implementation which returns TRUE. Particles can override this and return FALSE.

**virtual BOOL CanCacheObject() {return TRUE;}**

Developers that have an object that flows down the pipeline may want to take a look at the source code for the **TriObject** which provides implementations of all the methods discussed above. This code is available in **\MAXSDK\SAMPLES\HOWTO\MISC\TRIOBJECT.CPP**.

## Getting and Setting User Preferences

See Also: [Class Interface](#), [Class ViewExp](#), [Class Point3](#).

## **Overview**

This section provides an overview of the global functions, classes and methods used to retrieve and set the user preferences of MAX. Many of these are from the various pages of the Customize / Preferences... dialog.

## User Interface Colors

A 3ds max user may set various colors in the user interface via the settings in the Customize... Preferences... Color tab dialog. A developer may access and set these settings using the following global functions:

### Prototype:

**Point3 GetUIColor(int which);**

### Remarks:

This function is available in release 2.0 and later only.

Returns the specified color value for drawing various items in the 3ds max viewports.

### Parameters:

**int which**

Specifies which color to retrieve. See [List of Viewport Drawing Color Indices](#).

### Prototype:

**void SetUIColor(int which, Point3 \*clr);**

### Remarks:

This function is available in release 2.0 and later only.

Sets the specified color value for drawing various items in the 3ds max viewports.

### Parameters:

**int which,**

Specifies which color to set. See [List of Viewport Drawing Color Indices](#).

**Point3 \*clr**

The color value to set.

### Prototype:

**Point3 GetDefaultUIColor(int which);**

### Remarks:

This function is available in release 2.0 and later only.

Returns the default color used for drawing various items in the 3ds max user interface. The values returned are not affected by the user's color selections or

those set by **SetUIColor()**.

**Parameters:**

**int which**

Specifies which color to retrieve. See [List of Viewport Drawing Color Indices](#).

Note the following #defines for getting the selection color, sub-object selection color and the frozen object color.

```
#define GetSelColor() GetUIColor(COLOR_SELECTION)
```

```
#define GetSubSelColor()
```

```
GetUIColor(COLOR_SUBSELECTION)
```

```
#define GetFreezeColor() GetUIColor(COLOR_FREEZE)
```

```
#define SMALL_VERTEX_DOTS 0
```

```
#define LARGE_VERTEX_DOTS 1
```

**Function:**

```
void setUseVertexDots(int b);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the Use Vertex Dots preference to on or off. This corresponds to the 'Show Vertices As Dots' in the 3ds max user interface.

**Parameters:**

**int b**

Nonzero for on; zero for off.

**Function:**

```
int getUseVertexDots();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the Use Vertex Dots preference -- nonzero for on; zero for off.

**Function:**

**void setVertexDotType(int t);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the Vertex Small Dots / Large Dots preference. This corresponds to the 'Show Vertices As Dots' Small Dots / Large Dots radio buttons in the 3ds max user interface.

**Parameters:**

**int t**

One of the following values:

**SMALL\_VERTEX\_DOTS**

**LARGE\_VERTEX\_DOTS**

**Function:**

**int getVertexDotType();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the Vertex Small Dots / Large Dots preference. One of the following values:

**SMALL\_VERTEX\_DOTS**

**LARGE\_VERTEX\_DOTS**

## **Access to the MAX Viewport settings:**

Access to the user settings includes the 3ds max toolbars, snap settings, crossing setting, animate button, 3ds max window handle, sub-object selection color, axis constraints, coordinate centers, and reference coordinate system setting.

See Class Interface - Access to User Interface Properties and Controls. The above settings and more are accessed using these methods.

Some of the viewport settings are global in nature -- that is they affect all the viewports. These methods are in class **Interface**. Examples are **getBkgImageName()**, **GetGridSpacing()**, etc.

Other viewport settings are specific to a viewport. These are available in class **ViewExp**. Examples are **getBkgImageDsp()**, **getSFDisplay()**, etc.

## **Access to the MAX gamma settings:**

The [Class GammaMgr](#) allows access to the user specified gamma settings for display gamma, file input and output gamma.

## System Settings

There is also another API that allows a plug-in to query various system settings.

### **int GetSystemSetting(int id);**

This method will return nonzero if the setting whose **id** is passed is on; otherwise zero.

You may pass the following values:

#### **SYSSET\_ENABLE\_EDITABLEMESH**

Used to verify if TriObjects created will be the standard ones or the editable variety. Nonzero is returned if editable meshes will be created; otherwise zero for standard TriObjects.

#### **SYSSET\_CLEAR\_UNDO**

For release 1.1 and later if this ID is passed to this method the undo buffer is flushed.

#### **SYSSET\_EDITABLEMESH\_ENABLE\_KEYBOARD\_ACCEL**

This option is available in release 2.0 and later only.

Determines if keyboard accelerators are enabled for the editable mesh. Nonzero if so; otherwise zero.

#### **SYSSET\_ENABLE\_EDITMESHMOD**

This option is available in release 2.0 and later only.

Determines if the edit mesh modifier is enabled. Nonzero if so; otherwise zero.

# Globalization

See Also: [Character Strings](#).

3D Studio is sold in numerous countries outside the United States. To allow your plug-in to have the largest possible audience, it is a good practice to create a globalized application. This means the language-specific strings used in the plug-in can be easily translated to another language.

To accomplish this, you should separate out all your language specific strings into a separate resource-only DLL. In this way, only the resource DLL, and not the rest of the code, needs to be changed to move the application to another language.

The basic concepts involved in creating and using a resource-only DLL are simple:

- Create a VC++ project that builds a separate DLL containing only the resources of the plug-in.

- When your plug-in begins execution, in the **DllMain()** function, attempt to load the resource DLL. This is done by calling a function of the Windows API **LoadLibraryEx()**. This maps the specified executable module into the address space of the calling process, making the strings accessible to the plug-in.

- After the resource DLL has been loaded and you need to use a string, use the Windows function **LoadString()**. This loads the string from the resource DLL and copies it into a local string buffer. The local string is then used as usual.

- Release the resource DLL when the plug-in is finished executing. This may be done using the Windows **FreeLibrary()** function. Call this from the **DLL\_PROCESS\_DETACH** case of the **DllMain()** function. This decrements the reference count of the loaded DLL module. When the reference count reaches zero, the module is unmapped from the address space of the calling process.

The sample code below demonstrates how this is done. It shows two functions from the EPS file format I/O module.

The first function is **DLLMain()**. Note the use of **LoadLibraryEx()** and **FreeLibrary()**. Also see how **GetModuleFileName()** is used to find the location of the DLL (placed in the same directory as the plug-in). For details on

these functions see the Windows API on-line help.

```
#define MAX_PATH_LENGTH 257
#define MAX_STRING_LENGTH 256
int triedToLoad = FALSE;
int resourcesLoaded = FALSE;
HINSTANCE hResource = NULL;
BOOL WINAPI DllMain(HINSTANCE hinstDLL,
    ULONG fdwReason,LPVOID lpvReserved)
{
// If we have already tried to load the resource
// file and failed just give up.
if (triedToLoad && ! resourcesLoaded)
return FALSE;
// Load our resources. We look for the file in the
// same directory where this DLL was found
if (! resourcesLoaded) {
// Where this DLL resides
char dirName[MAX_PATH_LENGTH];
// Full path name to resource DLL
char dllName[MAX_PATH_LENGTH];
char *chPtr;
GetModuleFileName (hinstDLL, dirName,
MAX_PATH_LENGTH);
// Strip off the file name
chPtr = dirName + strlen (dirName);
while (*(--chPtr) != '\\')
;
*(chPtr+1) = 0;
// Add in "epsres.dll"
strcpy (dllName, dirName);
strcat (dllName, "epsres.dll");
// Load resource DLL
// Turn off error reporting
int errorMode =
```

```

SetErrorMode(SEM_NOOPENFILEERRORBOX);
hResource = LoadLibraryEx(dllName, NULL, 0);
SetErrorMode(errorMode);
// Be sure to check to see if we succeeded
// loading resource DLL
if (hResource) {
resourcesLoaded = TRUE;
InitCustomControls (hResource);
} else {
triedToLoad = TRUE;
MessageBox (NULL,
"EPS Plugin failed to load due to missing resource file
EPSRES.DLL",
"EPS", MB_ICONINFORMATION);
return FALSE;
}
}
switch(fdwReason) {
case DLL_PROCESS_ATTACH:
break;
case DLL_THREAD_ATTACH:
break;
case DLL_THREAD_DETACH:
break;
case DLL_PROCESS_DETACH:
if (hResource)
FreeLibrary (hResource);
break;
}
return(TRUE);
}

```

To load and use one of the strings from the resource DLL, follow the approach shown below:

```

const TCHAR *EPSClassDesc::ClassName () {

```

```
static int loaded = 0;
static TCHAR stringBuffer[MAX_STRING_LENGTH];
if (! loaded) {
LoadString (hResource, IDS_CLASS_NAME, stringBuffer,
MAX_STRING_LENGTH);
loaded = 1;
}
return stringBuffer;
}
```

In summary, it is a good development practice to separate the literal strings of a plug-in into a separate resource-only DLL. This creates a globalized application with the potential to reach the largest possible audience. For additional information on the plug-in functions shown above (**DLLMain()** and **ClassName()**) see the Advanced Topics section [DLL Functions and Class Descriptors](#).

## Hit Testing

See Also: [Class BaseObject](#), [Class GraphicsWindow](#), [Class Mesh](#), [Class ViewExp](#), [Class Interface](#), [Class HitRecord](#), [Class HitRegion](#), [Class HitLog](#).

Hit testing is used throughout 3ds max as the user selects items in the scene using the mouse. Hit testing is the process of determining if a given mouse point intersects an item (node, modifier gizmo/center, or controller gizmo). A plug-in developer's responsibility regarding hit testing depends upon the type of plug-in and how much functionality is supplied by the base class the plug-in is derived from. For example, plug-ins that are sub-classed from **SimpleObject**, **SimpleMod** or **StdControl** are not required to implement methods for hit testing as it is all handled internally by the base class. Other plug-ins that don't subclass from these classes (for example the boolean object or the mapping modifier) must perform their own hit testing.

There are two types of hit testing:

Node level. This is determining if a given node in the scene has been hit.

Sub-object level. This can be for objects, modifier or controllers. This is for determining if a sub-component of an item has been hit. Object space modifiers can have a visual representation in the scene that the user can manipulate. For example, many of the 3ds max object space modifiers have a gizmo and center mark. These are the modifier's sub-object levels. In the case of an edit modifier this can be the sub-object parts of the object itself (like vertices, edges, or face for the Edit Mesh modifier). An object may hit test sub-object parts as well. For example, a compound object like the boolean object may hit test the operands of the boolean operation. The loft model compound object may hit test parts of the loft model such as the path or shapes.

There is also a distinction to be made between 'simple' hit testing and 'smart' hit testing. Simple hit testing involves finding **any** hit on an item. As soon as a single hit is found the process of searching for hits can stop. Smart hit testing involves finding the part of an item that was the **closest** to the pick point. This involves hit testing everything and then searching through all the hits to find the closest one.

If the plug-in procedural object, modifier or controller must perform its own hit testing, methods of the GraphicsWindow class make this fairly simple. There is a special rendering mode that items may be drawn or rendered in that performs the hit testing. In this mode (called **GW\_PICK**), the item is not actually drawn, but

is instead tested for intersection with the specified hit region. So if an item needs to be hit tested, a developer must simply draw the item using a rendering level **GW\_PICK** and then check the result. The following sample code demonstrates controller gizmo hit testing. This method hit tests a series of 'footstep' shaped polylines that serves as the controller's gizmo.

```
int FootStepControl::HitTest(TimeValue t, INode* inode,
int type, int crossing, int flags, IPoint2 *p, ViewExp *vpt) {
int savedLimits, res = 0;
GraphicsWindow *gw = vpt->getGW();
Matrix3 ntm = inode->GetNodeTM(t);
HitRegion hr;
MakeHitRegion(hr,type,crossing,4,p);
gw->setHitRegion(&hr);
gw->setRndLimits(((savedLimits =
gw->getRndLimits()) | GW_PICK) & ~GW_ILLUM);
gw->clearHitCode();
BOOL abortOnHit = flags&SUBHIT_ABORTONHIT?
TRUE:FALSE;
BOOL selOnly = flags&SUBHIT_SELONLY?TRUE:FALSE;
BOOL unselOnly = flags&SUBHIT_UNSELONLY?
TRUE:FALSE;
for (int i=0; i<NUM_FOOTSTEPS; i++) {
if (selOnly && !sel[i]) continue;
if (unselOnly && sel[i]) continue;
gw->setTransform(fs[i]*ntm);
gw-
>polyline(FOOT_POINTS,footPts,NULL,NULL,TRUE,NULL);
if (gw->checkHitCode()) {
res = TRUE;
vpt->CtrlLogHit(inode,gw->getHitDistance(),i,0);
if (abortOnHit) {
break;
}
}
gw->clearHitCode();
```

```

    }
    }
    gw->setRndLimits(savedLimits);
    return res;
}

```

Note that the example above hit tests polylines. If you are hit testing polygons they must be drawn as 3 sided entities for 3ds max to properly hit test them. Polygons drawn with more than 3 sides will not hit test properly. For more information on hit testing using the methods of Graphics Window see the Advanced Topics section on [The Interactive Renderer : GraphicsWindow](#).

### Setting hitCode and hitDistance in R4.0

**setHitCode()** and **setHitDistance()** are new methods that make it possible to work with GraphicsWindow hit-testing in otherwise impossible situations. Why are they necessary? An example from MAX's CORE.DLL is shown below.

#### Sample Code:

The patch object contains bezier spline-based edges which can consist of up to 102 vertices. Since the **GraphicsWindow::polyline** function can only plot lines with up to 32 vertices, it is impossible to plot these in a single call to the polyline function. Multiple calls to the polyline call do not return a proper hitcode when using a "window"-type hit region. By using the new setHitCode method, code can properly handle this situation. The code below shows the function in use from the **PatchMesh::renderEdge** method:

```

    int steps = GetMeshSteps();
    int segNum = steps+2;
    float fsegNum = (float) (segNum-1);
    // If steps are too high for GraphicsWindow's buffer,
    // we must draw it manually
    if((steps + 2) > GW_MAX_VERTS) {
        Point3 line[2];
        Point3 prev,current(.0f,.0f,.0f);
        BOOL hitAll = TRUE;
        BOOL hitAny = FALSE;
    }

```

```

        DWORD hitDist = 0xffffffff;
for(int terp = 0; terp < segNum; terp++) {
prev = current;
current = work.InterpCurve3D((float)terp / fsegNum);
if (terp != 0) {
line[0] = prev;
line[1] = current;
gw->clearHitCode();
gw->polyline(2, line, NULL, NULL, 0, NULL);
if(gw->checkHitCode()) {
hitAny = TRUE;
if(gw->getHitDistance() < hitDist)
hitDist = gw->getHitDistance();
}
else hitAll = FALSE;
}
}
if(hr && !hr->crossing && hr->type != POINT_RGN)
gw->setHitCode(hitAll);
else
gw->setHitCode(hitAny);
gw->setHitDistance(hitDist);
}
else {
for(int terp = 0; terp < segNum; terp++)
fixedBuf[terp] = work.InterpCurve3D((float)terp / fsegNum);
gw->polyline(steps+2, fixedBuf, NULL, NULL, 0, NULL);
}
}

```

Note that the **gw->polyline** call is preceded by a call to **clearHitCode**, and followed by code which checks the hit code, maintaining "hitAny" and "hitAll" flags. When all the segments are drawn, the **gw->setHitCode** call is made, setting the hit code depending on the hit region type. When the code which called this function checks the GraphicsWindow's hit code, it will contain the

proper value. This code also keeps track of the closest hit distance and places that into the GraphicsWindow when all line segments are drawn.

## Sub-Object Hit Testing

Sub-object hit testing is similar to object level hit testing except that the object is not simply determining if it was hit or not; the sub-object element that was hit needs to be determined as well.

What exactly a sub-object element is depends on the modifier. It may be a modifier's gizmo or part of its gizmo. For example, when an FFD modifier is in vertex sub-selection mode, the sub-object elements that are being hit tested are the control points of the lattice. Edit modifiers usually hit test components of the object in the pipeline such as vertices or faces.

When a modifier's **HitTest()** method is called, it traverses the sub-elements of the current sub-object selection level and checks each one to see if it has been hit. If so, it registers a hit record with the active viewport. A hit record contains the following information:

- A pointer to the node that was hit. Modifiers may be instanced across multiple nodes, so the instance that was hit must be identified.

- A pointer to the ModContext.

- The 'distance' of the hit. To classify as a hit, the sub-object component must be within some threshold distance of the mouse. This distance is recorded in the hit record so that of all the hits below the threshold, the one that is the closest can be identified. What the distance actually represents depends on the rendering level. For wireframe modes, it refers to the distance in the screen XY plane from the mouse to the sub-object component. In a shaded mode, it refers to the Z depth of the sub-object component. In both cases, smaller distances indicate that the sub-object element is 'closer' to the mouse cursor.

- A general unsigned long value. Most modifiers will just need this to identify the sub-object element. The Edit Mesh modifier uses the value to store the index of the vertex or face that was hit.

- In case the 4 bytes above isn't enough space to identify the sub-object element, a pointer to a **HitData** class is included. To use this, a developer would define a class derived from this class that would contain the necessary data. The **HitData** class has one member function, a virtual destructor, so the derived class can be properly deleted when the **HitRecord** instance is deleted.

When **HitTest()** is called on an edit modifier, the edit modifier needs to hit test elements of the object that is flowing through the pipeline. This presents a problem since a modifier usually only has a pointer to this object when its

**ModifyObject()** method is called. In order to be able to hit test the object, a modifier must keep the object cached. Caching the object the modifier is modifying is generally useful as well. For example, the Edit Mesh modifier keeps a cache of the mesh it is editing. When the Edit Mesh modifier is evaluated, if it has a cache it can simply return the cache. This way, as the user applies incremental edits to the mesh, all of the previous edits don't need to be reapplied. Instead, when an edit is made it is applied to the cache *and* stored away in some form in case the modifier needs to be reapplied.

Again, modifiers can be instanced, so storing an instance-specific cache in the modifier might not be the best idea. It is the [ModContext](#), specifically in the **localData** field, where instance-specific modifier data belongs. This is where edit modifiers can put a cache of the object that it modifies. NOTE: typically this cache is only needed while the object is being edited. A hit test will never be called on a modifier unless it is being edited. In terms of evaluation, the pipeline already places caches where appropriate so there is normally no need for the modifier to maintain a cache.

Below is a summary of the hit test related methods from [Class GraphicsWindow](#).

**virtual void setHitRegion(HitRegion \*rgn) = 0;**

Sets the hit region used for hit testing.

**virtual void clearHitCode() = 0;**

This methods clears the hit code. Call this method before performing a hit test.

**virtual BOOL checkHitCode() = 0;**

Returns TRUE if a hit was made; otherwise FALSE.

**virtual DWORD getHitDistance() = 0;**

If **checkHitCode()** returns TRUE you may call this method to return the hit distance. In wireframe mode this is the distance to the line. In shaded mode, this is the z distance. This allows you to perform 'smart' hit testing by choosing the item with the smallest hit distance.

The system maintains a list of nodes for node level hit testing, a **HitLog** for modifier sub-object hits, and a **CtrlHitLog** for controller sub-object hits. The methods of [Class ViewExp](#) are used to work with these results. After hit testing has been performed these methods are available to examine the results.

For node level hit-testing

**virtual void ClearHitList()=0;**

Clears the list of hits.

**virtual INode \*GetClosestHit()=0;**

Returns the INode pointer of the node that was the closest all those hit.

**virtual int HitCount()=0;**

Returns the number of hits recorded.

For modifier sub-object level hit-testing

**virtual void LogHit(INode \*nr, ModContext \*mc, DWORD dist, ulong info, HitData \*hitdat = NULL)=0;**

This method records a sub-object level hit record with the system using the specified parameters. This is frequently called when a plug-in needs to select a node from the scene.

**virtual HitLog& GetSubObjHitList()=0;**

Returns the sub-object hit list.

**virtual void ClearSubObjHitList()=0;**

Clears the sub-object hit list.

**virtual int NumSubObjHits()=0;**

Returns the number of sub-object hits.

For controller gizmo hit testing

**virtual void CtrlLogHit(INode \*nr,DWORD dist,ulong info, DWORD infoExtra)=0;**

This method records a sub-object level hit record with the system using the specified parameters.

**virtual CtrlHitLog& GetCtrlHitList()=0;**

Returns the sub-object hit list.

**virtual void ClearCtrlHitList()=0;**

Clears the sub-object hit list.

If a plug-in wants to perform hit testing of other nodes in the scene it may use the following methods of Class Interface.

**virtual INode \*PickNode(HWND hWnd, IPoint2 pt)=0;**

This method hit tests the screen position for nodes and returns a INode pointer if one is hit, NULL otherwise.

**virtual int SubObjHitTest(TimeValue t, int type, int crossing, int flags, IPoint2 \*p, ViewExp \*vpt)=0;**

This method performs a sub-object hit test. You may access the number of hits

using: **vpt->NumSubObjHits()**. To return a list of the hits use **vpt->GetCtrlHitList()**.

The following code demonstrates the use of this method:

```
BOOL GenControlSelectionProcessor::HitTest(  
  ViewExp *vpt, IPoint2 *p, int type, int flags )  
  {  
  vpt->ClearCtrlHitList();  
  ip->SubObHitTest(ip->GetTime(),type,ip-  
  >GetCrossing(),flags,p,vpt);  
  if (vpt->GetCtrlHitList().First()) {  
    return TRUE;  
  } else {  
    return FALSE;  
  }  
}
```

Plug-ins that are not subclassed from **SimpleObject SimpleMod** or **StdControl** need to implement one of the methods shown below. These allows the system to hit test the plug-in item.

From [Class BaseObject](#). This is the procedural object version:

```
virtual int HitTest(TimeValue t, INode* inode, int type,  
  int crossing, int flags, IPoint2 *p, ViewExp *vpt);
```

The modifier version of this method has an extra argument (**ModContext \*mc**).

```
virtual int HitTest(TimeValue t, INode* inode, int type,  
  int crossing, int flags, IPoint2 *p, ViewExp *vpt, ModContext*  
  mc)
```

From [Class Control](#).

```
virtual int HitTest(TimeValue t, INode* inode, int type,  
  int crossing, int flags, IPoint2 *p, ViewExp *vpt)
```

Finally, the following function is commonly used to initialize the **HitRegion** data structure.

```
void MakeHitRegion(HitRegion& hr, int type,  
  int crossing, int epsi, IPoint2 *p);
```

# The Interactive Renderer: GraphicsWindow

See Also: [Class GraphicsWindow](#), [Class Mesh](#), [Class Light](#), [Class HitRegion](#).

## Overview

The display of geometric structures in 3ds max is handled through a class called [GraphicsWindow](#). An instance of a GraphicsWindow sets up access to underlying drivers that allow 2D and 3D primitives to be rasterized and appear on-screen.

The methods in the GraphicsWindow class are designed to provide a fast, low-impedance pipeline to the underlying driver, and are optimized for mesh rendering.

If you wish to display a geometric object that is represented as a triangular mesh (consisting of a vertex list and a connectivity list), you should not use the GraphicsWindow methods directly. Rather, you should create a 3ds max [Mesh](#), and then call that mesh's **render()** method. This assures that your mesh is rendered using the optimal path through the display subsystem.

A GraphicsWindow provides the following services:

## **Window Access Services**

When the system creates an instance of a GraphicsWindow, the constructor opens a window for output and connects that window to an instance of a low-level rasterizer (driver). There are methods to resize and move the window, and to configure or change drivers.

Note: Since the driver has complete control over the window's palette (if one exists), each GraphicsWindow within 3ds max must be connected to the same driver.

The physical window consists of two parts: an image buffer (containing the colored pixels, as they appear on the screen) and a Z buffer. The Z buffer is used to record depth at each pixel as geometric primitives are rasterized.

Most drivers control two image buffers. One is displayed on the screen, and the other is used to rasterize geometric primitives. When rasterization of a complete frame is done, the off-screen buffer is blitted onto the display screen.

There are methods within the GraphicsWindow class that can be used to read/write the image and Z buffers.

## Rendering Modes

A GraphicsWindow instance can be in any of several different rasterizing modes. The various rendering modes may be combined in a large number of ways. For example, it is possible to request Gouraud-shaded, z-buffered lines, or textured, flat-shaded triangles. (The actual output will depend on the capabilities of the underlying device driver.)

Each GraphicsWindow has a master mode (called the rendering limit) and a current mode. The current mode is always a "subset" of the master mode, in that any limits imposed by the master mode are forced onto the current mode.

For example, if the master mode restricts primitives to wireframe rendering, then setting the current mode to filled polygons will have no effect. On the other hand, if the master mode restricts rendering to flat shading, then the current mode can be set to wireframe to force polygons to render at the wireframe level.

In MAX, the rendering limits for each viewport can be set in the Rendering Method page of the Views / Viewport Configuration dialog box. Note that even when the level is set to "Smooth + Highlights", some primitives (for example lights and cameras) appear in wireframe. This is done by setting their private mode to wireframe.

The various rendering modes may be combined in a large number of ways. For example, it is possible to get Gouraud-shaded, z-buffered lines, or textured, flat-shaded triangles.

## Coordinate Systems

There are two primary coordinate systems within a GraphicsWindow: device-independent 3D eye coordinates and device-dependent window coordinates.

Device-dependent coordinates are integer triples that have their origin at the lower-left corner of the window and increase toward the upper right. One unit corresponds to a single pixel. Z values closest to the user are 0, and they increase into the screen. The maximum Z value is driver-dependent. This coordinate system is left-handed.

Eye coordinates are floating point triples that correspond to relative distances from the camera (or "eye") associated with the window. In eye coordinates, x increases toward the right, y increases upward, and z decreases away (and in front of) the eye point. This coordinate system is right-handed.

There are two ways to set up eye coordinates. In the first, the camera is assumed to be sitting at the origin looking down the negative z axis. You specify camera parameters (perspective/orthographic, field-of-view, etc.) and then provide a transformation matrix for an object's position relative to the camera. This is most easily accomplished by concatenating the inverse affine transformation for the camera's position in world space with the (forward) affine transformation of the object in world space.

In the second method, a camera matrix is specified that includes both the world position (affine transformation) and projection information. This allows for a simple camera "look at" transformation model.

Note that the system stores two matrices: a camera matrix (which can be a pure projection), and an affine transformation matrix. When deriving device coordinates for a geometric primitive, each position is logically transformed through the affine transformation first, followed by the camera transformation. This allows each geometric primitive to be represented in its own local coordinates.

In addition to providing a method for transforming points from local, model coordinates to floating point device coordinates, a method is provided that transforms model coordinates into integer window coordinates with the origin at the upper left. These integer coordinates correspond to the coordinates used by GDI and the Windows mouse routines.

Both model-to-device coordinate methods return clipping values indicating whether the transformed point lies outside of the view volume. Flags are

provided for each of the six planes bounding the viewing frustum.

## Primitives

The geometric primitives supported by the GraphicsWindow class were designed to provide optimal support for rendering triangular meshes. As noted earlier, the preferred way to render such meshes is through the **render()** method of the Mesh class.

If you need to use the geometric primitives in the GraphicsWindow class directly, you should be aware that there is no direct support within GraphicsWindow for lit primitives, so rendering illuminated surfaces is a two-step process.

There are two levels of primitive support within the GraphicsWindow class: one set uses device-independent 3D eye coordinates and the other uses device-dependent window coordinates. At each level there are routines for polylines, polygons (triangles), markers, and annotation text.

The device-dependent routines do not provide any support for clipping or range testing! For speed, these routines assume that all coordinate values passed in are valid. If invalid values are present, the results are unpredictable, but a program crash or corruption of the 3ds max system is almost guaranteed.

The higher-level routines transform their coordinate lists through the current affine and camera transformations, clipping when necessary, and then call the corresponding low-level routines.

These routines are designed to optimize the rendering of meshes composed of vertex and connectivity lists. This process works as follows:

Each vertex is transformed using the model-to-device transformation routine, and the resulting device coordinates and clip flags are cached.

For each triangle in the connectivity list, the clip flags are examined to classify the triangle into one of the following three cases:

If all vertices lie outside the viewing frustum, the triangle is trivially rejected.

If all vertices lie inside the viewing region, it is rasterized using the cached device coordinates and the low-level primitive routines.

If the triangle is clipped by the viewing region, its model coordinates are sent to the high-level routines where the triangle is appropriately clipped and rasterized.

Note that this approach only transforms and clip-checks most vertices once. (Only in the rare case that a primitive is clipped do the associated vertices have to be transformed again.) Since most vertices in a mesh are shared, this provides

an efficient means to rasterize the entire mesh.

## Materials and Lighting

As mentioned above, the geometric primitives in a GraphicsWindow do not directly support lighting. Rather, polylines and polygons can be provided with colors for each vertex. If provided (and the rendering limit allows for it), the device driver will interpolate the vertex colors as the primitive is rasterized.

To facilitate the lighting process, there are methods to specify light properties (type, color, angles, etc.), position and direction, and to specify material properties (ambient, diffuse, and specular colors, etc.) Once a material and a set of lights are specified, a given vertex / normal vector pair may be "rendered" by calling the **lightVertex()** method. This method uses the position and normal vector direction to produce an RGB triple representing the lit color.

Logically speaking, for lit primitives the pipeline described above is modified by having each vertex lit at the time it is transformed. The resulting RGB triples can then be cached and later used during rasterization of the primitive. In the mesh class, the actual renderer uses a lazy evaluation algorithm so that only vertices that may appear on-screen get lit.

## Hit Testing

It is also possible to test each primitive for intersection or containment within a specified "hit region". The region may be a point (+/- epsilon), a rectangle, a circle, or a fence (an arbitrary polygon region).

Hit testing is done through a side-effect mechanism. When hit-testing is enabled, primitives set a flag if they pass through or are fully contained in the hit region. (When in this mode, the primitives do not cause any visual side-effects -- in particular, they do not change the image or Z buffers.)

By clearing the flag before a primitive is rendered, and checking it immediately afterwards, a hit-check for that particular primitive is made. Hit testing on vertices, edges, entire meshes, etc., can be accomplished by rendering only the desired primitives and checking the hit flag at the desired granularity.

When the hit region is a single point, each "hit" primitive also sets a distance variable (representing distance from the hit point to the line in wireframe mode, and Z distance in filled mode), so that "smart" hit testing -- i.e. choosing the closest primitive to the hit point -- may be implemented.

## Intervals

See Also: [Class Interval](#), Advanced Topics section [Time](#).

An Interval is a class that represents a length of time. It has two private data members, **start** and **end**, that are each **TimeValues**. A **TimeValue** is a single instant in time. For more information on **TimeValues** see the Advanced Topics section [Time](#).

Intervals are used throughout 3ds max in describing a range of time. The most common use is to describe a range of time over which an item is said to be 'Valid'. This type of interval is referred to as a **Validity Interval**. It is normally used in association with some item which is cached. The validity interval describes the range of time over which the cache accurately reflects the state of the item. When comparing a given time to see if the cache is up to date, if the time is inside the interval, the cache is valid at that time. If it is outside the interval, the cache is invalid.

The geometry pipeline system of 3ds max uses intervals as part of its caching scheme. Many of the methods procedural object or modifier plug-ins must call or implement (associated with intervals) are required for use with MAX's caching system. In terms of this cache, the interval represents a period of constancy for an item around a certain time. In other words, given a certain time, how far before and how far after the time is the item constant (not changing). Part of the caching algorithm needs to determine if an item is relatively constant or not.

As an example consider a procedural object which is not animated over an entire 100 frame animation. This object would have a validity interval of FOREVER. It is always up to date since nothing ever changes. Then apply a Bend modifier to this object and animate the angle parameter at frame 50. The bent object is now always changing from frame 0 to 50. After frame 50, it never changes. Any validity intervals computed in the first 50 frames would be instantaneous intervals (the **start** time equal to the **end** time). Any validity intervals computed at a time after frame 50 would be from 51 to 100.

3ds max may cache a representation of the bent object from frame 51 to 100. It will then never have to re-compute this bent object over this entire range. 3ds max would not cache a representation from frame 0 to 50 since the object is always changing.

Plug-ins must provide 3ds max with information about when they are changing

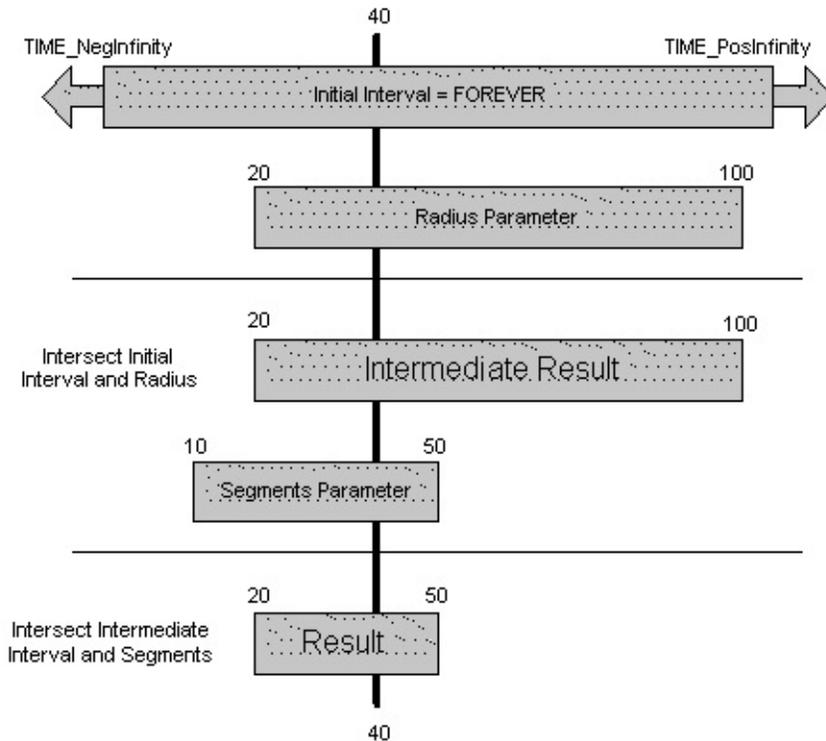
and when they are not. Below are some examples of methods which plug-in procedural objects or modifiers call or implement that return or modify intervals.

### **GetValue()**

This method of **IParamBlock** has several parameters, one of which is a C++ reference to an Interval. This method is frequently used by developers to 'whittle' down an interval. When a parameter of a parameter block is animated, for any given time there is an interval over which the parameter is constant. If the parameter is constantly changing the interval is instantaneous. If the parameter does not change for a certain period the interval will be longer. If the parameter never changes the interval will be FOREVER. By passing an interval to the **GetValue()** method of the parameter block you ask the parameter block to 'intersect' the interval passed in with the interval of the parameter. Intersecting two intervals means returning a new interval whose start value is the greater of the two, and whose end value is smaller of the two. In this way, the resulting interval represents a combined period of constancy for the two intervals.

This technique is used frequently to compute a validity interval for an object. The developer starts an interval off as FOREVER, then intersects this interval with each of its animated parameters (by calling GetValue()). GetValue() 'whittles' down the interval with each call. When all the parameters have been intersected, the result is the overall validity interval of an object at a specific time.

Consider the example shown in the diagram below. A validity interval is computed at time 40 for an object with two animated parameters, Radius, and Segments. The interval is computed by first intersecting an initial interval of FOREVER with the Radius parameter. The Radius interval is from 20 to 100. The result of this intersection is the interval from 20 to 100. This interval is intersected with the Segment parameter interval (from 10 to 50). The result of this intersection is an interval from 20 to 50. Thus the validity interval of the object at time 40 is from 20 to 50.



### **ObjectValidity(TimeValue t)**

This method, implemented by the plug-in, returns the validity interval of the procedural object around the time passed. This method is computed by the object by starting an interval at FOREVER, and intersecting this interval with the intervals of each of its animated parameters. In this way, an interval is whittled down as it is intersected with each of the parameters. The resulting interval represents a period of constancy about the TimeValue passed. The interval represents how far before the TimeValue and how far after it the object is constant.

### **LocalValidity(TimeValue t)**

This method returns the validity interval of the modifier itself around the time passed. In general, this would be the intersection of the validity intervals of all controllers that the modifier uses to control its parameters, so if a modifier was not animated, this interval would be FOREVER.

As an object flows up the pipeline, the validity interval of each modifier is intersected into the object state's validity interval. When the object gets to the end of the pipeline, its validity interval reflects the intersection of all the elements in the pipeline.

### **GetValidity(TimeValue t)**

The **SimpleMod** class calls this method to retrieve the validity interval of the modifier. The modifier provides this interval by starting an interval at FOREVER and intersecting each parameter of the modifier with the interval. **SimpleMod** then intersects the validity intervals of its own controllers with the returned interval in its implementation of **LocalValidity()**.

### **UpdateValidity(int nchan, Interval v)**

A modifier calls the **UpdateValidity()** method of an object. When a modifier is applied to an object, it needs to include its own validity interval. Frequently this is called by the modifier in its **ModifyObject()** method.

### **SetChannelValidity(int nchan, Interval v)**

This method is called to specify a validity interval for a certain channel of the pipeline.

See also: Class [Interval](#) and [Geometry Pipeline System](#) in the Reference section.

# Inverse Kinematics

See Also: [Class IKChainActions](#), [Class IKSolver](#), [Class IIKChainControl](#), [Class IIKControl](#), [Class ZeroPlaneMap](#), [Class RootLink](#), [Class Link](#), [Class LinkChain](#), [Class Animatable](#)

## Introduction to Inverse Kinematics

The Inverse Kinematics (IK for short) API's are divided over three files: **IIKSys.h**, **IKSolver.h**, and **IKHierarchy.h**. **IIKSys.h** contains the interface definition of the two types of controllers, **IIKControl** and **IIKChainControl**. They are both TM controllers. The former is used in joints (nodes on which IK will be applied), and the latter is used in the IK goals (nodes that serve as IK goals).

The **IKControl** and **IKChainControl** are knit together as the IK system. When an IK solution is called for, the IK system will invoke the IK solver, which is a mathematical algorithm that, given the goal position/orientation, and incidental parameters, produces a solution in terms of joint angles (rotation values of respective nodes). While the controllers involved in IK are not expandable via plug-in, the IK solver is actually plug-able. The IK system will recognize a plug-in solver by putting it in the solver list wherever it appears. The API of the IK solver, defined in **IKSolver.h** and **IKHierarchy.h**, contains the data structure of transformation hierarchy used in **IKSolver.h**.

An interface pointer to the **IKControl** class can be obtained by using **Animatable::GetInterface(I\_IKCONTROL)**.

An example on using the IK API can be found in the 3ds max SDK, **\MAXSDK\SAMPLES\IKSOLVERS\IKLIMB**.

## Concepts

### Degrees of Freedom

A node transform contains six degrees of freedom. Three are translational

(sliding) and three are rotational. In the API, they have enum names, **IKSys::TransX** to **IKSys::TransZ** and **IKSys::RotX** to **IKSys::RotZ**. When the context implies rotational or translational, **IKSys::DofX** to **IKSys::DofZ** are used.

A degree of freedom can be active or inactive. An inactive degree of freedom is not to be used by IK. In other words, only active degrees of freedom are degrees of freedom as far as IK is concerned. The methods **DofActive()**, **ActiveTrans()**, **ActiveRot()**, and **ActiveDofs()** are provided to determine which degrees of freedom are active and inactive. The first query answers whether a specific (translational / rotational) axis is active. The following three return a set of DOFs, which is defined in the same file. The set returned by **ActiveTrans()** and **ActiveRot()** is to be tested by **IKSys::DofX** to **IKSys::DofZ**, and the set returned by **ActiveDofs()** is to be tested by **IKSys::TransX** to **IKSys::RotZ**.

### Relationship to IK Chains

A node transformation can be conceptually viewed as consisting of two joints, sliding joint (translational degrees of freedom) and rotational joint. The enum type, **IKSys::JointType**, contains two enums: **IKSys::SlidingJoint** and **IKSys::RotationalJoint**. An IK chain starts at the rotational joint of the Start Joint and ends at the sliding joint of the End Joint. Two IK chains covering the same joint, are considered overlapping IK chains.

Let's call a node an IK chain node if its TM controller supports interface **IIKChainControl**, i.e., **node->GetTMController()->GetInterface(I\_IKChainControl) != NULL**. The following query returns a list of IK chain nodes; **InodeTab IKChains(JointType) const**  
The parameter that decides whether an individual degree of freedom is (IK) active is not animatable. There is an animatable variable of IK chain that decides whether the goal defined in the IK chain actually affects the joints it covers at a specific time, this can be queried using **IKBound()**.

### Joint Limits

A degree of freedom can be bound to a limited range. The lower bound and upper bound can be set separately. At each end, there are a boolean parameter, Limited, and a float number that sets the limit. The limit is effective only if

the boolean parameter, Limited, is true. Following queries concern the joint limits. When a **Point2** is returned, it is ordered as the lower limit followed by the upper limit. When a **Point3** is returned, the order is X, Y, and Z. For example, **TransLowerLimits()** returns lower limits of **TransX**, **TransY**, and **TransZ**, respectively.

### Preferred Angles

Some solver may start off the solution process with joint angles being set to special values, these are called preferred angles. **PrefPosition()** and **PrefRotation()** return preferred angles of the translation and rotation joints, respectively. The APIs allows them to be animatable. However, in 3ds max 4, they are constant with regard to animation time. The methods **SetPrefTrans()**, **SetPrefRot()**, and **SetPrefTR()** set their values at a specific time.

### Joint Angles

**TransValue()** and **RotValue()** will query joint angles of sliding and rotational joints, respectively, at a specific time. If the user supplies non-null pointer to a validity interval, the validity interval will be updated.

Their values can be assigned using **AssignTrans()** and **AssignRot()**. The methods **AssignActiveTrans()** and **AssignActiveRot()** will skip those degrees of freedom that are not active. These same two methods allow active DOFs to be given as the first argument of type **DofSet**, while having the new values supplied as float array whose size should be the same as the DofSet [**DofSet::Count()**]. The order in the array is: **DofZ** if the DofSet includes **DofZ**, **DofY** if the DofSet includes it, and followed by **DofX** if it is included in the **DofSet**. IK controllers are designed to be controlled by the IK goals (through the internal IK system). Therefore, their values are not to be set as independent variables are. These assignment methods just assign new values to the respective variables and that's all. They do not adjust the validity interval, this is done through the methods **SetTransValid()**, **SetRotValid()** and **SetTRValid()**. The last method sets the validity intervals for translation and rotation values to the same interval.

### FK Sub-controller

The Forward Kinematics sub-controller and the pointer to the node that holds this controller as its TM controller can be obtained using **Control\* FKSubController()** and **INode\* GetNode()**. Note that the IK controller is not designed to be instanced. It is expected to have a unique node.

## Interface Class **I IKChainControl**

### Reference Indices

The following enum constants are indices to the reference targets made by the **I IKChainControl**. For example, **GetReference(I IKChainControl::kPBlockRef)** will return a pointer to **IParamBlock2**. These reference indices are:

#### **kPBlockRef**

The IparamBlock2 parameter block.

#### **kGoalTMRef**

A Matrix3 controller representing the sub-controller that defines the transformation matrix of this node

#### **kEndJointRefINode**

The INode reference for the End Joint.

#### **kEnableRef**

Bool (float) controller. This switch indicates when the IK chain is enabled or effective.

#### **kStartJointRef**

The INode reference for the Start Joint.

#### **kLastRef**

Place holder.

### Parameter Block Indices

There is just one parameter block. Its index to **Animatable::GetParamBlock()** is given by enum constant **kParamBlock**.

Parameter Indices

The following enum constants represent the indices of the parameters in the parameter block:

**kStartJoint** INode, referenced by kStartJointRef

**kEndJoint** INode, referenced by kEndJointRef

**kSolverName** String, the solver name.

**kAutoEnable** BOOL, auto-enable.

**kSwivel** Angle, swivel angle.

**kPosThresh** Float, pos threshold.

**kRotThresh** Float, rot threshold.

**kIteration** Integer, iterations.

**kEEDisplay** BOOL, EE display.

**kEESize** Float, EE size.

**kGoalDisplay** BOOL, display goals.

**kGoalSize** Float, goal size.

**kVHDisplay** BOOL, VH display.

**kVHSize** Float, VH size.

**kVHLength** Float, VH length.

**kSolverDisplay** BOOL, display solvers.

**kAutoSnap** BOOL, auto snap.

**kVHUseTarget** BOOL, VH use target.

**kVHTarget** INode, VH target.

**kLastParam** Place holder.

Most parameters have separate querying methods. For example;

**GetParamBlock(IKChainControl::kParamBlock)>GetINode(IKC**

will return turns the same pointer as does **StartJoint()**. The **kStartJoing**

and **kEndJoint** parameters that define the IK Chain Delimiters can be

obtained by using the **INode\* StartJoint()** and **INode\* EndJoint()**

methods.

## Swivel Angles

Most of the time, joints are drawn on a plane (planar IK chains). A desirable quality of an IK solver is not to disturb this plane. Those solvers would use only the Start Joint to rotate the plane. When the End Joint is adhered to the goal, the rotation about the axis from the Start Joint to the End Joint is called Swivel. Swivel angle is the amount of this rotation.

In order for the swivel angle to make sense, however, we need a reference point. When a chain is put at the preferred pose, meaning that all the joint angles are set to preferred angles, the plane expanded by the pivots of all the joints on the chain is called *Initial Plane*, and the axis from the Start Joint to End Joint is called *Initial End Effector Axis* (EEAxis). Since Preferred Angles are animation variable, so is the *Initial Plane* and *EEAxis*. The **InitPlane()** and **InitEEAxis()** methods return the normal to the Initial Plane and the unit vector of the Initial EEAxis. These are represented in the parent space of the Start Joint. To obtain the normal in object space, use the method **ChainNormal()**. The relationship between **ChainNormal()** and **InitPlane()** is (in pseudo code):

**InitPlane() = ChainNormal() \* StartJoint()->PrefRotation().**

When the joints do not lie on a plane, the closest plane, in certain sense, can be used. Let's call this plane the *solver plane*.

The class **ZeroPlaneMap** is defined in **IKHierarchy.h**. This provides the functionality that, given a unit axis, which is to be substituted for by EE Axis, produces a unit vector, which will be interpreted as the normal to a plane. This plane will be taken as the zero plane to which the swivel angle is relative: the plane corresponds to swivel angle being zero. The amount of rotation that will bring the Zero Plane to the current solver plane is the Swivel Angle, which corresponds to the parameter of index **kSwivel** and can be obtained using the **SwivelAngle()** method. The following relationship holds (in pseudo code):

**ChainNormal() \* StartJoint()->Rotation() =  
ZeroPlaneMap(EEAxis) \* <Rotation about EEAxis by  
SwivelAngle(>**

A plugin solver may have its own ZeroPlaneMap. In case it does not have a preference, it can use the default one provided by the IK system as **IKSys::ZeroPlaneMap\* DefaultZeroPlaneMap()**.

The solver plane can be controlled by an extra node and this is often desired by the user. In this case, they want the solver plane to align with the plane defined by the Start Joint, the End Joint, and the extra node, called the *swivel angle target*. Since the target and the swivel angle are meant to control the same thing, only one of them can be used. The parameter of index **kVHUseTarget** determines which will be used. It is not animatable. If it is true, the swivel angle target will be used. The target node can be obtained from the parameter of index **kVHTarget**. When the target node is NULL, the swivel angle will be used.

### Solver Properties

The method **IKSolver\* Solver()** returns the IK solver whose responsibility it is to solve the IK problem. Its class name is stored as the parameter of index **kSolverName**. As mentioned, there is an animatable variable that determines, at a specific time, whether IK is enabled. Whether the solver is enabled or disabled can be checked using the **SolverEnabled()** method.

This is just an interface to the sub-controller with reference index **kEnableRef**. Since its value is used to determine whether joints in an IK chain depend on the IK goal when reference message is received, developers should not replace it, using the 3ds max Reference APIs, with an arbitrary controller whose dependency structure is general. This would create a time variable dependency structure and is not desirable. The built in controller is a key-frame controller.

Sometimes, we still want to invoke IK by moving the goal even if **SolverEnabled()** is false. In such cases the purpose of IK is to set keys on the FK sub-controllers of the joints. The feature is called "Use IK to FK", which is only meaningful during an interactive session. For this purpose **CanAutoEnabled()** corresponds to the parameter of index **kAutoEnable** and indicates whether the feature of "Use IK to FK" is desired. The **AutoEnableSet()** method provides the transient state of the IK chain: whether IK is currently enabled due to this feature. It is only set, if it is ever set, during the next update cycle, with regard to when the goal is moved, or its TM is set to new value.

### Valid IK Chains

A valid IK chain is one that has a good Solver, Start and End joints, where the

Start Joint is an ancestor of the End Joint in the scene hierarchy. The **Valid()** and **INode\* GetNode()** methods are available for this. The latter returns the node that holds it as its TM controller. The IK Chain Controller is not designed to be instanced. It is expected to have a unique node.

## **Class IKSolver and Solver plugins**

Being a plugin itself, the IK system allows the solver to be a separate plugin. This class defines the base class that plugin solver should derive from. The IK solver is a pure mathematical function: it does not hold state and just solves a given, self-contained, mathematical problem. In other words, the plugin solver does not have influence on when IK is invoked and what an IK problem is (what is the goal and what are the joints, etc.), but contributes to IK by answering how to solve. Structurally, it is independent of the SDK and, hence, can be built independently, except for some theoretically independent math library.

### **Class Identity**

Plugin solvers are recognized by the IK system by the Super Class ID. A unique enum, **IK\_SOLVER\_CLASS\_ID**, defined in **animtbl.h**, is given to it. The **SClass\_ID SuperClassID()** method should not to be overridden. Since it is a pure mathematical function that does not hold state, an individual plugin solver is identified by its class name which can be obtained using the **GetClassName()** method. When class name clashes, a suffix may be appended. The class name will appear in the solver list from that users can pick for or assign to IK chains.

### **Solver Traits**

Following methods are meant to be overridden by the plugin solver queried by the IK system.

**IsInteractive()**. IK can be used as a controller or as an interactive manipulation tool. In the former, the relationship between the goal and the joints are permanent: joints are completely controlled by the goal. In the latter, the relationship is transient, existing only during interactive manipulation. In the end, IK solutions are registered at each joint, mostly likely as key-frames, and it no longer matters how joints got their joint angles. Only non-interactive, or controller, IK solvers are used.

**IsHistoryDependent()**. At a specific animation time, the history dependent solver will reach solutions not only based the state of goal at the time, but also its previous states (hence they are history dependent). On the contrary, the history independent solver does its job based on the state of the goal just at the time. The procedural implication is that, when the goal is changed at time  $t$ , the IK system would have to invalidate joints, at time  $t$  for the history independent solver, and at all times that are greater or equal to  $t$  for the history dependent solver. Only history dependent solvers are used by the IK system.

The methods **UseSlidingJoint()** and **UseSwivelAngle()** are used to tell whether the plugin solver intends to use the sliding joint (translational degrees of freedom) or the swivel angle parameter of the IK chain.

When two IK chains overlap, i.e., if there's a joint belonging to both IK chains, some solvers are able to negotiate between the possibly contending goals and some are not. The method **DoesOneChainOnly()** is used to determine this. For those that can only solve one chain at a time, the IK system will pass to the solvers one chain at a time in a definitive order. Only solvers that "do one chain only" are used.

The method **DoesRootJointLimits()** and **DoesJointLimitsButRoot()** deal with the concern of joint limits. If the solver supports joint limits, the IK system will trust it. Otherwise, the IK system will, after calling the solver, clamp the results according the joint limit constraints. The root joint in **DoesRootJointLimits()** refers to the Start Joint. It is treated differently from the rest of joints.

The method **SolveEERotation()** tells whether the rotation part of the goal node will be used. If it returns false, only the position of the goal node is taken as the IK goal and rotation threshold will be irrelevant.

Solvers can reach solutions with closed formula, analytically, or going through iterations. For an analytic solver, thresholds and maximum iteration numbers are not relevant. Checking whether or not a solver is analytical can be done using the **IsAnalytic()** method.

### **Solution Parameters**

The methods **GetPosThreshold()**, **GetRotThreshold()**, **GetMaxIteration()**, **SetPosThreshold()**, **SetRotThreshold()**, **SetMaxIteration()** are used to get and set the Position Threshold, Rotation

Threshold, and the Maximum number allowed for iterations and are not relevant for all solvers. For an analytic solver, for example, these are not used at all. The IK system, however, may set values to them.

As mentioned earlier, plugin solvers may have their own Zero Plane Map. If so, they must override the **IKSys::ZeroPlaneMap\***

**GetZeroPlaneMap()** method. The IK system will need it to perform IK snapping: setting the swivel angle based on the current pose so that the pose is consistent with the swivel angle.

## Solve

**Solve()** is the method which the IK system will call when it's time to update the joints according to the IK goal and other parameters. The IK system will compile an IK problem represented in the 3ds max scene data structure, nodes, controllers, etc., into a pure mathematical representation,

**IKSys::LinkChain**. This data structure only represents one IK chain.

Overlapping IK chains are not dealt with. The **ReturnCondition**, returned by the **Solve()** method is a bit-set. Its definition is copied from

**IKHierarchy.h**:

```
typedef unsigned ReturnCondition;
enum ConditionBit {
    bLimitReached = 0x00000001,
    bLimitClamped = 0x00000002,
    bMaxIterationReached = 0x00000004,
    // The first eight bits are reserved for mild condition.
    // They are still considered successful.
    bGoalTooCloseToEE = 0x00000100,
    bInvalidArgument = 0x00000200,
    bInvalidInitialValue = 0x00000400
};
```

The data structure passed to the solver is transient, meaning that it will be discarded once the solution is copied back to the joints. If the return condition indicates failure, (**return\_condition > 0xff**) then the result will not be

copied back to the joint nodes in the 3ds max scene database.

## **Class LinkChain**

A LinkChain is a hierarchy of transformations. At the top is the **RootLink** and it is followed by a number of Link's . Each link, be it a **RootLink** or **Link**, evaluates to a matrix, called the link matrix (**LinkMatrix**). This matrix consists of two parts, one is a variable part (degrees of freedom) and the other is a relatively constant part, called rigid extend (**RigidExtend**). It can be graphically represented as:

### **RootLink**

The link variable is a rotation, **rotXYZ**. The link matrix is defined as  $\text{RigidExtend} * R_x(\text{rotXYZ}.x) * R_y(\text{rotXYZ}.y) * R_z(\text{rotXYZ}.z)$  where  $R_x(.)$  stands for rotation about the x-axis, etc.

### **Link**

The variable part of a Link is of one degree of freedom. It can be translational or rotational. A typical 3 degrees of freedom (xyz) rotational joint can be decomposed into 3 Links, with the first two being null links:

### **LinkChain**

A **LinkChain** consists of a **RootLink** and a number of Link's. The reason that the **LinkChain** can start with rotation, the variable part of the **RootLink**, is that the IK system will collapse all the node transformations from the root of the scene to the parent matrix of the Start Joint and the translation part of the Start Joint, called the parent matrix, and transform the node hierarchy of the IK chain into this space. The author of the plugin solver does not have to be concerned about **parentMatrix**. They work on the problem as it is rooted at the origin of the world. The data structure of **LinkChain** should be sufficient for those solvers that **DoesOneChainOnly()** and **UseSwivelAngle()**.

## IK System

The IK system services two interfaces, one through the CORE, and the other through the IK Chain Controller class.

### Interface Class [IKCmdOps](#)

The interface ID is defined in **IIKSys.h** as **IK\_FP\_INTERFACE\_ID** and is published through the CORE, i.e., to get a pointer to this interface, call **GetCOREInterface(IK\_FP\_INTERFACE\_ID)**.

To create a new IK chain, use the method **CreateIKChain()**. Given two pointers, **start** and **end**, that are intended for the Start and End joints respectively, and the solver's name, **solver**, this method will create a node that has an **IKChainControl** as the TM controller. This method will return false and fail if: 1) **start** joint is not a proper ancestor of **end** joint in the scene graph; or 2) there is a node along the path from the start joint to the end joint whose TM controller is not an IK Control and is not replaceable, or 3) no IK solver is found with the given name.

### Interface Class [IKChainActions](#)

The interface ID is defined in **IIKsys.h** as **IKCHAIN\_FP\_INTERFACE\_ID** and is published through the class **IKChainControl**, i.e., to get a pointer to this interface, call;

**GET\_IKCHAIN\_CD-  
>GetInterface(IKCHAIN\_FP\_INTERFACE\_ID)**

where **GET\_IKCHAIN\_CD** is a macro defined in the same file that produces a pointer to the class descriptor of class **IKChainControl**.

This interface also publishes a number of actions. An action acts on the currently selected IK chain node. The following method returns TRUE if there is exactly one valid IK chain node selected: **IsSnapEnabled()**. The following methods execute snapping on the currently selected IK chain node: **SnapAction()**, **IKSnapAction()**, and **FKSnapAction()**. The first one does IK snapping if its Enabled is false and FK snapping if its Enabled is true. The next two will do IK snapping and FK snapping, respectively.

# Keyboard Accelerators and Dialog Messages

See Also: [Class Interface](#), [Custom Controls](#).

## Keyboard Shortcuts

**Important Note:** A new system to handle keyboard accelerators was added for R4. This system supercedes those used in previous release. For information on this system see [Class ActionTable](#). The information shown below applies to previous version of the SDK APIs.

**Important Note:** In release 3.0 a new system to handle keyboard shortcuts was established. This system is used to register shortcuts in a uniform manner. In the Customize / Preference Settings / Keyboard tab dialog there is a section to assign Plug-In shortcuts to commands. For details on how this works see the topic [Keyboard Shortcut System](#).

## Keyboard Accelerators

Developers might want to implement keyboard accelerators for some of their plug-ins functions. Because 3ds max plug-ins are modeless, when using R1 there was inherently a conflict between the accelerator keys that the user expected to run standard 3ds max functions, and those supplied by the plug-in. For this reason, a developer had to limit their accelerator key usage as much as possible. However in R2 and later the 'Plug-In Keyboard Shortcut Toggle' icon was added. This controls this conflict such that the user can choose if the plug-in accelerators are in effect or if the system accelerators are. Thus in R2 and later developer are free to registers all the accelerators they need and the user can control which are available.

When a plug-in has its parameters up and registers some accelerators, these accelerators take precedence over 3ds max when the 'Plug-In Keyboard Shortcut Toggle' is on. The methods below (from Class **Interface**) enable the developer to do this.

**virtual void RegisterAccelTable( HWND hWnd, HACCEL hAccel )=0;**

This method registers a keyboard accelerator table. Window messages generated by the accelerator table are sent to the **hWnd** parameter.

**virtual int UnRegisterAccelTable( HWND hWnd, HACCEL hAccel )=0;**

This method un-registers a keyboard accelerator table.

## Receiving Input to Edit Controls

If a plug-in wants to receive keyboard input using a Windows edit control, it must disable the keyboard accelerators. 3ds max uses un-modified keys (such as 'f' to change to the front view). Unless a plug-in disabled the accelerators it would not get this input. The following global functions are used to enable, disable and check the enabled state of the keyboard accelerators.

### **void DisableAccelerators();**

When this method is called, ALL keyboard accelerators are disabled and are no longer processed.

### **void EnableAccelerators();**

When this method is called, the keyboard accelerators are processed again.

### **BOOL AcceleratorsEnabled();**

Determines if the keyboard accelerators are enabled. Returns TRUE if enabled; FALSE if disabled.

For modal dialogs, Windows itself essentially handles disabling and enabling accelerators automatically. For modeless dialogs, a plug-in may have to deal with these functions. If a plug-in uses all standard 3ds max custom edit controls for its editing, it won't need to deal with enabling and disabling accelerators as this is handled by the custom controls. On the other hand, if a plug-in uses any Windows edit controls, such as a combo box that has an edit field and drop down, they will have to call these functions.

For example, in a combo box, when an edit field gets focus, the Windows message CBN\_SETFOCUS is sent. At this time, a plug-in should call **DisableAccelerators()**. If the plug-in did not do this, and if the user typed an 'f', the current 3ds max viewport would change to the front view. By disabling all the accelerators, the plug-in will receive the characters itself. When the dialog proc receives CBN\_KILLFOCUS, the accelerators may be enabled using **EnableAccelerators()**.

## Floating Dialogs

If a plug-in creates a floating dialog box there are two important things that should be done.

1) A plug-in must use the 3ds max window handle as the parent passed to the "DialogBox" of "DialogBoxParam". One can get this handle using the method **Interface::GetMAXHwnd()**.

2) The method **Interface::RegisterDlgWnd(HWND hDlg)** must be called passing the handle of the dialog window.

Following these steps will prevent problems with the user of the dialog being able to operate the rest of 3ds max while your dialog is active.

## Keyframe and Procedural Controller Data Access

See Also: [Class IKeyControl](#), [Class INode](#), [Class Control](#), [Class Matrix3](#), [Class AngAxis](#), [Class Quat](#), [Class ScaleValue](#), [Matrix Representations of 3D Transformations](#).

## Overview

This topic presents information on accessing the data of MAX's keyframe and procedural controllers. There are sections which explain how the keyframe data for the PRS controller can be interpreted, various APIs for keyframe data access, and procedural controller data access.

Example keyframe controllers are the Tension/Continuity/Bias (TCB), Bezier, and Linear controllers. Each keyframe controller in 3ds max is used to animate a value. The keys of the controller store the values at the key frames. The controller's job is to interpolate between these keys. This topic discusses the way developers may access and interpret the keys stored by keyframe controllers (in particular the keys stored by the Position/Rotation/Scale transform controllers). 3ds max also allows the use of procedural controllers. Procedural controllers are those that compute their value algorithmically, based on user specified parameters or outside data, rather than interpolating between stored keyframes. Examples include the Noise Controller, Expression Controller, Audio Controller, Path Controller, and Waveform Controller. This topic presents information on how procedural controllers may be sampled to retrieve their data.

Source code is provided demonstrating how developers may access both keyframe and procedural controller data.

## Interpretation of Keyframe Key Data for the PRS Controller

The default keyframe transform controller in 3ds max is a PRS controller. This means that it handles orienting a node in the scene using separate controllers for Position, Rotation and Scale. To generate a complete transformation, first the position is applied, then the rotation, then the scaling.

Position and scale keys are stored in the local coordinate system of the controller, with each key independent of the others. Rotation keys are stored in local coordinates as well, but each is stored relative to the previous key in the track. Thus the first key stored represents the actual value for the rotation key in terms of the relative space of the controller. The second key is just an offset from the first key. Thus to get the actual rotation for the second key you need to multiply it on the left by the first key. To get the third key, you need to multiply it by the second computed key (which was computed from the first). These computed rotation key values will then be the absolute quaternions for each key for that controller.

It is important to understand however that the entire animation track for position, rotation and scale are all relative to the matrix passed into the controller when its **GetValue()** method is called. For additional details on how a controller updates **GetValue()** see the Advanced Topics section [Node and Object Offset Transformations](#).

Note the following:

- When developers access key data from MAX, the data is returned in the local coordinate system of the controller.
- The key values that appear in the Key Info dialog for a controller are in the local coordinate system of the controller (with the appropriate units for display -- for example, angles are shown in degrees in Key Info but are stored in radians).

## Keyframe Access Classes and Methods

This section discusses access to keyframe controller data. The 3ds max API provides a class **IKeyControl**. This class provides an interface into the TCB, Bezier, and Linear keyframe controllers allowing a developer to add, delete, retrieve and update the keys of the controller.

Use of the **IKeyControl** methods requires you to include `\MAXSDK\INCLUDE\ISTDPLUG.H`, i.e.:

```
#include "ISTDPLUG.H"
```

The standard 3ds max PRS keyframe controllers provide access to their keys using this class. To get an interface you can use to call methods of **IKeyControl** use the following macro (defined in `\MAXSDK\INCLUDE\ANIMTBL.H`):

```
#define GetKeyControlInterface(anim)  
((IKeyControl*)anim->GetInterface(I_KEYCONTROL))
```

A plug-in developer may use this macro as follows:

```
IKeyControl *ikc = GetKeyControlInterface(anim);
```

The return value will either be NULL or a pointer to a valid controller interface. A NULL pointer indicates the controller does not support this interface to allow access to its data.

The following is an example of getting a position controller interface from a node in the scene. The first thing that happens is the position controller is retrieved from the node (using methods of class **Inode** and **Control**) and then the controller interface is returned via the **GetKeyControlInterface()** macro:

```
Control *c;  
c = node->GetTMController()->GetPositionController();  
IKeyControl *ikeys = GetKeyControlInterface(c);  
if (!ikeys) return; // No interface available to access the keys...
```

The methods of **IKeyControl** may be called using the **ikeys** pointer. For example the method **GetKey(int i,IKey \*key)** retrieves the 'i-th' key and stores the result in **key**. You'll need to check the ClassID of the controller so you can pass the appropriate class to the method to retrieve the keys. There are three type of controllers and five data types that are supported by the **IKeyControl** interface. The controller types are Tension/Continuity/Bias (TCB), Bezier, and Linear. The data types are floating point (**float**), Position (**Point3**), Rotation

(**Quat** or **AngAxis**), and Scale (**ScaleValue**). The type of key you pass to the **GetKey()** or **SetKey()** methods depends on both the controller and data type. The following classes are used for key storage for each valid possibility:

**Tension/Continuity/Bias:**

[Class ITCBFloatKey](#)

[Class ITCBPoint3Key,](#)

[Class ITCBRotKey](#)

[Class ITCBScaleKey](#)

**Bezier:**

[Class IBezFloatKey](#)

[Class IBezPoint3Key](#)

[Class IBezQuatKey](#)

[Class IBezScaleKey](#)

**Linear:**

[Class ILinFloatKey](#)  
[Class ILinPoint3Key](#)  
[Class ILinRotKey](#)  
[Class ILinScaleKey](#)

See the following code for an example:

```
ITCBPoint3Key tcbPosKey;  
int numKeys = ikeys->GetNumKeys();  
if (c->ClassID() ==  
Class_ID(TCBINTERP_POSITION_CLASS_ID, 0)) {  
  for (i = 0; i < numKeys; i++) {  
    ikeys->GetKey(i, &tcbPosKey);  
    DebugPrint(_T("\nPosition Key: %d=(%.1f, %.1f, %.1f)",  
      i, tcbPosKey.val.x, tcbPosKey.val.y, tcbPosKey.val.z);  
  }  
}
```

Note how an instance of the **ITCBPoint3Key** class was passed to **GetKey()** since the ClassID indicates the controller is a TCB controller and we want position keys (which operate on **Point3s**).

## Keyframe Interpolation in MAX

The function used for interpolation between keys depends on the type of controller. In general, 3ds max uses cubic (polynomial of degree three) splines for interpolation. Rotations are done using spherical linear interpolation (slerps). Registered developers who are interested in the exact details of keyframe interpolation have the source code 3ds max uses internally as part of the Debug SDK. The bezier interpolation code is available in

`\MAXSDKDB\SDKSRC\INTERP.CPP` and the TCB interpolation is in `\MAXSDKDB\SDKSRC\TCBINTRP.CPP`. This code is not available for non-registered developers. See the Advanced Topics section on [Debugging](#) for more details on the Debug SDK.

Several developers have tried to use the **IKeyControl** interface and then operate on the keys to get and interpolate them in 'world space'. This is really a problematic thing to try do because the keys are interdependent. For instance, consider scale keys. In world space scale keys are dependent on the position and rotation of the controller as well as the parent transformation. This is because first an object is scaled, then it is rotated, then its position is applied, then the parent transformation is applied. So the scale is rotated. If you scale an object along its X axis and then rotate it the scale becomes rotated. The scale is no longer rotated about the world X axis, it's rotated about some other axis. This is normally what you want, i.e. you don't want the object to rotate through the scaled space (this would cause it to skew). So to talk about the scale in world space is a strange concept.

Further, the interpolation of keys that are in world space is equally problematic. If keys are put into world space, then interpolated in world space, the resulting animation will be very different that what would be seen inside MAX. For example, consider an object that is rotating 360 degrees about its local Z axis and is tilted at some arbitrary angle. If you interpolate the way 3ds max does, all the interpolation is done in the local coordinate system. Thus the object would rotate about its local Z axis (which is tilted). However if you were to try to interpolate about in world space the rotation would occur about the world Z axis. The object will still end up in the same place, but it will be a very different animation than what 3ds max would have.

The bottom line is that developer need to understand the way 3ds max computes its node transformation based on the keyframes and use the information to interpolate between keys accordingly.

## Procedural Controller Data Access

3ds max allows users to use procedural controllers in addition to keyframe controllers. Procedural controllers are those that compute their value algorithmically rather than interpolate between stored keyframes. Developers may need to retrieve values from these controllers but won't be able to use the **IKeyControl** interface since no keys are available. In these cases, one must sample the controller to retrieve its value at each frame required. This is done by calling the **Control::GetValue()** method directly. The following sample code samples the controller for each frame in the current animation interval and **DebugPrint()**s the values to the VC++ IDE debug window. Note how the parent matrix of the node is passed into the **GetValue()** call. This allows **GetValue()** to update the appropriate matrix. In the code below, the position coordinates printed by **DebugPrint()** will be in world space.

```
void Utility::SampleController(INode *n, Control *c) {
    TimeValue t;
    Point3 trans;
    Matrix3 pmat;
    Interval ivalid;
    int tpf = GetTicksPerFrame();
    int s = ip->GetAnimRange().Start()/tpf,
        e = ip->GetAnimRange().End()/tpf;

    // Sample the controller at every frame in the anim range
    for (int f = s; f <= e; f++) {
        t = f*tpf;
        ivalid = FOREVER;
        pmat = n->GetParentTM(t);
        c->GetValue(t, &pmat, ivalid, CTRL_RELATIVE);
        trans = pmat.GetTrans();
        DebugPrint(_T("\nPosition at frame: %d of %d=(%.1f, %.1f, %.1f)"),
            f, e, trans.x, trans.y, trans.z);
    }
}
```

A developer could also get the relative values from the PRS controller by passing in the identity matrix and not the parent matrix to **GetValue()**. This is appropriate for the PRS transform controller but not the Look At transform

controller. With a Look At controller, its relative values are actually a function of the input matrix. If you pass in the identity to its **GetValue()** method, it will think that the object is positioned at the center of the world and the results won't be meaningful.

## Sample Code

The following sample code demonstrates access to the controller data for the first node in the current selection set. The code checks to see if the controller provides its data via the **IKeyControl** interface. If it does, this interface is used to get the keys. If it does not, the controller is sampled at each frame to get its data. For the keyframe controllers, note how the ClassIDs are checked to ensure the proper classes are used when getting key values. Also note how the rotation keys are derived by multiplying on the left by the previous key. For the procedural controllers, note how the parent matrix of the node is passed to the **GetValue()** method.

```
// Display the data of the controller. For TCB, Bezier and Linear  
// keyframe PRS controllers the position, rotation and scale values  
// are displayed in the local space of the controller. For  
// procedural controllers (or those that don't support the  
// IKeyControl interface), position data is displayed in world space.  
void Utility::KeyTest() {  
    int i, numKeys;  
    INode *n;  
    Control *c;  
    Quat newQuat, prevQuat;  
    IKeyControl *ikeys;  
    ITCBPoint3Key tcbPosKey;  
    ITCBRotKey tcbRotKey;  
    ITCBScaleKey tcbScaleKey;  
    IBezPoint3Key bezPosKey;  
    IBezQuatKey bezRotKey;  
    IBezScaleKey bezScaleKey;  
    ILinPoint3Key linPosKey;  
    ILinRotKey linRotKey;  
    ILinScaleKey linScaleKey;  
  
    // Get the first node in the selection set  
    if (!ip->GetSelNodeCount()) return;  
    n = ip->GetSelNode(0);  
  
    // --- Process the position keys ---
```

```

c = n->GetTMController()->GetPositionController();
ikeys = GetKeyControlInterface(c);
if (!ikeys) {
    // No interface available to access the keys...
    // Just sample the controller to get the position
    // data at each key...
    SampleController(n, c);
    return;
}
numKeys = ikeys->GetNumKeys();
DebugPrint(_T("\nThere are %d position key(s)"), numKeys);

if (c->ClassID() == Class_ID(TCBINTERP_POSITION_CLASS_ID, 0))
{
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &tcbPosKey);
        DebugPrint(_T("\nTCB Position Key: %d=(%.1f, %.1f, %.1f)"),
            i, tcbPosKey.val.x, tcbPosKey.val.y, tcbPosKey.val.z);
    }
}
else if (c->ClassID() ==
Class_ID(HYBRIDINTERP_POSITION_CLASS_ID, 0)) {
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &bezPosKey);
        DebugPrint(_T("\nBezier Position Key: %d=(%.1f, %.1f, %.1f)"),
            i, bezPosKey.val.x, bezPosKey.val.y, bezPosKey.val.z);
    }
}
else if (c->ClassID() == Class_ID(LININTERP_POSITION_CLASS_ID,
0)) {
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &linPosKey);
        DebugPrint(_T("\nLinear Position Key: %d=(%.1f, %.1f, %.1f)"),
            i, linPosKey.val.x, linPosKey.val.y, linPosKey.val.z);
    }
}

```

```

// --- Process the rotation keys ---
c = n->GetTMController()->GetRotationController();
ikeys = GetKeyControlInterface(c);
if (!ikeys) return;

numKeys = ikeys->GetNumKeys();
DebugPrint(_T("\nThere are %d rotation key(s)"), numKeys);

if (c->ClassID() == Class_ID(TCBINTERP_ROTATION_CLASS_ID,
0)) {
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &tcbRotKey);
        newQuat = QFromAngAxis(tcbRotKey.val.angle, tcbRotKey.val.axis);
        if (i) newQuat = prevQuat * newQuat;
        prevQuat = newQuat;
        DebugPrint(_T("\nTCB Rotation Key: %d=(%.1f, %.1f, %.1f,
%.1f)"),
            i, newQuat.x, newQuat.y, newQuat.z, newQuat.w);
    }
}
else if (c->ClassID() ==
Class_ID(HYBRIDINTERP_ROTATION_CLASS_ID, 0)) {
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &bezRotKey);
        newQuat = bezRotKey.val;
        if (i) newQuat = prevQuat * newQuat;
        prevQuat = newQuat;
        DebugPrint(_T("\nBezier Rotation Key: %d=(%.1f, %.1f, %.1f,
%.1f)"),
            i, newQuat.x, newQuat.y, newQuat.z, newQuat.w);
    }
}
else if (c->ClassID() ==
Class_ID(LININTERP_ROTATION_CLASS_ID, 0)) {
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &linRotKey);
        newQuat = linRotKey.val;
    }
}

```

```

    if (i) newQuat = prevQuat * newQuat;
    prevQuat = newQuat;
    DebugPrint(_T("\nLinear Rotation Key: %d=(%.1f, %.1f, %.1f,
%.1f)"),
    i, newQuat.x, newQuat.y, newQuat.z, newQuat.w);
}
}

// --- Process the scale keys ---
c = n->GetTMController()->GetScaleController();
ikeys = GetKeyControlInterface(c);
if (!ikeys) return;

numKeys = ikeys->GetNumKeys();
DebugPrint(_T("\nThere are %d scale key(s)"), numKeys);

if (c->ClassID() == Class_ID(TCBINTERP_SCALE_CLASS_ID, 0)) {
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &tcbScaleKey);
        DebugPrint(_T("\nTCB Scale Key: %2d=(%.1f, %.1f, %.1f)"),
        i, tcbScaleKey.val.s.x, tcbScaleKey.val.s.y,
        tcbScaleKey.val.s.z);
    }
}
else if (c->ClassID() ==
Class_ID(HYBRIDINTERP_SCALE_CLASS_ID, 0)) {
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &bezScaleKey);
        DebugPrint(_T("\nBezier Scale Key: %2d=(%.1f, %.1f, %.1f)"),
        i, bezScaleKey.val.s.x, bezScaleKey.val.s.y,
        bezScaleKey.val.s.z);
    }
}
else if (c->ClassID() == Class_ID(LININTERP_SCALE_CLASS_ID, 0))
{
    for (i = 0; i < numKeys; i++) {
        ikeys->GetKey(i, &linScaleKey);

```

```

    DebugPrint(_T("\nLinear Scale Key: %2d=(%.1f, %.1f, %.1f)",
    i, linScaleKey.val.s.x, linScaleKey.val.s.y,
    linScaleKey.val.s.z);
}
}
}

// Display the position data of controller in world coordinates for each
// frame in the animation range
void Utility::SampleController(INode *n, Control *c) {
    TimeValue t;
    Point3 trans;
    Matrix3 pmat;
    Interval ivalid;
    int tpf = GetTicksPerFrame();
    int s = ip->GetAnimRange().Start()/tpf,
        e = ip->GetAnimRange().End()/tpf;

    // Sample the controller at every frame in the anim range
    for (int f = s; f <= e; f++) {
        t = f*tpf;
        ivalid = FOREVER;
        pmat = n->GetParentTM(t);
        c->GetValue(t, &pmat, ivalid, CTRL_RELATIVE);
        trans = pmat.GetTrans();
        DebugPrint(_T("\nPosition at frame: %d of %d=(%.1f, %.1f, %.1f)",
        f, e, trans.x, trans.y, trans.z);
    }
}
}

```

## Summary

This section provided an overview of accessing 3ds max controller data. Since 3ds max supports both keyframe and procedural controllers different techniques are required to retrieve the data. Keyframe controller can usually use the **IKeyControl** interface. Procedural controller must be sampled using **GetValue()**, usually at each frame. The values that a controller stores are in the local coordinate system of the controller. The matrix passed to **GetValue()** for a transform controller defines the coordinate system of the transformation.

## Loading and Saving Plug-In Data

See Also: [Class ReferenceMaker](#), [Class ILoad](#), [Class ISave](#), [Class AppSave](#), [Class AppLoad](#).

## Overview

Plug-in developers need to be aware of the way 3ds max saves and loads a plug-in's data from disk. In certain cases, the plug-in's data will be saved automatically without the plug-in developer having to provide any special code. In other cases, the plug-in must save and load the data it maintains for its operation.

The following two cases are examples of 3ds max automatically saving plug-in data.

- 1 3ds max takes care of loading and saving any references the plug-in has when it is saved to disk. The plug-in does not need to explicitly save its references, nor does it need to load its references. After a scene is loaded, the references will automatically be restored. See the Advanced Topics section on [References](#) for more details on the loading and saving of references.
- 2 If a plug-in's parameters are all handled by parameter blocks, the plug-in does not need to save its parameters because the parameter block is responsible for loading and saving to disk.

If the plug-in maintains any other type of data it needs for its operation, it **must** specifically save this data to disk.

There are two methods of the **ReferenceMaker** class which are called by the system when the plug-in's data needs to be loaded or saved. These methods are named **Load()** and **Save()**. The plug-in implements these methods to load or save any special data it has.

The process works as follows: When a 3ds max file is saved, each plug-in object is asked to save its data. It is provided with an interface which has methods to write data. This data can be partitioned into chunks. A chunk is simply a container used to organize the data in the file. When an object needs to be saved, 3ds max creates a chunk for the object. Inside that chunk, another chunk is created to contain reference information which the system writes. The plug-in's **Save()** method is then called. When the plug-in begins execution in its **Save()** procedure, the file pointer is positioned inside the plug-in's chunk but after the first chunk containing the reference data. The plug in can then create its own chunks. Each chunk may contain sub-chunks or data, but not both. In other words, any single chunk may contain either data only, or other sub-chunks only. If you want to put both data and chunks inside another chunk, the data needs to be bracketed inside a chunk itself. To save data, a **Write()** method is used that

simply writes a block of bytes to the output stream. See the sample program below for an example using chunks and data to save information.

Loading a 3ds max file is similar. The plug-in is asked to load its data and is given an interface pointer which allows it to read chunks. When a plug-in DLL is asked to create a new instance of its class (using the **Create()** method of the plug-ins class descriptor), a parameter is passed to indicate whether the new instance is being created with the intention of loading an object from a disk file.

In this way, if the parameter is TRUE, the object doesn't have to perform any initialization because it is guaranteed that it will be asked to read data from a file. The parameter referred to here is the loading flag passed into

**ClassDesc::Create(BOOL loading).**

## Sample Code

The following code demonstrates how both **Save()** and **Load()** may be implemented for a plug-in with two pieces of data it needs to save and restore. This sample writes an array of 10 floating point values and a single DWORD containing flags. The process of saving is very simple. The output file is already open and ready to be written to. A pointer to the [ISave](#) class has been passed in with which the plug-in can call methods to save data. The plug-in begins by creating a chunk using the **BeginChunk()** method and passing an ID it has defined. This ID can be any **USHORT**, as only the plug-in's loading and saving procedures will ever use it. It then writes the data using the **Write()** method. It then closes the chunk using **EndChunk()**. It begins a new chunk and writes the flag data. After ending this chunk it returns **IO\_OK** to indicate a successful save.

The data is saved in chunks like this so that it may be loaded in a manner which is not order dependent. One could write the flags first and the array second and it would still be read correctly.

```
#define SAMPLE_DATA_CHUNK 1000
#define SAMPLE_FLAGS_CHUNK 1010
IOResult Sample::Save(ISave* isave) {
    ULONG nb;
    isave->BeginChunk(SAMPLE_DATA_CHUNK);
    isave->Write(myArray, sizeof(float) * 10, &nb);
    isave->EndChunk();
    isave->BeginChunk(SAMPLE_FLAGS_CHUNK);
    isave->Write(&flags, sizeof(DWORD), &nb);
    isave->EndChunk();
    return IO_OK;
}
```

The following code demonstrates the loading process. Again, in preparation for this call the system has opened the file and positioned the file pointer at the plug-in's objects chunk. The code loops, opening chunks and reading the data until **OpenChunk()** no longer returns **IO\_OK**. This indicates there are no more chunk at this level and the process is done.

Inside the loop, the code switches on the chunk ID. For each ID it recognizes it

reads the data using **Read()**. The chunk is then closed using **CloseChunk()**. If **Read()** did not return **IO\_OK** an error occurred and this error code is returned. Otherwise, the loop begins again.

If the loading was successful, **IO\_OK** is returned to indicate so.

```
IOResult Sample::Load(ILoad* iload) {  
    ULONG nb;  
    IOResult res;  
    while (IO_OK==(res=iload->OpenChunk())) {  
        switch(iload->CurChunkID()) {  
            case SAMPLE_DATA_CHUNK:  
                res=iload->Read(myArray, sizeof(float)*10, &nb);  
                break;  
            case SAMPLE_FLAGS_CHUNK:  
                res=iload->Read(&flags, sizeof(DWORD), &nb);  
                break;  
        }  
        iload->CloseChunk();  
        if (res!=IO_OK)  
            return res;  
        }  
    return IO_OK;  
}
```

The possible return values for **IOResult** are:

**IO\_OK** - The result was acceptable - no errors.

**IO\_END** - This is returned from **OpenChunk()** when the end of the chunks at a certain level have been reached. It is used as a signal to terminate the processing of chunks at that level.

**IO\_ERROR** - This is returned if an error occurred. Note that the plug-in should **not** put up a message box if a read error occurred. It should simply return the error status. This prevents a overabundance of messages from appearing.

For Reference information see: [ILoad](#), [ISave](#).

## **Loading and Saving Hierarchical Data**

There are two classes available for writing and reading hierarchical data structures to a linear stream, such as an AppData block. These are [Class AppSave](#) and [Class AppLoad](#). Please see those classes for more details.

# Matrix Representations of 3D Transformations

See Also: [Class Matrix3](#), [Class Inode](#), [Class Quat](#), [Class AngAxis](#), [Structure AffineParts](#), [Class BigMatrix](#).

## Overview

This section discusses the way transformation matrices are constructed and used in MAX, and various APIs that are available for working with these matrices.

Transformation matrices can be used, for example, to transfer the coordinates of an object from local space to world space, or to position nodes in the scene. The transformations these matrices provide are translation, scaling and rotation.

Methods of the **Matrix3** class (and several global functions) are provided to easily create transformation matrices and perform matrix arithmetic.

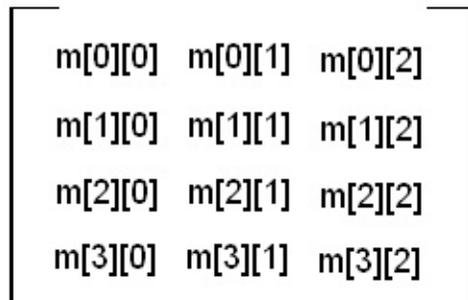
## Matrix Fundamentals

A matrix is a two-dimensional array of numbers. In MAX, 4x3 matrices are used. For 3ds max matrices, the first number is the number of rows (4) and the second number is the number of columns (3). Thus there are a total of 12 elements.

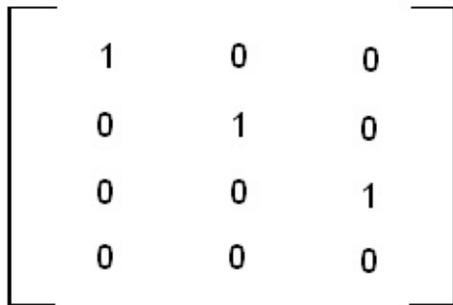
An instance of **Matrix3** has a private data member that contains the values:

```
float m[4][3];
```

The layout of these elements is shown in the following diagram:



The following matrix is called the 'Identity Matrix'. This special matrix, when multiplied by a vector or another matrix, yields the original matrix.



The 3ds max API provides several ways to create identity matrices. If a value of 1 is passed to the constructor of a **Matrix3** it is initialized to the identity. For example

```
Matrix3 tmat(1); // Identity matrix
```

You may also use a method of **Matrix3** to reset a matrix to the identity, for example:

```
tmat.IdentityMatrix(); // Reset to the identity matrix
```

You can verify if a matrix is equal to the identity by using:

```
BOOL isIdent = tmat.IsIdentity();
```

The following sections discuss the ways matrices can be used in transformations.

## **Multiplying a Vector by a Matrix**

One thing that can be done is to multiply a vector (**Point3**) by a matrix (**Matrix3**). This results in a new vector as a result. This is often used to transform vectors from one coordinate space to another. There is a global function in the SDK that does this.

**Point3 VectorTransform(const Matrix3& M, const Point3& V);**

Transform the vector (**Point3**) with the specified matrix.

Note: In MAX, all vectors are assumed to be row vectors. Under this assumption, multiplication of a vector with a matrix using the \* operator can be written either way (Matrix\*Vector or Vector\*Matrix), for ease of use, and the result is the same -- the (row) vector transformed by the matrix.

## **Multiplying Two Matrices**

It is often also very useful to compute the multiplication of a matrix by another matrix. This produces a matrix as the result. Matrix multiplication is associate but not commutative. That is,  $AB$  is not the same as  $BA$ , but  $(AB)C$  is the same as  $A(BC)$ . This property is very useful since separate matrix transformations can be concatenated together and then all applied at once. 3ds max provides several methods to perform matrix multiplication; each are operators of the **Matrix3** class.

**Matrix3& operator\*=(const Matrix3& M);**

Multiplies this Matrix3 by the specified Matrix3.

**Matrix3 operator\*(const Matrix3&) const;**

Performs matrix multiplication.

## Scaling

Now that we have seen some basic matrix math, let's see how we can create matrices capable of performing typical transformations. Scale transformations can be used, for example, to multiply each component of a vector by a scale factor. Scale transformations use the following matrix positions:

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \\ 0 & 0 & 0 \end{bmatrix}$$

The following global function is provided in the API to create a scale matrix given a **Point3** with the individual scale factors (the scale factors are in the **x**, **y**, **z** data members of the **Point3**):

**Matrix3 ScaleMatrix(const Point3& s);**

Builds a new matrix for use as a scale transformation.

The following method of the **Matrix3** class nulls out the scale values in a matrix:

**void NoScale();**

This method nulls the scale portion of the matrix.

## Translation

Translation matrices use the following matrix positions:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ T_x & T_y & T_z \end{bmatrix}$$

There are global functions in the 3ds max API to create these translation matrices and to apply translation to existing matrices. Several of these are given below:

### **Matrix3 TransMatrix(const Point3& p);**

Builds a new matrix for use as a translation transformation.

A method of **Matrix3** allows the translation portion of the matrix to be set.

### **void SetTrans(const Point3 p)**

Sets the translation portion of this matrix to the specified values.

### **void NoTrans();**

This method zeros the translation portion of this matrix.

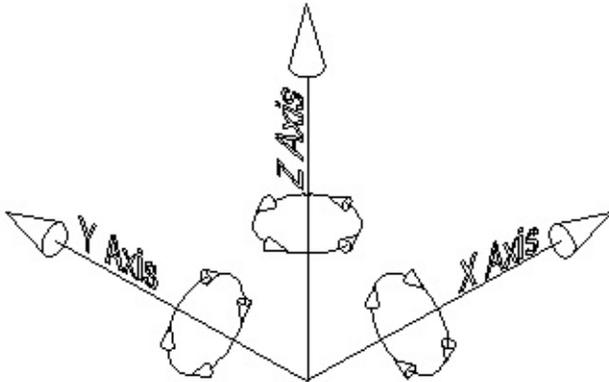
One can also retrieve the translation portion of a matrix using the **Matrix3** method:

### **Point3 GetTrans();**

Returns the translation component of this matrix.

## Rotation

The cells in a rotation matrix are the sines and cosines of the angle of rotation. In the **Matrix3** right-handed coordinate system, positive angles move in a counterclockwise direction, when looking down an axis from positive to negative. In the diagram below, the cones along the XYZ axes point in the positive direction, and the cones in the circles around each axis point in the positive direction of rotation.



The matrices for rotation about the three axes are shown below. In each case **a** represents the angle of rotation in radians:

### X rotation

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(a) & \sin(a) \\ 0 & -\sin(a) & \cos(a) \\ 0 & 0 & 0 \end{bmatrix}$$

### Y rotation

$$\begin{bmatrix} \cos(a) & 0 & -\sin(a) \\ 0 & 1 & 0 \\ \sin(a) & 0 & \cos(a) \\ 0 & 0 & 0 \end{bmatrix}$$

### Z rotation

$$\begin{bmatrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

There are global functions in the 3ds max API to create these rotation matrices and to apply rotation to existing matrices. Several of these are given below:

**Matrix3 RotateXMatrix(float angle);**

Builds a new matrix for use as a X rotation transformation. The angle is specified in radians.

**Matrix3 RotateYMatrix(float angle);**

Builds a new matrix for use as a Y rotation transformation.

**Matrix3 RotateZMatrix(float angle);**

Builds a new matrix for use as a Z rotation transformation.

Developer can also remove the rotation from a given matrix using the **Matrix3** method:

**void NoRot();**

This method nulls the rotation portion of the matrix.

## Creating Quaternions from Rotation Matrices

A quaternion can be created from a rotation matrix. By passing the **Matrix3** to the constructor of the **Quat**, its rotation values are used to initialize the quaternion. The following code shows an example of this. It takes a **Point3 v** and creates a quaternion based on the angles stored after snapping them using the current 3ds max angle snap settings.

```
Matrix3 mat;  
mat.IdentityMatrix();  
mat.RotateX(ip->SnapAngle(v.x));  
mat.RotateY(ip->SnapAngle(v.y));  
mat.RotateZ(ip->SnapAngle(v.z));  
Quat q(mat);
```

## A **Matrix3** as an Axis System

A **Matrix3** can also be used to specify an axis system. For example, the **INode** method **Rotate()** is used to rotate a node about a specified axis system. One of the arguments to this method is a **Matrix3** that specifies this axis system.

Essentially, a matrix **is** just an axis system.

For example, if you think of an axis tripod in a viewport, this is basically all the information a matrix has. The axis tripod has three vectors (the directions of the axes: X, Y, Z) and a position in space (the point where the axes converge). A transformation matrix holds this same information. The 0th row of a matrix is the X vector, the 1st row of the matrix is the Y vector, the 2nd row of the matrix is the Z vector, and the 3rd row of the matrix is the position.

You can picture this if you consider the identity matrix. Its 0th row (X) is [ **1 0 0** ] and this is just a vector along the X axis. Its 1st row (Y) is [ **0 1 0** ] and this is simply a vector along the Y axis. Its 2nd row (Z) is [ **0 0 1** ] and this is just a vector along the Z axis. And the 3rd row (position) is [ **0 0 0** ] and this is the origin. Therefore the identity matrix is just an axis system without any rotation, positioned at the origin. This same thing holds true for any transformation matrix, it's just not as easy to visualize as it is with the identity matrix.

Therefore, any transformation matrix is really just an axis system.

To construct an axis system then, you can simply create an identity matrix, and apply transformations to the matrix to put the matrix into the coordinate system you need. For example, if you had an axis system that was rotated 45 degrees about the Z axis and centered at (10, 20, 30) you could use the following code to build it:

```
Matrix3 mat(1); // Identity  
mat.RotateZ(DegToRad(45.0f));  
mat.SetTrans(Point3(10.0f, 20.0f, 30.0f));
```

## **Transformation Matrix Demonstration Program**

There is a program available in `\MAXSDK\SAMPLES\HOWTO\TMATTEST\TMATTEST.CPP` that shows visually how matrices are constructed for move/rotate/scale transformations. This program also shows the results of some of the **INode** methods that return transformation matrices (for example, **GetNodeTM()**, **GetObjectTM()**, etc.). Developers can load this project into VC++ and build it. Then copy the DLU from `\MAXSDK\PLUGIN\TMATTEST.DLU` into the **STDPLUGS** directory where 3ds max can load it. To run it select **TMatTest** from the **HowTo** section of the Utilities menu. The program operates on the first object in the selection set. Create a node in the scene (for instance a Box) and experiment with animating the node, binding it to space warps, etc., and viewing the resulting matrices returned from the INode methods. You can also build transformations from scratch and apply them to the first selected object.

# Memory Allocation

See Also: [Class Animatable](#), [Dll Functions and Class Descriptors](#), [Class ClassDesc](#).

## Overview

The 3ds max API provides a method used to allocate memory when a new instance of a plug-in class is required. It also provides several different ways to deallocate this memory depending on the plug-in type. In the most common case, these methods are **ClassDesc::Create()** and **Animatable::DeleteThis()**.

## Methods

When 3ds max needs to create a new instance of a plug-in class, it calls the **Create()** method of the class descriptor. This method returns a new instance of the plug-in class. The memory for the class may be allocated by using the **new** operator as shown below:

```
void *Create(BOOL loading = FALSE) { return new MyObject; }
```

For example, when 3ds max loads a file from disk it needs to create instances of the plug-ins that are used (the geometric objects, lights, cameras, materials, controllers, etc.). It does this by calling **ClassDesc::Create()**.

The memory must be freed when 3ds max is done with the item. Usually a method named **DeleteThis()** is called. This method is called to free the memory associated with the plug-in class. Since the memory was allocated with the **new** operator, it must be de-allocated using the **delete** operator as shown below:

```
void DeleteThis() { delete this; }
```

For plug-ins that are part of the **Animatable** class hierarchy, **DeleteThis()** is a method of **Animatable**. For plug-ins that are **not** derived from **Animatable** there may be a non-inherited **DeleteThis()** method. For example, the **UtilityObj** class used to create Utility plug-ins is **not** derived from **Animatable** and has its own **DeleteThis()**.

The **DeleteThis()** method also gives the developer control over deleting. For example, a Utility Plug-In object may be statically declared and not actually declared in the heap. A utility plug-in such as this would implement **DeleteThis()** to do nothing since there is no heap memory to free.

```
void DeleteThis() {}
```

This is just what the

**\MAXSDK\SAMPLES\UTILITIES\ASCIIOUT.CPP** code does.

A few plug-in types have no **DeleteThis()** method at all -- 3ds max deletes the memory directly. The Image Filter and Compositor plug-in types (subclassed from **ImageFilter**) are allocated using **ClassDesc::Create()** but are deleted by the system. In this case the plug-in has no **DeleteThis()** method to implement. The system just calls delete on the plug-in internally (using the **delete** operator).

## Modifier Stack Branching

See Also: [Geometry Pipeline](#), [Class Object](#), [Class INodeTransformed](#).

## Overview

This section discusses the concept of modifier stack branching. This section uses the boolean object plug-in as an example. The SDK sample code for the boolean object is available in

**\MAXSDK\SAMPLES\OBJECTS\BOOLOBJ.CPP**. Note: A developer may wish to read through the Advanced Topics section on the [Geometry Pipeline System](#) before reading the material below.

Normally the geometry pipeline is a single string of object references. For example, the node may have a reference to a derived object. This derived object may reference another derived object. That derived object may then reference the base object. Normally this is where the pipeline would end.

In some cases however, the base object itself has references to other objects. For example, the boolean object (referred to as a compound object) has references to its two operands. The boolean object acts like a multiplexer that decides which parts of the history is available to the user.

In the 3ds max user interface the modifier history is just a linear list. It does not show any kind of tree (branch) structure. Thus at any one time the user can only go down a single branch of the tree. It is up to the object to decide which branch the user is to go down. The standard way of doing this is to provide a user interface control to allow the user to select which branch they want to go down.

With the boolean object the user can go into sub-object selection and select one of the operands. Then that operand's history will become available in the modifier stack. Alternatively the boolean object provides a list of the operands and the user can select them there. When this is done this history becomes available and the user can operate on the selected history.

## Methods used in Modifier Stack Branching

This section presents the class methods a developer works with to manage an object that uses modifier stack branching. The following methods from class **Object** are involved:

### **virtual int NumPipeBranches()**

This method returns the number of pipeline branches combined by the object. This is not the total number of branches, but rather the number that are active. For example, in the boolean object, if the user does not have any operands selected, this methods would return zero. If they have one selected it would return one. Here is the boolean object implementation of this method:

```
int BoolObject::NumPipeBranches()  
{  
    int num=0;  
    if (TestFlag(BOOL_OB1SEL) && ob1) num++;  
    if (TestFlag(BOOL_OB2SEL) && ob2) num++;  
    return num;  
}
```

The boolean object is set up so that a user can only branch down one pipeline at a time. This is not always the case for all compound objects. In the 3ds max loft object, the user may select many shapes on the loft path. For example, the user could select five shapes on the path. **NumPipeBranches()** would return five to indicate that five pipelines are currently active. If all five shapes were the same instance, the user could go down their history all at the same time. This is analogous to when you select five objects in the scene, and if all these objects are all instances (they have commonality) then their history is available. If you select five objects and they don't have commonality their history is not available. If they share commonality to a certain point, their history is available up to and including this point and then the history stops. The pipeline branches are similar. If there is commonality, the common history will be displayed.

### **virtual Object \*GetPipeBranch(int i)**

This method is called to return the 'i-th' branch of the compound object. As the system is calculating the history, when it reaches the base object where it would normally stop, it checks to see if there is any branching. The system calls **NumPipeBranches()** on the object. If this returns anything greater than zero it will then get each of the branches and continue its evaluation. Here is the

boolean object implementation of this method:

```
Object *BoolObject::GetPipeBranch(int i)  
{  
if (i) return ob2;  
if (TestFlag(BOOL_OB1SEL)) return ob1;  
return ob2;  
}
```

```
virtual INode *GetBranchINode(TimeValue t,INode *node,int i)
```

Compound objects like the boolean object or the lofted contain multiple objects and the history of each of these objects can be edited. A problem arises because these compound objects usually apply a transformation to their input objects. For example, the boolean object has a transformation for each of its two operand objects. The boolean object has two operands and each has a transform controller. These transform controllers represent the transformation relative to the boolean object's transformation (its local space). This allows the user to move the two boolean operands around relative to each other without modifying the boolean object's node's transform controller. This arrangement simply provides an additional transformation for the operands.

If the user goes back into the history to edit one of the boolean object's operands and applies an Edit Mesh modifier, the Edit Mesh modifier does not know about the transformation that is happening in the pipeline downstream. Thus it thinks the object is positioned somewhere where it is not since it has no way of knowing about this other transformation. This is a problem (for example if you try to hit test) because the plug-in thinks the object is in the wrong place.

In general every item in the pipeline applies a transformation of some sort. An Edit Mesh modifier deals with this by forcing 'Show End Result' to be off. Currently show end result only effects downstream **modifiers** and not downstream objects so the transformation by the compound object still applies.

The transformation applied by the compound object typically is a linear transformation which, of course, can be inverted. So one **can** edit in the context of the transformation if it is known what it is. Methods are available to make it possible for compound objects to provide access to this transformation:

```
INode *GetBranchINode(TimeValue t, INode *node, int i)
```

This **Object** method will allow a compound object to alter a node's transformation. The way it does this is by creating an item called an

**INodeTransformed.** An **INodeTransformed** is simply a wrapper around an **INode** that adds in an extra transformation to the node's transformation. An object can implement this method by returning a pointer to a new **INodeTransformed** that is based on the node passed into this method. Here is the boolean object implementation of this method:

```
INode *BoolObject::GetBranchINode(TimeValue t, INode *node,  
int i)  
{  
    assert(i<2);  
    int index = 0;  
    if (i) index = 1;  
    else if (TestFlag(BOOL_OB1SEL)) index = 0;  
    else index = 1;  
    return CreateINodeTransformed(node,GetOpTM(t,index));  
}
```

When a modifier, like an Edit Mesh, calls **GetModContexts()** some of the **INodes** in the **INodeTab** may actually be pointing to **INodeTransforms**. When the modifier is done with the table, these wrappers need to be discarded. The method **INodeTransformed::DisposeTemporary()** is provided for this purpose.

## Notifying Dependents of the History Change

One other task needs to be done by a compound object that deals with modifier stack branching. The system needs to be notified when the branching changes. For example, in the boolean object if the user has selected operand A, and then selects operand B (which de-selects operand A) the history must be updated to show the history for operand B. 3ds max does not know that the user just changed operands as that is handled by the boolean object. To let the system know, the boolean object sends the message **REFMSG\_BRANCHED\_HISTORY\_CHANGED** via **NotifyDependents()**. This lets the system know it needs to re-build the history from this point on. For more details on sending messages see the Advanced Topics section on [References](#).

# Network Rendering

See Also: [Class MaxNetManager](#), [Class MaxNet](#)

## Overview

One of the new components of 3ds max R4 is a redesigned Network Rendering pipeline and API that allows developers to take full advantage to this easily accessible architecture. The core of the Network Rendering API is exposed through the **MaxNetManager** class which allows clients to connect to the Network Rendering Manager to perform any and all functions offered. The Network Rendering API encapsulates all the network rendering details and thus leaves the client code to concentrate on whatever it needs to do. The API handles all the networking code and communication protocols.

Within MAX, the Network Job Assignment dialog uses the Network Rendering API to submit new jobs to the network queue while the Queue Manager uses the API to provide all of its functionality. Neither of these components have any other type of network access or private access to network rendering facilities. The **MaxNet** API provides all of these itself.

The entire API is derived from the **MaxNet** class, which is used solely for exception handling. In order to handle exceptions you can catch a MaxNet type as follows:

```
try {  
    // the code being tried  
} catch (MaxNet* maxerr) {  
    // handle the error  
    // do NOT delete maxerr  
    // use maxerr->GetErrorText() to get the error description  
    // use maxerr->GetError() to get the error code  
}
```

The MaxNet class provides the error code in both numeric form, through **GetError()** in the form of **maxnet\_error\_t** type, and in text form, through **GetErrorText()**. Most methods in the Network Rendering API will return errors only through this mechanism, particularly methods that handle network transactions.

The Network Rendering API provides two functions that deal with the creation and destruction of the **MaxNetManager** class instance, these are, **MaxNetManager\* CreateManager()** and **DestroyManager(MaxNetManager\* mgr)**. The **CreateManager()** function will return a newly created instance of the **MaxNetManager** class which can be deleted, when done, by **DestroyManager()**.

## Queue Control

At any given moment, only one connected client can be in control of the network queue. This means, only one connected client can perform changes. All other clients operate in a "read only" mode.

## MaxNetCallback

The Network Rendering API provides a callback mechanism through the **MaxNetManager** class. If you wish to use the callback mechanism you can create your own class derived from **MaxNetCallback** and pass it as the argument for **MaxNetManager::SetCallback()**. All methods are optional, thus you only need to implement those you require.

## Network Rendering API by Example

Two full examples on how to use the MaxNet API are included in the SDK under **\MAXSDK\SAMPLES\NETRENDER\LISTJOBS** and **\MAXSDK\SAMPLES\NETRENDER\JOBASSIGN**.

Note with these samples that the use of the Network Rendering API is not limited to 3ds max plugins only.

# Nodes

See Also: [Class INode](#), [The Node and Object Offset Transformations](#).

## **Overview**

This section presents information on nodes. This includes what they are, their function within MAX, the main class for working with nodes, an overview of their reference structure, and a section on instancing nodes.

## **What are Nodes?**

Nodes are the items in the 3ds max that have a one to one correspondence with objects in the scene. Every procedural object, light, camera, helper object, etc. that appears in the viewports has an associated node. The node stores many properties that allow the item associated with the node to relate to the scene. These are properties such as a transform controller, a material used for rendering, a visibility controller, hidden/unhidden state, frozen/thawed state, wireframe color, and many more.

## **Class INode**

The methods available to work with nodes are from [Class INode](#). These methods provide the functions such as evaluating the geometry pipeline for the node, getting and setting the node name, working with parent/child hierarchies, accessing display attributes of the node, providing access to controllers, etc.

## Node Grouping

A 3ds max user can take a node in the scene and make it part of a group. Groups allow the user to select all the nodes in the group as one element without having to select each piece. Groups can be included in other groups so nesting can occur. There is a method in **INode** that indicates if a node is included in a group. This is **IsGroupMember()**. It returns TRUE if the node is in a group and FALSE if not.

The group is constructed by creating an invisible dummy object and linking the group members to this dummy. The dummy node is called the group head. The **INode** method **IsGroupHead()** tells if the node is one of the dummy nodes that heads a group. It returns TRUE if the node is a group head dummy and FALSE if not. Note that if the group was nested, that is it's a member of another group, it may have its **IsGroupMember()** boolean set as well.

If you have a member of a group and want to get a pointer to the group head use **GetParentNode()**. If you go through the children of the head (and all their children), and all the member booleans are set, the nodes are still part of the same group.

Groups may be opened allowing the user to edit the individual components (without otherwise breaking up the group). The method **IsOpenGroupMember()** returns TRUE if the node is a member of an open group and FALSE if not. Heads can check if they are open by calling **IsOpenGroupHead()**.

## Nodes and References

A node maintains six references (see the Advanced Topics section [References](#) for more details) that allow it to get notified when items that affect it change. These references are listed and defined below:

### **Reference 0: The Transform Controller.**

This is a reference to the node's transform controller. This controller governs the position, rotation and scaling of the node in the scene. Developers can get and set the transform controller used by the node using **INode::**

**GetTMController()** and **SetTMController()**. Most nodes use the Position/Rotation/Scale transform controller (for example mesh objects). Other items (such as target spotlights and cameras) use the Look At transform controller.

### **Reference 1: The Object Reference.**

Each node maintains a pointer to an object. This is a pointer to the base procedural object (mesh, light, camera) or a derived object (see the section [Geometry Pipeline System](#) for details on derived objects). The path that follows between object references are what make up a geometry pipeline in MAX. The node's object reference is where the pipeline starts.

### **Reference 2: The Pin Node for IK.**

This reference is a pointer to the pin node for Inverse Kinematics. This reference is not assigned by default and is thus NULL initially.

### **Reference 3: The Material Reference.**

Materials in 3ds max are assigned at the node level. There is a single material assigned per node. This reference is a pointer to the material for the node. A developer can get and set the material using the methods **INode::GetMtl()** and **SetMtl()**. If a material is assigned, this reference tracks it. This reference is not assigned by default and is thus NULL initially.

### **Reference 4: The Visibility Controller.**

In Track View the user can assign this visibility controller using the 'Add Visibility Track' icon. When this controller is assigned this reference points to the controller managing the node visibility. This reference is not assigned by default and is thus NULL initially.

### **Reference 5: The Image Blur Controller.**

This reference is not assigned by default and is thus NULL initially.

## **Node Hierarchies -- Parent / Child and Groups**

Nodes in 3ds max can be linked to one another to form parent / child hierarchies.

When a node is linked it receive the reference message

**REFMSG\_NODE\_LINK**. When it is unlinked it receives this same message.

Developers can manipulate the hierarchy using methods

**Inode::AttachChild()**, **Detach()**, and **GetParentNode()**. One can check if

a node is linked using **GetParentNode()**. If the pointer returned from that

method is the same as that returned from **IsRootNode()** then the node is not

linked.

## **Nodes and Transformations**

There are two transformation that affect all nodes. These are the node transformation and the object-offset transformation. For details on these see the Advanced Topic Section [Node and Object Offset Transformations](#).

## **Nodes and Instancing**

Nodes cannot be instanced. However, a developer can achieve functionally the same thing by instancing everything that the node points to. So, given two different nodes, if you instance the controllers, objects, and materials of all the child nodes then the hierarchies will behave as instances.

## The Node and Object Offset Transformations

See Also: [Class INode](#), [Class Interval](#), [Class Quat](#), [Class ScaleValue](#), [Matrix Representations of 3D Transformations](#), [Keyframe and Procedural Controller Data Access](#).

## Overview

This topic provides information on two transformations related to nodes -- the node transformation matrix and the object-offset transformation. This section also presents information on how these transformation are constructed, how they are used by MAX, and how they may be used by developers. This section also discusses the methods of the **Inode** class that access these transformations.

## **The Pivot Point -- The Node Transformation Matrix**

The transform center, or pivot point, is the location about which a rotation takes place, or to and from which a scale occurs. All nodes in 3ds max have a pivot point. You can think of the pivot point as representing a node's local center and local coordinate system.

The pivot point of a node is used for a number of purposes:

- As the center for rotation and scaling.
- Defines the transform origin for linked children.
- Defines the joint location for Inverse Kinematics.

The thing that most users think of as the pivot point -- graphically represented in 3ds max by the axis tripod that is displayed when a node is selected and the coordinate system is set to local -- is actually just a visual representation of the node's transformation matrix (NodeTM).

The node's transformation matrix is what is controlled by the transform controller that places the node in the scene. The transform controller controls the transformation relative to the node's parent.

## How a Transform Controller Computes the NodeTM

This section provides an overview of how 3ds max gets data from a transform controller to create the node transformation matrix (NodeTM).

A controller can be thought of as operating internally within its own local coordinate system. For both keyframe and procedural controllers, the data generated is provided in the local space of the controller. It is not until the controller supplies its transformation to 3ds max does the coordinate system become relative to anything other than the controller itself. The way this works is as follows:

All controllers are derived from class **Control** (or **StdControl**, which is itself derived from **Control**). Every controller, regardless of type, provides data to 3ds max at a specified time via a method of the **Control** class called **GetValue()**.

**Prototype:**

```
virtual void GetValue(TimeValue t, void *val, Interval &valid,  
    GetSetMethod method=CTRL_ABSOLUTE)=0;
```

One of the parameters passed to this method is **void \*val**. Depending on the controller type, different data types are passed using this pointer (see [Class Control::GetValue\(\)](#) for a complete discussion of all the data types). For example, a transform controller will be passed a **Matrix3** data type. It is the job of this transform controller to update the matrix with its transformation. It does this by **pre-multiplying** its transformation with the matrix passed. The controller operates within its own space internally, but when it pre-multiplies its data with the matrix passed, it essentially means the controller is then working in the coordinate system of the matrix. Another way to think of this is that the **Matrix3** passed is actually transforming the controllers data. Thus, what the final coordinate system ends up being depends upon the client of the controller and not on the controller itself -- it depends on the matrix passed to **GetValue()**.

To better understand this consider the following examples. For a transform controller, the matrix passed to **GetValue()** is the transformation matrix of the node's parent. Consider a simplified keyframe transform controller that only provided position (no rotation or scale). This controller would just update the bottom row of the matrix (translation data) passed by pre-multiplying its value. If the node the controller was assigned to was not hierarchically linked, that is it had no parent other than the world, then the matrix passed to **GetValue()** would

be the identity. In this case, when the controller pre-multiplied its value, it would simply be setting the translation row.

Consider a different case when the node the controller is assigned to is hierarchically linked. As before, the keys of the controller are stored in the local space of the controller. But in this case, when **GetValue()** is called, the matrix passed is not necessarily the identity. Rather it will be the transformation matrix of the parent of the node the controller is assigned to. If the parent is not located at the origin, or is scaled or rotated in some way, the parent matrix passed will provide an additional transformation to the one the controller itself provides.

For transform controllers, the matrix that is updated in **GetValue()** becomes the NodeTM and is used to position and orient the node in the scene. The next section discusses the way the transformations are applied to the matrix by the PRS controller to create the NodeTM.

## Construction of the NodeTM for the PRS Transform Controller

This section describes how 3ds max constructs the NodeTM for the PRS controller. The PRS controller uses sub-controllers to create the final transformation. There are three sub-controllers -- one for Position, one for Rotation and one for Scale. The PRS controller is passed a matrix to its **Control::GetValue()** method. This matrix is the parent of the node. If the node has no parent, the matrix starts out as the identity matrix. **GetValue()** is first called on the position controller, then the rotation controller, then the scale controller. In the case of a node with no parent, when its position controller's **GetValue()** method is called, the identity is passed. First the position controller pre-multiplies its position. If the matrix passed is the identity, this is equivalent to setting the bottom row (the translation row) of the matrix. Next, **GetValue()** is called on the rotation controller and the matrix passed includes the position controller information. The rotation is pre-multiplied in. Next, **GetValue()** is called on the scale controller and the matrix passed includes the position and rotation controller information. The scale is pre-multiplied in.

The position is not affected by rotation or scale. The rotation is affected by the position. The scale is affected by the position and rotation. Said another way, the position is in the space of the parent. The rotation is in the space of the parent PLUS the position (the position has already applied itself to the matrix). Scale works in the coordinate system of the parent PLUS position PLUS rotation. The scaling is pre-multiplied however, so the scaling takes place about the local coordinate system, but then if the node is rotated, this scaling is rotated around. If the transformations were not applied in this way, the scaling would be skewed by the rotation.

Note that some position controllers can actually apply more than just position to the matrix. For example, the Path Controller when the Follow switch is active. In this case, the position controller actually applies some rotation to have the node remain tangent to the path it is following. Thus when the rotation controllers receives the matrix in **GetValue()**, the matrix already has some rotation applied. The matrix passed to **GetValue()**, after the position, rotation and scale transformation have updated it, becomes the NodeTM 3ds max uses to position, rotate and scale nodes in the scene.

## Methods

There is a method of the **INode** class that a developer may use to get the node transformation matrix:

```
virtual Matrix3 GetNodeTM(TimeValue t, Interval*  
valid=NULL)=0;
```

This method returns the world space transformation matrix of the node at the specified time. This matrix contains its parents transformation. The Node TM is inherited. The Node TM may be considered the world space transformation as far as kinematics is concerned.

There are two methods that a developer may use to set the node transformation matrix. From **INode** is:

```
virtual void SetNodeTM(TimeValue t, Matrix3& tm)=0;
```

This sets the node's world space transformation matrix. This will call **SetValue()** on the transform controller.

The following method is from Class **Interface**:

```
virtual void SetNodeTMRelConstPlane(INode *node, Matrix3&  
mat)=0;
```

This sets the node's transform relative to the current construction plane. This may be used during creating so you can set the position of the node in terms of the construction plane and not in world units.

## **The Object-Offset Transformation**

The object offset transformation provides an offset of the geometry of an object from its node. This section describes the object offset transformation and its function in MAX.

One can see a node's pivot point graphically represented in 3ds max by selecting a node, going to the Hierarchy branch of the command panel, selecting the 'Pivot' button, and choosing either the 'Affect Pivot Only' or 'Affect Object Only' button. This displays a large axis tripod that shows the location and orientation of the node's pivot point (NodeTM). By choosing one of these buttons, and using the Move/Rotate/Scale toolbar controls, a user can manipulate the position of the geometry of the object independent of the pivot point. Or they may manipulate the pivot point independent of the geometry of the object position. One can think of the geometry of the object as the 'mesh' of the object.

The way the user is able to independently manipulate the pivot and the object is managed internally using the object-offset transformation. The object-offset transformation affects how the geometry of the object is related to the node. The object-offset transformation transforms the geometry of the object itself some amount from the node.

To understand how the object-offset is used, consider the following example from the 3ds max user interface. In the Hierarchy branch under 'Pivot', when the user has chosen the 'Affect Object Only' button they are free to move the geometry of the object around independent of the node. The pivot point does not move -- only how the geometry of the object is oriented relative to the pivot. What is happening internally is the object-offset transformation is being manipulated. This transformation is simply an additional offset that may be applied to the geometry of the object that is independent of the node. The object-offset transformation is not inherited by any child nodes.

As another example consider the use of the 'Affect Pivot Only' button. This mode lets the user move the pivot without affecting the position of the geometry of the object. When the system allows the user to move the pivot point, what is actually happening is the node's transformation is being altered (to re-orient the pivot point), then the object-offset transformation is adjusted to counter the node transformation. This lets the geometry of the object stay in the same place while the pivot point moves around. So again, when the user is moving the pivot point, 3ds max is actually adjusting the node transformation matrix and counter-adjusting the object-offset transformation.

## **Construction of the Object Offset Transformation**

The object offset transformation consists of separate position, rotation and scale transformations. Like the NodeTM, these are applied by pre-multiplying position, then rotation, then scale. Thus the object offset transformation is:

Object Offset Transformation = Offset Scale \* Offset Rotation \* Offset Position

Unlike the NodeTM, the Object Offset Transformation is not inherited by children of the node.

## Methods

The **INode** class has methods to get and set individual transformations of the object offset. These are:

**virtual void SetObjOffsetPos(Point3 p)=0;**

Sets the position portion of the object offset from the node.

**virtual Point3 GetObjOffsetPos()=0;**

Returns the position portion of the object-offset from the node as a Point3.

**virtual void SetObjOffsetRot(Quat q)=0;**

Sets the rotation portion of the object offset from the node.

**virtual Quat GetObjOffsetRot()=0;**

Returns the rotation portion of the object-offset from the node.

**virtual void SetObjOffsetScale(ScaleValue sv)=0;**

Sets the scale portion of the object offset from the node.

**virtual ScaleValue GetObjOffsetScale()=0;**

Returns the scale portion of the object-offset from the node.

Constructing an offset TM for a node is done with the following code:

```
Matrix3 tm(1);
```

```
Point3 pos = node->GetObjOffsetPos();
```

```
tm.PreTranslate(pos);
```

```
Quat quat = node->GetObjOffsetRot();
```

```
PreRotateMatrix(tm, quat);
```

```
ScaleValue scaleValue = node->GetObjOffsetScale();
```

```
ApplyScaling(tm, scaleValue);
```

The **INode** class has a set of methods that allow a developer to transform the node or geometry in the same way as the hierarchy Affect Pivot Only and Affect Object Only modes do. The entire node and object may be transformed, just the pivot, or just the geometry of the object. These methods are:

```
virtual void Move(TimeValue t, const Matrix3& tmAxis,  
const Point3& val, BOOL localOrigin=FALSE,  
BOOL affectKids=TRUE, int pivMode=PIV_NONE,  
BOOL ignoreLocks=FALSE)=0;
```

This method may be called to move the node about the specified axis system. Either the pivot point, or the geometry of the object, or both the pivot and the

object may be transformed. Optionally, any children of the node can be counter transformed so they don't move.

```
virtual void Rotate(TimeValue t, const Matrix3& tmAxis,  
    const AngAxis& val, BOOL localOrigin=FALSE,  
    BOOL affectKids=TRUE, int pivMode=PIV_NONE,  
    BOOL ignoreLocks=FALSE)=0;
```

This method may be called to rotate the node about the specified axis system. Either the pivot point, or the geometry of the object, or both the pivot and the object may be transformed. Optionally, any children of the node can be counter transformed so they don't rotate. There is also a Quat version of this method.

```
virtual void Scale(TimeValue t, const Matrix3& tmAxis,  
    const Point3& val, BOOL localOrigin=FALSE,  
    BOOL affectKids=TRUE, int pivMode=PIV_NONE,  
    BOOL ignoreLocks=FALSE)=0;
```

This method may be called to scale the node about the specified axis system. Either the pivot point, or the geometry of the object, or both the pivot and the object may be transformed. Optionally, any children of the node can be counter transformed so they don't scale.

## The ObjectTM

The `INode` method **GetObjectTM()** returns the transformation matrix an object needs to be multiplied by to transform it into world space. This includes the parent transformation, the node transformation (NodeTM) and the Object Offset Transformation. Thus, the entire transformation used to transform the points of any object is:

$$\text{ObjectTM} = \text{Offset Scale} * \text{Offset Rotation} * \text{Offset Position} * \text{Controller Scale} * \text{Controller Rotation} * \text{Controllers Position} * \text{Parent Transformation}.$$

A developer can retrieve the entire transformation of the object-offset, node and parent using the **INode** method:

```
virtual Matrix3 GetObjectTM(TimeValue time, Interval* valid=NULL)=0;
```

This is the entire transformation of a node. When an object is actually translated into world space, this is the matrix used. This is the Object Offset Transformation, plus the NodeTM, plus the world space modifier effect.

This matrix could be used, for example, if you have a **TriObject** and wanted to get the world space coordinate of one of its vertices. You could do this by taking the vertex coordinate in object space and multiplying it by the matrix returned from this method.

The ObjectTM is not inherited. See the remarks on this method in the reference section documentation for more details.

## **A Sample Plug-In Using the Node and Object Offset Transformations**

The following code is from the Reset Transform utility (**MAXSDK\SAMPLES\MODIFIERS\RESETTM.CPP**). This utility is used to push object rotation and scale values onto the Modifier Stack and align object pivot points and bounding boxes with the world coordinate system. Reset Transform removes all rotation and scale values from selected objects and places those transforms in an XForm modifier. The code below demonstrates the use of the TMs discussed above.

As previously noted, there are two transforms that affect an object: the node transform which is the result of the transform controller plus inheritance, and the object offset transform which positions the geometry relative to the pivot point. The Reset Transform utility takes the node transform (minus inheritance), and the offset transform, and combines them into a matrix which is given to an XForm modifier which is then placed at the top of the modifier stack.

The node and offset transforms can then be set back to the identity (actually, the position component of the node TM is left unchanged). Here is the code -- it operates on all the selected nodes.

```
for (int i=0; i<ip->GetSelNodeCount(); i++) {  
    INode *node = pi->GetSelNode(i);  
    Matrix3 ntm, ptm, rtm(1), piv(1), tm;  
  
    // Get Parent and Node TMs  
    ntm = node->GetNodeTM(ip->GetTime());  
    ptm = node->GetParentTM(ip->GetTime());  
  
    // Compute the relative TM  
    ntm = ntm * Inverse(ptm);  
  
    // The reset TM only inherits position  
    rtm.SetTrans(ntm.GetTrans());  
  
    // Set the node TM to the reset TM  
    tm = rtm*ptm;  
    node->SetNodeTM(ip->GetTime(), tm);
```

```
// Compute the pivot TM
piv.SetTrans(node->GetObjOffsetPos());
PreRotateMatrix(piv,node->GetObjOffsetRot());
ApplyScaling(piv,node->GetObjOffsetScale());

// Reset the offset to 0
node->SetObjOffsetPos(Point3(0,0,0));
node->SetObjOffsetRot(IdentQuat());
node->SetObjOffsetScale(ScaleValue(Point3(1,1,1)));

// Take the position out of the matrix since
// we don't reset position
ntm.NoTrans();

// Apply the offset to the TM
ntm = piv * ntm;

// Apply a derived object to the node's object
Object *obj = node->GetObjectRef();
IDerivedObject *dobj = CreateDerivedObject(obj);

// Create an XForm mod
SimpleMod *mod = (SimpleMod*)ip->CreateInstance(
OSM_CLASS_ID,
Class_ID(CLUSTOSM_CLASS_ID,0));

// Apply the transformation to the mod.
SetXFormPacket pkt(ntm);
mod->tmControl->SetValue(ip->GetTime(),&pkt);

// Add the modifier to the derived object.
ModContext* mc = new ModContext(new Matrix3(1), NULL,
NULL);
dobj->AddModifier(mod, mc);
```

```
// Replace the node's object  
node->SetObjectRef(dobj);  
}
```

## The 'Local' Transformation Matrix

To retrieve what is considered the local transformation (the transformation of a node relative to its parent) you must perform some matrix arithmetic on the nodes world space transformation (NodeTM). This is because there is not a local transformation matrix of a node stored by MAX.

What is often considered the local transformation is the transformation matrix of the node relative to its parent. However, the transformation of a node relative to its parent may be some function of the node's parent's transform. For example, a transform controller takes the parent's TM and modifies it. When a node evaluates itself it takes the parent's TM and passes it as an argument when it calls **GetValue()** on its transform controller. The task of a transform controller in its implementation of **GetValue()** is to modify this matrix. It applies its transformation to the parent transformation passed in.

In some cases this modification may be just a simple pre-multiplication by what is considered the local transformation matrix. But it may be some other more-complicated process. For example, if rotation or scale inheritance is turned off, the transform controller takes the parent's matrix and perhaps removes rotation from it, or removes scaling from it, and then applies itself. As another example, it may use the parent's position as a function to derive rotation -- as in a Look At controller. Thus, what is considered the local transformation is a function of the transform controller and is not stored by MAX.

What 3ds max does store is the node's world space transformation matrix (NodeTM). This matrix includes the parent's transformation. To understand how the local transformation can be extracted from the world space transformation consider the following:

Any transformation is made up by starting with the node, and then multiply it by its parent, and then by its parent, and so on, all the way through the ancestors.

This is shown below:

$$\text{NodeWorldTM} = \text{NodeLocalTM} * \text{ParentLocalTM} * \text{ParentLocalTM} * \text{ParentLocalTM}, \text{ etc.}$$

The parents world transformation is equal to the product of all its ancestors, so:

$$\text{ParentWorldTM} = \text{ParentLocalTM} * \text{ParentsLocalTM} * \text{ParentsLocalTM}, \text{ etc.}$$

The nodes world transform can then be expressed as:

$$\text{NodeWorldTM} = \text{NodesLocalTM} * \text{ParentWorldTM}.$$

If we multiply both sides of this equation by the Inverse of the ParentsWorldTM

and simplify we get:

$$\text{NodeLocalTM} = \text{NodeWorldTM} * \text{Inverse}(\text{ParentWorldTM})$$

So, to retrieve what is considered the local transformation you must get the parent's transform and multiply the node's transform by the inverse of the parent. The way this is computed using the methods of 3ds max is shown in the following code fragment. This example assumes **node** is the **INode\*** that we want the local TM from, and the result will get stored in **localTM**:

```
...
INode *parent;
Matrix3 parentTM, nodeTM, localTM;
nodeTM = node->GetNodeTM(0);
parent = node->GetParentNode();
parentTM = parent->GetNodeTM(0);
localTM = nodeTM*Inverse(parentTM);
...
```

## Node Linking and Grouping

This section discusses the linking and grouping of nodes, and the effect this has on the transformations of the nodes.

In MAX, nodes in the scene may be linked together to form a hierarchy. A node that is linked to another node is referred to as a child node. A node that has children is referred to as a parent node. A node may have several children, but only a single parent. See [Parent Child Hierarchy](#) for more details.

Nodes may also be grouped. When nodes are grouped together, 3ds max effectively creates a new dummy node, and links the group nodes to the parent dummy node. 3ds max maintains other information such as if the group is open or closed, etc., but this just effects the user interface. Essentially grouping is just like linking all the nodes to a dummy.

The following discussion applies to both linking and grouping (since grouping is just a special case of linking).

When a parent-child link is created, the user will not want the nodes in the scene to move. To ensure that the nodes won't move, the transform controllers of the nodes need to be modified. This is because the reference coordinate system has changed. The TM controller is working in a coordinate system which is its parent's coordinates system. When a node is unlinked, its reference coordinate system is the world coordinate system. When a node is linked, its reference coordinates system becomes its parents. So, when nodes are linked the parents change, and 3ds max must adjust the transformation matrices to counteract the change in reference coordinates system. All the keys created by a keyframe transform controller must be updated to reflect the change. When a developer creates the link, the nodes and keys must be adjusted in a similar fashion. An option to the INode method **AttachChild()** that links a node as a child allows this to be done automatically.

**virtual void AttachChild(INode\* node, int keepPos=1)=0;**

Makes the specified node a child of this node. If **keepPos** is nonzero, the child is adjusted as needed to keep it from moving. **keepPos** counters the change in the reference coordinate system.

## Access to Node Controller Data

In addition to accessing the node and object offset transformations, developers may want to access the keyframes or data of a node's transform controller. See the Advanced Topics section [Keyframe and Procedural Controller Data Access](#) for information and sample code.

## Summary

The node maintains two transformations. The node's transformation matrix is what is controlled by the transform controller that places the node in the scene. The object-offset transformation represents a separate position, rotation and scale orientation of the geometry of the object independent of the node. The primary methods available to work with the node and these matrices are part of the **INode** class.

## Object Creation Methods

See Also: [Class BaseObject](#), [Class Interface](#), [Class CommandMode](#), [Class MouseCallBack](#), [Class ClassDesc](#), [Class Matrix3](#), [Class IDerivedObject](#).

## Overview

This section discusses several different approaches a developer may use to manage the object creation process for their plug-in.

In many cases the creation process of an object is handled by the system. For example, in the procedural sphere, the developer simply provides a callback object to handle the user / mouse interaction. This is done using the method **BaseObject::GetCreateMouseCallback()**. This method returns a callback object that handles the user input during the creation process. The system takes care of calling this callback as the user works with the mouse. It also takes care of creating the instance of the plug-in class, creates the node in the scene, and sets the node's object reference to point to the plug-in object. Thus the system handles most of the work.

In certain cases a developer may need to manage the creation process at a lower level in order to provide special functionality. Two methods of the plug-in's Class Descriptor, **BeginCreate()** and **EndCreate()**, allow the plug-in to manage the process. When the system is about to create a plug-in object **BeginCreate()** is called. If the plug-in does not wish to have a custom creation process it can return 0 from this method and the system will invoke the default process. The default implementation of this method returns 0, so if the plug-in does not override it the system handles the creation. If the plug-in does require a custom creation phase, it does so in its implementation of this method. When it is finished it returns nonzero to indicate it does not want the default process invoked.

The target camera is an example of a plug-in that must handle its own creation process. This is because it creates two nodes in the scene: the camera and the target. It also must link the camera to the target using a lookat controller that ensures the camera is always 'looking at' the target. The system-managed creation procedure using **GetCreateMouseCallback()** is not adequate to meet the needs of this plug-in. Therefore the target camera implements **ClassDesc::BeginCreate()** and **ClassDesc::EndCreate()**. You may review the source code for this plug-in in **\MAXSDK\SAMPLES\OBJECTS\CAMERA.CPP**.

Note that **GetCreateMouseCallback()** is pure virtual. Therefore plug-ins derived from **BaseObject** that use **ClassDesc::BeginCreate()** must still provide a NULL implementation of this method. This may be done by

implementing it as:

```
CreateMouseCallBack* GetCreateMouseCallBack() {return NULL;}
```

Another plug-in that manages its own creation process by implementing

**BeginCreate()** and **EndCreate()** is the Ring Array system.

As a final note about these methods, a plug-in developer can force the system to call **ClassDesc::EndCreate()** by calling the method

**Interface::StopCreating()**. This terminates the creation process.

## Objects, Nodes and Derived Objects

A developer may wish to create instances of the standard 3ds max primitives. For example, an import / export plug-in might want to create standard 3ds max spheres, cones, tori, etc. APIs are provided so a plug-in can create objects and nodes in the scene. Additionally plug-ins can create derived objects to add modifiers to objects. This section introduces the methods required and provides some sample code demonstrating their use.

The standard 3ds max primitives, or in fact any plug-in procedural object that 3ds max has registered, may be created given the ClassID and SuperClassID of the object. This is done using the following method:

```
virtual void *CreateInstance(SClass_ID superID, Class_ID classID)=0;
```

This creates an instance of a registered class. This will call **Create()** on the class descriptor. This method returns a pointer to the specified object. For reference see [List of Class IDs](#) and [List of Super Class IDs](#). Note: This method is from class Interface, but there is also a global version of this API that may be used at any time (defined in `\MAXSDK\INCLUDE\PLUGAPI.H`).

To set the parameters for these objects methods of **BaseObject** are available:

```
virtual IParamArray *GetParamBlock();
```

If an object or modifier wishes, it can make its parameter block available for other plug-ins to access. This method returns a pointer to the item's parameter block.

```
virtual int GetParamBlockIndex(int id);
```

If a plug-in makes its parameter block available (using **GetParamBlock()**) then it will need to provide #defines for indices into the parameter block. These #defines should not be directly used with the parameter block but instead converted by this function that the plug-in implements. In this way if a parameter moves around in a future version of the plug-in the #define can be remapped.

For reference see [List of Parameter Block IDs](#). This lists all the parameter block indices for the standard plug-ins that ship with MAX.

Note: To use these parameter block IDs, make sure you have the following statement in your source code:

```
#include "istdplug.h"
```

To create a node in the scene given an object pointer use this method from class Interface:

```
virtual INode *CreateObjectNode( Object *obj )=0;
```

This creates a new node in the scene with the given object. Methods of INode allow the node to be named, and assigned a transform controller.

Developers may also create derived objects and add modifiers to objects.

Methods of class **IDerivedObject** are available for this purpose. For example:

```
virtual void AddModifier(Modifier *mod, ModContext  
*mc=NULL,  
int before=0)=0;
```

Adds a modifier to this derived object.

The following code, from

**\MAXSDK\SAMPLES\UTILITIES\UTILTEST.CPP**, demonstrates the use of these methods.

```
#include "istdplug.h"  
void UtilTest::MakeObject()  
{  
    // Create a new object through the CreateInstance() API  
    Object *obj = (Object*)ip->CreateInstance(  
        GEOMOBJECT_CLASS_ID,  
        Class_ID(CYLINDER_CLASS_ID,0));  
    assert(obj);  
    // Get a hold of the parameter block  
    IParamArray *iCylParams = obj->GetParamBlock();  
    assert(iCylParams);  
    // Set the value of radius, height and segs.  
    int rad = obj->GetParamBlockIndex(CYLINDER_RADIUS);  
    assert(rad>=0);  
    iCylParams->SetValue(rad,TimeValue(0),30.0f);  
    int height = obj->GetParamBlockIndex(CYLINDER_HEIGHT);  
    assert(height>=0);  
    iCylParams->SetValue(height,TimeValue(0),100.0f);  
    int segs = obj->  
>GetParamBlockIndex(CYLINDER_SEGMENTS);
```

```

assert(segs>=0);
iCylParams->SetValue(segs,TimeValue(0),10);
// Create a derived object that references the cylinder
IDerivedObject *dobj = CreateDerivedObject(obj);
// Create a bend modifier
Modifier *bend = (Modifier*)ip->CreateInstance(
    OSM_CLASS_ID,
    Class_ID(BENDOSM_CLASS_ID,0));
// Set the bend angle
IParamArray *iBendParams = bend->GetParamBlock();
assert(iBendParams);
int angle = bend->GetParamBlockIndex(BEND_ANGLE);
iBendParams->SetValue(angle,TimeValue(0),90.0f);
// Add the bend modifier to the derived object.
ModContext* mc = new ModContext(new Matrix3(1), NULL,
NULL);
dobj->AddModifier(bend, mc);
// Create a node in the scene that references the derived object
INode *node = ip->CreateObjectNode(dobj);
// Name the node and make the name unique.
TSTR name(_T("MyNode"));
ip->MakeNameUnique(name);
node->SetName(name);
// Redraw the viewports
ip->RedrawViews(ip->GetTime());
}

```

Here is another piece of sample code that creates a shape object using the CreateInstance API. This example creates a circle, but one could modify it to create lines or other shapes.

```

// Vector length for unit circle
#define CIRCLE_VECTOR_LENGTH 0.5517861843f

void Utility::Test() {
    Interface *ip = theUtility.ip;
    Object *obj =

```

```

(Object*)ip->CreateInstance(SHAPE_CLASS_ID, splineShapeClassID);

SplineShape *ss =(SplineShape *)obj;
BezierShape &ashape = ss->GetShape();
ashape.NewShape();
Spline3D *spline = ashape.NewSpline();

float radius = 300.0f;
float vector = CIRCLE_VECTOR_LENGTH * radius;

for(int ix=0; ix<4; ++ix) {
    float angle = 6.2831853f * (float)ix / 4.0f;
    float sinfac = (float)sin(angle), cosfac = (float)cos(angle);
    Point3 p(cosfac * radius, sinfac * radius, 0.0f);
    Point3 rotvec = Point3(sinfac * vector, -cosfac * vector, 0.0f);
    spline->AddKnot(SplineKnot(KTYPE_BEZIER,LTYPE_CURVE,
        p,p + rotvec,p - rotvec));
}
spline->SetClosed();
spline->ComputeBezPoints();
ashape.UpdateSels();
ashape.InvalidateGeomCache();

INode *node = ip->CreateObjectNode(obj);
TSTR name(_T("MySplineNode"));
ip->MakeNameUnique(name);
node->SetName(name);

// Redraw the views
ip->RedrawViews(ip->GetTime());
}

```

## Other Methods of Object Creation - NonMouseCreate()

A method of class Interface named **NonMouseCreate()** may be used to create a new object / node without going through the usual create mouse proc sequence. This call must be made when the system is ready to create -- that is during execution of the system-provided default object creation process. To understand how this works consider the example of the procedural sphere.

When the user selects Sphere from the 3ds max UI, an instance of the sphere is created and its parameters appear. There is not a node in the scene, but an instance of the object is created. The user can then edit the sphere parameters before it is put in the scene. After a sphere is created, the creation process is in a different phase -- the user is editing the parameters of the sphere that was just created -- not the one about to be created. Therefore the first sphere is a special case -- the user can edit the parameters both before and after it is created. Subsequent spheres can only have their parameters edited after they are created. The procedural sphere's user interface allows the user to type in a center position and radius value and then press a 'Create' button. This creates a new sphere in the scene with these parameters. When the button is pressed it calls the method **NonMouseCreate()**.

When the **NonMouseCreate()** method is used, the default creation manager looks to see if a sphere has just been created in the scene. If there is one, it ends the creation phase on that sphere and creates a new one and puts it in the scene. If there is not a newly-created sphere yet, **NonMouseCreate()** will simply put the object into the scene.

Thus calling **NonMouseCreate()** treats the object creation just as if the user clicked the mouse in the viewport to create the object -- but they cannot interactively adjust the parameters. The method takes a single **Matrix3** argument which is the transformation relative to the construction plane. In the code below, the translation part of this matrix is set to the values the user typed in for the X, Y, and Z location of the sphere. The user is thus typing in values that are relative to the current construction plane.

The following code is from the procedural sphere. This is the code that is executed if the user presses the 'Create' button in the 'Keyboard Entry' rollup page. This allows the user to type in the values for the sphere's position and radius using the keyboard and then create the object.

**BOOL SphereTypeInDlgProc::DlgProc(**

```

TimeValue t,IParamMap *map,HWND hWnd,UINT msg,
WPARAM wParam,LPARAM lParam) {
switch (msg) {
case WM_COMMAND:
switch (LOWORD(wParam)) {
case IDC_TI_CREATE: {
// The user has pressed the create button...
// If the radius is zero there is nothing to create
// so just return.
if (so->crtRadius==0.0) return TRUE;
// We only want to set the value if the object is
// not in the scene.
if (so->TestAFlag(A_OBJ_CREATING)) {
so->pblock->SetValue(PB_RADIUS,0,so->crtRadius);
}
// Create an identity matrix
Matrix3 tm(1);
// Update the translation portion of the matrix to the
// creation position type-in parameter
tm.SetTrans(so->crtPos);
// Call the interface method to create the node...
// This will use the values in the objects pblock.
so->ip->NonMouseCreate(tm);
so->suspendSnap = FALSE;
// NOTE that calling NonMouseCreate() will cause this
// object to be deleted. DO NOT DO ANYTHING BUT RETURN.
return TRUE;
}
}
break;
}
return FALSE;
}

```

## **Summary**

There are several ways a developer may create objects in the scene. Some rely on the developer providing a custom creation process command mode and mouse proc. Others use the default object creation manager. In either case, developers may create any plug-in object that the system has loaded from its DLL at 3ds max start up. Plug-ins have access to the parameter block of all the standard 3ds max primitives. Developers may also add object modifiers to any of the objects created.

## Object Modification

See Also: [Class Modifier](#), [Class SimpleMod](#), [Class Object](#), [Class Deformer](#), [Class ModContext](#), [Class Mesh](#), [List of Channel Bits](#), [List of Class IDs](#), [Class BitArray](#).

## **Overview**

This topic presents information on modifying objects in MAX. It discusses how objects and modifiers interact to enable object modification, explains the various types of object modification, and shows key portions of sample code from modifiers of each type.

## Objects, Modifiers, and the Geometry Pipeline

Modifiers operate on objects flowing down a geometry pipeline. A geometry pipeline is the system 3ds max uses to process the modification of objects. There are several methods implemented by the objects and the modifiers that allow this to happen. First we'll look at what modifiers need to do to inform 3ds max about the nature of their modification. Then we'll examine what procedural objects need to do to make themselves 'modifyable'.

### Modifiers Request A Certain Type of Input

Modifiers may only operate on certain type of object. For example, the Edit Spline modifier only works on spline shapes. The Volume Select Modifier only works on TriObjects (triangle mesh objects). In order for the modifier to inform 3ds max what type of object it needs it implements the method

**Modifier::InputType()**. This method returns the **Class\_ID** of the type of object it needs for input. For example, the Edit Spline modifier implements this method as shown below:

```
Class_ID InputType() { return  
Class_ID(SPLINESHAPE_CLASS_ID,0); }
```

This tells 3ds max which objects the modifier can be applied to. Only those objects that are spline shapes, or are able to convert themselves to spline shapes, may be modified by Edit Spline. The same is true for Volume Select Modifier. It requires objects that are triangle meshes for it to do its work. Therefore it implements **InputType()** as shown below:

```
Class_ID InputType() { return Class_ID(TRIOBJ_CLASS_ID,0);  
}
```

When an object is selected in MAX, only those modifiers that are appropriate for modifying it are enabled in the user interface. This is done by querying the modifiers to see if their **InputType()** matches the **Class\_ID** of the selected object, or the objects that it can convert itself to. If it is, the modifier is enabled, otherwise its disabled.

### Modifiers Need To Indicate Which Geometry Pipeline Channels They Require and Alter

Objects flowing down the geometry pipeline are broken into separate channels. That is, the pipeline can operate on only portions of objects and not the entire thing as a whole. Some examples of channels that objects are broken into are the geometry portion (points), the topology portion (polygons or faces), the selection

level of the object (object, face, vertex, etc.), and its texture coordinates. The purpose of this is basically to allow the pipeline to operate more efficiently. Modifiers must inform 3ds max which channels they require to perform their modification. Copies of the needed channels are made and passed up the pipeline. 3ds max will only pass copies of a minimum set of needed channels up the pipeline. If a certain channel is not required by any modifiers anywhere in a pipeline it is not copied and passed along.

Modifiers inform 3ds max which channels they require by implementing the method **Modifier::ChannelsUsed()**. This list of channels must include the channels the modifier actually alters but may include more. Here is the implementation of **ChannelsUsed()** by the Volume Select modifier:

```
ChannelMask ChannelsUsed()  
{  
    return OBJ_CHANNELS;  
}
```

Note that **OBJ\_CHANNELS** is defined as:

```
#define OBJ_CHANNELS  
(TOPO_CHANNEL | GEOM_CHANNEL |  
SELECT_CHANNEL | TEXMAP_CHANNEL |  
MTL_CHANNEL | SUBSEL_TYPE_CHANNEL |  
DISP_ATTRIB_CHANNEL | VERTCOLOR_CHANNEL |  
GFX_DATA_CHANNEL)
```

This means that the pipeline must have all these channels up to date in the object that gets passed to the modifier before it can properly do its work. Technically, this is actually overkill for the Normals modifier. It wouldn't need the **TEXMAP\_CHANNEL** and **VERTCOLOR\_CHANNEL** since they aren't in fact used. It really only needs the **GEOM\_CHANNEL**, **TOPO\_CHANNEL**, and the **SELECT\_CHANNEL**.

A modifier must also inform 3ds max which channels it will actually alter its function as a modifier. It does this by implementing the method **Modifier::ChannelsChanged()**. Here is the implementation of **ChannelsChanged()** by the Volume Select modifier:

```
ChannelMask ChannelsChanged()  
{
```

```

return
    SELECT_CHANNEL|SUBSEL_TYPE_CHANNEL|GEOM_CH
}

```

This indicates that this modifier alters the selection channel, the sub-object selection channel, and the geometry channel. Note that the geometry channel is modified because the vertices themselves get selected via the **vertSel BitArray** in the **Mesh** class and are thus modified.

### **Objects May Need to Convert Themselves**

Procedural Objects don't always start out in a form suitable for a particular type of modifier. Consider the case of a 3ds max user putting a volume select modifier on a procedural sphere in order to select some vertices within it. A procedural sphere is not defined by vertices and faces. Rather it is defined by its procedural definition, i.e. its radius, segment count, hemisphere setting, etc. But a user can put a volume select modifier on a procedural sphere. The way this is handled is that 3ds max asks the object if it can be converted to a triangle mesh object (which has vertices and faces). If the sphere responds yes, then 3ds max asks the sphere to convert itself to one. Then, once the object is in the form of the triangle mesh, the modifier may alter it. There are two methods that procedural objects implement to allow this to happen. These are **Object::CanConvertToType()** and **Object::ConvertToType()**.

The code executed by the procedural sphere implementation of **CanConvertToType()** is as follows:

```

int SphereObject::CanConvertToType(Class_ID obtype)
{
    if (obtype==patchObjectClassID || obtype==defObjectClassID ||
        obtype==triObjectClassID ||
        obtype==EDITABLE_SURF_CLASS_ID) {
        return 1;
    } else {
        return SimpleObject::CanConvertToType(obtype);
    }
}

```

```

int SimpleObject::CanConvertToType(Class_ID obtype)
{

```

```

if (obtype==defObjectClassID || obtype==mapObjectClassID ||
    obtype==triObjectClassID || obtype==patchObjectClassID) {
    return 1;
}
return Object::CanConvertToType(obtype);
}

```

```

int Object::CanConvertToType(Class_ID obtype)
{
    return obtype==ClassID();
}

```

Note that the sphere answers true when asked if it can convert itself to a patch object, a deformable object (a generic object with points that can be modified), a triangle mesh object, and an editable surf object (NURBS object). In other cases it calls the base class method from **SimpleObject**. If **SimpleObject** doesn't recognize the type it calls its base class method from **Object**. The **Object** implementation returns nonzero if the object is asked to convert to itself, otherwise zero.

When 3ds max needs to actually have the object convert itself to the type required by the modifier it calls **ConvertToType()** on the object as passes the needed **Class\_ID**. Below is the code from the procedural sphere's implementation.

```

Object* SphereObject::ConvertToType(TimeValue t, Class_ID
obtype)
{
    if (obtype == patchObjectClassID) {
        Interval valid = FOREVER;
        float radius;
        int smooth, genUVs;
        pblock->GetValue(PB_RADIUS,t,radius,valid);
        pblock->GetValue(PB_SMOOTH,t,smooth,valid);
        pblock->GetValue(PB_GENUVS,t,genUVs,valid);
        PatchObject *ob = new PatchObject();
        BuildSpherePatch(ob->patch,radius,smooth,genUVs);
    }
}

```

```

ob->SetChannelValidity(TOPO_CHAN_NUM,valid);
ob->SetChannelValidity(GEOM_CHAN_NUM,valid);
ob->UnlockObject();
return ob;
} else if (obtype == EDITABLE_SURF_CLASS_ID) {
Interval valid = FOREVER;
float radius, hemi;
int recenter, genUVs;
pblock->GetValue(PB_RADIUS,t,radius,valid);
pblock->GetValue(PB_HEMI,t,hemi,valid);
pblock->GetValue(PB_RECENTER,t,recenter,valid);
pblock->GetValue(PB_GENUVS,t,genUVs,valid);
Object *ob = BuildNURBSSphere(radius, hemi,
recenter,genUVs);
ob->SetChannelValidity(TOPO_CHAN_NUM,valid);
ob->SetChannelValidity(GEOM_CHAN_NUM,valid);
ob->UnlockObject();
return ob;

} else{
return SimpleObject::ConvertToType(t,obtype);
}
}

```

The cases handled above cover converting to a patch or NURBS representation. The other cases are handled by the base classes.

In summary, an object must implement two methods to allow a modifier to operate upon it. These are **CanConvertToType()** and **ConvertToType()**.

## Types of Object Modification

This section presents information on the general types of modifiers possible in 3ds max and an overview of how they accomplish their modification. Code is presented for each showing the needed input type, channels changed and used, and the method that does the object modification.

There are many types of modifiers possible in MAX. The way they are classified is by which channel(s) of the geometry pipeline they operate on. Modifiers may operate on one or more channels. These channels are things such as geometry, topology, texture coordinates, sub-object selection level, vertex colors, as well as several others. See [List of Channel Bits](#) to review the full list of channels the pipeline is broken down into. The sections below show examples of modifiers that operate on many of these different channels. For each of these an analysis of the critical code fragment that actually handles the alteration of the object is shown -- this is the method **Modifier::ModifyObject()**. This method is defined like this:

### Prototype:

```
virtual void ModifyObject(TimeValue t, ModContext &mc,  
ObjectState* os, INode *node)=0;
```

### Remarks:

This is the method that actually modifies the input object. This method is responsible for altering the object and then updating the validity interval of the object to reflect the validity of the modifier.

### Parameters:

#### **TimeValue t**

The time at which the modification is being done.

#### **ModContext &mc**

A reference to the ModContext. The ModContext stores information about the space the modifier was applied in including.

#### **ObjectState\* os**

The object state flowing through the pipeline. This contains a pointer to the object to modify.

#### **INode \*node**

The node the world space modifier is applied to. This parameter is always

NULL for Object Space Modifiers and non-NULL for World Space Modifiers (Space Warps). This is because a given WSM is only applied to a single node at a time whereas an Object Space Modifier may be applied to several nodes. This may be used for example by particle system space warps to get the transformation matrix of the node at various times.

Below are examples of many of the modifier types.

### **Modifiers Which Change Topology**

These modifiers alter the face or polygon structures of the objects they are applied to. Smoothing groups and materials are also part of the topology channel. Edge visibility is also part of this channels since it is an attribute of the face structure. The face normals are also part of the topology channel. The example below is from the Normals modifier which allows the user to unify and flip the face normals of a mesh. The full source code is available in `\MAXSDK\SAMPLES\MODIFIERS\SURFMOD.CPP`.

Below are the implementations of **InputType()**, **ChannelsUsed()**, and **ChannelsChanged()**.

```
Class_ID InputType() {return triObjectClassID;}  
ChannelMask ChannelsUsed() {return OBJ_CHANNELS;}  
ChannelMask ChannelsChanged() {return  
GEOM_CHANNEL|TOPO_CHANNEL;}
```

Note that this modifier only operates on triangle mesh objects, or those objects able to convert themselves to triangle meshes. Also note that this modifier specifies the object channels for those that it needs up to date to perform its modification. Since this modifier changes the vertex and face structure of the mesh to alter the normals it specifies it modifies the geometry channel and the topology channel in **ChannelsChanged()**.

Here is the implementation of **ModifyObject()**. Note that when the modifier is done it updates the validity of the object flowing down the pipeline by calling **UpdateValidity()** and passing the topology channel number.

```
void NormalMod::ModifyObject(TimeValue t, ModContext &mc,  
    ObjectState *os, INode *node)  
{  
    Interval valid = FOREVER;  
    int flip, unify;
```

```

pblock->GetValue(PB_FLIP,t,flip,valid);
pblock->GetValue(PB_UNIFY,t,unify,valid);

assert(os->obj->IsSubClassOf(triObjectClassID));
TriObject *triOb = (TriObject *)os->obj;
BOOL useSel = triOb->mesh.selLevel==MESH_FACE;

if (unify) {
    triOb->mesh.UnifyNormals(useSel);
}

if (flip) {
    for (int i=0; i<triOb->mesh.getNumFaces(); i++) {
        if (!useSel || triOb->mesh.faceSel[i]) {
            FlipMeshNormal(&triOb->mesh,(DWORD)i);
        }
    }
}

triOb->UpdateValidity(TOPO_CHAN_NUM,valid);
}

```

### **Modifiers Which Change Mapping (Texture Coordinates)**

These modifiers alter the texture coordinates of the objects they modify. These can add new mapping coordinates to unmapped objects or modify the existing mapping coordinates of objects. A modifier that adds new mapping coordinates is the UVW Mapping Modifier. The full source code is available in **\MAXSDK\SAMPLES\MODIFIERS\MAPMOD.CPP**. Below are the implementations of **InputType()**, **ChannelsUsed()**, and **ChannelsChanged()**.

```

Class_ID InputType() { return mapObjectClassID; }
ChannelMask ChannelsUsed()
{
    return PART_GEOM | PART_TOPO | PART_SELECT |
        PART_SUBSEL_TYPE | PART_VERTCOLOR;
}

```

```

    }
ChannelMask ChannelsChanged()
    {
        return TEXMAP_CHANNEL|PART_VERTCOLOR;
    }

```

The mapping modifier requests a modifier that is mappable by specifying the Class\_ID **mapObjectClassID**. Objects that know how to make themselves mappable respond to this Class\_ID in **CanConvertToType()** and return TRUE. This indicates that they may have texture coordinates assigned to them and they implement the method **Object::ApplyUVWMap()**. Note how the Vertex Color channel is needed and changed as well as the texmap channel. This is because 3ds max uses the second mapping channel as the vertex color channel and the user is able to specify that the second mapping channel may be used for the mapping coordinates.

Here is the implementation of **ModifyObject()**. Note that when the modifier is done it updates the validity of the texture map channel by calling **UpdateValidity()**.

```

void MapMod::ModifyObject(TimeValue t, ModContext &mc,
    ObjectState *os, INode *node)
    {
        // If it's not a mappable object then we can't help
        Object *obj = os->obj;
        if (!obj->IsMappable()) return;

        // Get pblock values
        int type, uflip, vflip, wflip, cap, channel;
        float utile, vtile, wtile;
        pblock->GetValue(PB_MAPTYPE,t,type,FOREVER);
        pblock->GetValue(PB_UTILE,t,utile,FOREVER);
        pblock->GetValue(PB_VTILE,t,vtile,FOREVER);
        pblock->GetValue(PB_WTILE,t,wtile,FOREVER);
        pblock->GetValue(PB_UFLIP,t,uflip,FOREVER);
        pblock->GetValue(PB_VFLIP,t,vflip,FOREVER);
        pblock->GetValue(PB_WFLIP,t,wflip,FOREVER);

```

```
pblock->GetValue(PB_CAP,t,cap,FOREVER);  
pblock->GetValue(PB_CHANNEL,t,channel,FOREVER);
```

```
// Prepare the controller and set up mats  
if (!tmControl || (flags&CONTROL_OP) ||  
(flags&CONTROL_INITPARAMS))  
    InitControl(mc,obj,type,t);  
    Matrix3 tm;  
    tm = Inverse(CompMatrix(t,&mc,NULL));
```

```
obj-  
>ApplyUVWMap(type,utile,vtile,wtile,uflip,vflip,wflip,cap,tm,char
```

```
// The tex mapping depends on the geom and topo so make sure  
the validity interval reflects this.
```

```
Interval iv = LocalValidity(t);  
iv = iv & os->obj->ChannelValidity(t,GEOM_CHAN_NUM);  
iv = iv & os->obj->ChannelValidity(t,TOPO_CHAN_NUM);  
os->obj->UpdateValidity(TEXMAP_CHAN_NUM,iv);  
}
```

To see an example of a modifier that alter existing texture coordinates take a look at the source code for the UVW XForm modifier in **\MAXSDK\SAMPLES\MODIFIERS\UVWXFORM.CPP**. or the UVW Unwrap modifier in **\MAXSDK\SAMPLES\MODIFIERS\UNWRAP.CPP**.

### **Modifiers Which Change Materials**

These modifiers alter the material ID stored by the object. An example of this type of modifier is the Material Modifier (**\MAXSDK\SAMPLES\MODIFIERS\SURFMOD.CPP**). Below are the implementations of **InputType()**, **ChannelsUsed()**, and **ChannelsChanged()**. Note that materials are rolled into the face structure of the object so this modifier alters the topology channel.

```
Class_ID InputType() {return triObjectClassID;}  
ChannelMask ChannelsUsed() {return OBJ_CHANNELS;}
```

```
ChannelMask ChannelsChanged() {return  
GEOM_CHANNEL|TOPO_CHANNEL;}
```

This modifier specifies it changes the geometry channel. Technically it doesn't need to since it only changes the material index at the face level and nothing at the vertex level. Again, this is a bit of overkill.

Here is the implementation of **ModifyObject()**. Note that when the modifier is done it updates the validity of the object flowing down the pipeline by calling **UpdateValidity()** and passing the topology channel number.

```
void MatMod::ModifyObject(TimeValue t, ModContext &mc,  
    ObjectState *os, INode *node)  
{  
    Interval valid = FOREVER;  
    int id;  
    pblock->GetValue(PB_MATID,t,id,valid);  
    id--;  
    if (id<0) id = 0;  
    if (id>0xffff) id = 0xffff;  
  
    assert(os->obj->IsSubClassOf(triObjectClassID));  
    TriObject *triOb = (TriObject *)os->obj;  
    BOOL useSel = triOb->mesh.selLevel==MESH_FACE;  
  
    for (int i=0; i<triOb->mesh.getNumFaces(); i++) {  
        if (!useSel || triOb->mesh.faceSel[i]) {  
            triOb->mesh.setFaceMtlIndex(i,(MtlID)id);  
        }  
    }  
  
    triOb->UpdateValidity(TOPO_CHAN_NUM,valid);  
}
```

### **Modifiers Which Change Selection**

These modifiers alter the selection level of objects. An example is the Volume Select modifier. The full source code is available in **\MAXSDK\SAMPLES\MODIFIERS\SELMOD.CPP**.

Below is the implementations of **InputType()**, **ChannelsUsed()**, and **ChannelsChanged()**.

```
Class_ID InputType() {return triObjectClassID;}
ChannelMask ChannelsUsed() {return OBJ_CHANNELS;}
ChannelMask ChannelsChanged()
{
    return
    SELECT_CHANNEL|SUBSEL_TYPE_CHANNEL|GEOM_CHAI
}
```

Note that **ChannelsChanged()** for the Volume Select Modifier doesn't specify **DISP\_ATTRIB\_CHANNEL** in **ChannelsChanged()**. Technically it really should since it changes the Mesh display flags. The reason this isn't a problem however is that this is only a single int in the Mesh and isn't dynamically allocated. So it is actually **always** present even without specifically specifying it.

Here is the implementation of **ModifyObject()**. Note that when the modifier is done it updates the validity of the object flowing down the pipeline by calling **UpdateValidity()** on all the appropriate channels.

```
void SelMod::ModifyObject(TimeValue t, ModContext &mc,
    ObjectState *os, INode *node)
{
    Interval valid = LocalValidity(t);
    int level, method, type, vol, invert;

    pblock->GetValue(PB_LEVEL,t,level,FOREVER);
    pblock->GetValue(PB_METHOD,t,method,FOREVER);
    pblock->GetValue(PB_TYPE,t,type,FOREVER);
    pblock->GetValue(PB_VOLUME,t,vol,FOREVER);
    pblock->GetValue(PB_INVERT,t,invert,FOREVER);

    assert(os->obj->IsSubClassOf(triObjectClassID));
    TriObject *obj = (TriObject*)os->obj;
    Mesh &mesh = obj->mesh;

    // Prepare the controller and set up mats
```

```

if (!tmControl || (flags&CONTROL_OP))
InitControl(mc,obj,type,t);
Matrix3 tm;
tm = Inverse(CompMatrix(t,&mc,NULL,TRUE,FALSE));

Box3 mcbox = *mc.box;
FixupBox(mcbox);

switch (level) {
case SEL_OBJECT:
    obj->mesh.selLevel = MESH_OBJECT;
    obj-
>mesh.ClearDispFlag(DISP_VERTTICKS|DISP_SELVERTS|DISP_S
break;

case SEL_VERTEX:
    obj->mesh.selLevel = MESH_VERTEX;
    obj->mesh.SetDispFlag(DISP_VERTTICKS|DISP_SELVERTS);
    SelectVertices(obj->mesh,method,type,vol,invert,tm,mcbox);
break;

case SEL_FACE:
    obj->mesh.selLevel = MESH_FACE;
    obj->mesh.SetDispFlag(DISP_SELFACES);
    SelectFaces(obj->mesh,method,type,vol,invert,tm,mcbox);
break;
}

obj->UpdateValidity(SELECT_CHAN_NUM,valid);
obj->UpdateValidity(GEOM_CHAN_NUM,valid);
obj->UpdateValidity(TOPO_CHAN_NUM,valid);
obj->UpdateValidity(SUBSEL_TYPE_CHAN_NUM,FOREVER);
}

```

Note how **UpdateValidity()** is called on four channels.

## Modifiers Which Change Geometry

These modifiers just alter the 'points' of the object. Many modifiers in 3ds max do this. For example, Bend, Taper, Twist, Spherify, and Wave all operate on the just the points of objects. Note that the 'points' may mean different things to different objects. For example, a Bend modifier may be applied to both a Cylinder object and a NURBS curve object. That's because they both have 'points' that can be modified, although the nature of the 'points' is quite different between them. Modifiers that operate on these generic points are said to operate on 'deformable' objects.

Modifiers that only alter the geometry channel can be sub-classed from **SimpleMod** rather than **Modifier** and have fewer methods to implement. In this case, the base class **SimpleMod** provides the default implementation of **InputType()**, **ChannelsUsed()** and **ChannelsChanged()**. Here they are:

```
Class_ID InputType() {return defObjectClassID;}  
ChannelMask ChannelsUsed()  
  {  
    return  
PART_GEOM|PART_TOPO|SELECT_CHANNEL|SUBSEL_TYPE_  
  }  
ChannelMask ChannelsChanged() { return PART_GEOM; }
```

Note that **InputType()** return the Class\_ID of deformable objects. As noted above, these are a type of objects that have 'points' to deform. Also notice that **SimpleModifiers** require the geometry, topology, and selection channels. They need the topology channel up to date because these modifiers can work on the selection set. It needs to find the vertices that are selected because they are part of selected faces. To make sure the face selection is up to date the topology channel is specified. The selection channels are required for the same reason -- the modifier may operate on only the selection set if that's what the object flowing down the pipeline is at.

**SimpleModifiers** don't directly implement **ModifyObject()**. Rather they implement a method called **GetDeformer()**. This method returns a callback object which is an instance of the class **Deformer**. This callback object is really like a pointer to a function that is called to perform the alteration of a single point of an object. The **Deformer** callback object returned from **GetDeformer()** has a single virtual method that the plug-in modifier

implements called **Map()**. This method is passed a single point of the object and its job is to modify the point and return it in altered form. To see an example of an implementation of this method see **BendDeformer::Map()** in **\MAXSDK\SAMPLES\MODIFIERS\BEND.CPP**.

The base class **SimpleMod** provides the implementation of **ModifyObject()** and calls a method of the input object called **Deform()** which in turn calls the deformer provided by **GetDeformer()**. Here is the implementation of **SimpleMod::ModifyObject()**:

```
void SimpleMod::ModifyObject(TimeValue t, ModContext &mc,
ObjectState *os, INode *node)
{
    Interval valid = GetValidity(t);
    Matrix3 modmat,minv;

    // These are inverted because that's what is usually needed
    // for displaying/hit testing
    minv = CompMatrix(t,mc,idTM,valid,TRUE);
    modmat = Inverse(minv);

    os->obj->Deform(&GetDeformer(t,mc,modmat, minv), TRUE);
    os->obj->UpdateValidity(GEOM_CHAN_NUM,valid);
}
```

Notice that the **Deform()** method of the input object is called passing the deformer provided by the modifier plug-in. The object that is being modified provides the implementation of **Deform()** (for example a triangle mesh object that is being modified). Here is the default implementation.

```
void Object::Deform(Deformer *defProc,int useSel) {
    int nv = NumPoints();
    for (int i=0; i<nv; i++)
        SetPoint(i,defProc->Map(i,GetPoint(i)));
    PointsWereChanged();
}
```

Notice how this method loops through the points of the object and calls the **Map()** method on each point. The **GetPoint()** method of the object returns the

'i-th' point of the object. This is passed to **Map()** which deforms and returns it. The result returned from **Map()** is set back into the object by calling **SetPoint()**. Then a method called **PointsWereChanged()** is called to let that object know that its points have been altered.

Some objects override the base class definition of this method to provide other ways of modifying the object. For example the **PatchObject** and **SplineShape** objects provide alternate implementations. If an object wanted to modify the selected points only for example, it would need to override **Object::Deform()** and respect the **useSel** flag that indicates it should use the current selection. The base class method shown above ignores that flag. In each case however, they still call the **Map()** method of the deformer passing it each point to modify.

### **Modifiers Which Change The Entire Object**

There are some modifiers that don't alter one or more parts of the object, they completely replace it with a new object. These modifiers effectively convert the object from one form to another. Examples of this are the Extrude and Lathe modifiers. Both Extrude and Lathe convert a spline object into a mesh, patch or NURBS object, thus completely replacing the entire object. The full source code for Extrude is available in

**\MAXSDK\SAMPLES\MODIFIERS\EXTRUDE.CPP.**

Below are the implementations of **InputType()**, **ChannelsUsed()**, and **ChannelsChanged()**.

```
Class_ID InputType() { return genericShapeClassID; }  
ChannelMask ChannelsUsed() { return  
PART_GEOM|PART_TOPO; }  
ChannelMask ChannelsChanged() { return PART_ALL; }
```

Note that the modifier indicates it changes **all** the channels by returning **PART\_ALL** from **ChannelsChanged()**.

Below is a subset of the implementation of **ModifyObject()** (the code for the NURBS and PATCH cases are removed for brevity and simplicity). In the mesh case, the key thing to notice is that a new **TriObject** is created, the TriObject's mesh is created from the spline, and the new object is placed into the pipeline by assigning it's pointer to the **ObjectState**'s object pointer.

```
void ExtrudeMod::ModifyObject(TimeValue t, ModContext
```

```

&mc,
    ObjectState *os, INode *node) {

// Get our personal validity interval...
Interval valid = GetValidity(t);
valid &= os->obj->ChannelValidity(t,TOPO_CHAN_NUM);
valid &= os->obj->ChannelValidity(t,GEOM_CHAN_NUM);

int output;
pblock->GetValue(PB_OUTPUT, TimeValue(0), output,
FOREVER);

switch (output) {
case NURBS_OUTPUT: {
// ...
break;
}
case PATCH_OUTPUT: {
// ...
break;
}
case MESH_OUTPUT: {
// BuildMeshFromShape fills in the TriObject's mesh,
// then we stuff the TriObj into the pipeline.
TriObject *tri = CreateNewTriObject();
BuildMeshFromShape(t, mc, os, tri->mesh);

tri->SetChannelValidity(TOPO_CHAN_NUM, valid);
tri->SetChannelValidity(GEOM_CHAN_NUM, valid);
tri->SetChannelValidity(TEXMAP_CHAN_NUM, valid);
tri->SetChannelValidity(MTL_CHAN_NUM, valid);
tri->SetChannelValidity(SELECT_CHAN_NUM, valid);
tri->SetChannelValidity(SUBSEL_TYPE_CHAN_NUM, valid);
tri->SetChannelValidity(DISP_ATTRIB_CHAN_NUM, valid);

```

```
os->obj = tri;  
break;  
}  
}  
os->obj->UnlockObject();  
}
```

## How Long is the Modification Valid For?

This section presents information on validity intervals. These describe a range of time over which the modification performed by the modifier is accurate. This is needed because so many things in 3ds max can be animated and thus change over time. For example, consider the case above of the procedural sphere and the volume select modifier applied to select a set of vertices. A 3ds max user might animate the number of segments in the sphere as an animated camera gets closer to it -- this keeps the silhouette of the sphere from appearing faceted. The volume select modifier applied to the sphere needs to select all the vertices in the region defined by its gizmo which represents its volume. Since the segment count in the sphere is changing, the number of vertices in the region defined by the modifier's gizmo is changing. Therefore the selection done at one frame may not be valid at the next frame. In order to tell 3ds max how long the modification is valid, what's called a validity interval is computed and returned. This describes the range of time over which the modification is accurate and up to date.

There are two methods that a modifier needs to call or implement. These are **Object::UpdateValidity()** and **Modifier::LocalValidity()**.

When a modifier is finished altering an object it needs to include its interval in the validity interval of the object. This way if an object was static, but had an animated modifier applied to it, 3ds max would know that the modifier would need to be re-evaluated if the user moves to a new time. To do this, the modifier calls the **UpdateValidity()** method on the object, specifying the channel and the modifier's interval. Modifiers that only affect the geometry channel would specify **GEOM\_CHAN\_NUM**. Modifiers that affect other channels would have to call this method once for each channel modified. The interval that is passed in to this method is then intersected with the interval that the object keeps for each channel. So as the object travels through the pipeline, its validity intervals are potentially getting smaller as (possibly) animated modifiers are applied to it.

Here are the calls to **UpdateValidity()** that the Volume Select Modifier makes to the object it modifies. These calls are made inside the modifiers

**ModifyObject()** method.

```
obj->UpdateValidity(SUBSEL_TYPE_CHAN_NUM,FOREVER);  
obj->UpdateValidity(SELECT_CHAN_NUM,valid);  
obj->UpdateValidity(GEOM_CHAN_NUM,valid);
```

**obj->UpdateValidity(TOPO\_CHAN\_NUM,valid);**

The modifier must also implement a method to return to 3ds max it's own validity, independent of the object it is modifying. This method is **Modifier::LocalValidity()**. This value is computed by starting an interval off at **FOREVER**, and intersecting the validity intervals of all the animated parameters of the modifier. In the case of the Volume Select Modifier, it has a single animated parameter -- it's gizmo which represents the actual extents of the volume it selects. Below is the implementation of this method by Volume Select.

**Interval SelMod::LocalValidity(TimeValue t)**

```
{  
Interval valid = FOREVER;  
if (tmControl) {  
Matrix3 tm(1);  
tmControl->GetValue(t,&tm,valid,CTRL_RELATIVE);  
}  
return valid;  
}
```

Note the interval is started as **FOREVER**. Then the gizmo's transform controller's **GetValue()** method is called passing it this initial interval of **FOREVER**. The controller's **GetValue()** method will update this interval to reflect the validity of the gizmo. This interval is then returned. Note that if the volume has not been animated yet, and thus there is not a controller assigned to the gizmo yet, **FOREVER** is returned. This means the modifier is valid at all times. To learn more about validity intervals see the Advanced Topics section [Intervals](#).

Some modifiers are sub-classed from SimpleMod and not Modifier. To return the validity interval of the modifier to **SimpleMod**, the developer must implement a method named **GetValidity()**. **SimpleMod** then provides the implementation of **LocalValidity()** itself but calls **GetValidity()** on the **SimpleModifier**.

## Summary

This section presented information about how objects and modifiers work together to accomplish object modification. Objects may need to convert themselves to another form to be modified. They do this in their **ConvertToType()** method. Modifiers need to specify which object types they operate on (**InputType()**) and which pipeline channels they need and use (**ChannelsUsed()** and **ChannelsChanged()**). Modifiers also perform their modification in the method **ModifyObject()**. Finally, Modifiers must update the validity intervals of the objects they modify to reflect their own validity. This is done inside the method **UpdateValidity()**.

# Palettes

See Also: [Class GPort](#), [Class Quantizer](#), [Class ColorPacker](#).

## Overview

When operating in 8-bit mode, 3ds max uses a 256 color palette. Plug-ins should avoid changing the system palette because this can cause the quality of the 3ds max display to degrade. The palette used by 3ds max reserves the bottom 10 slots and top 10 slots of the palette for Windows colors. It also contains a 6x6x6 RGB color cube, and a 16 gray level ramp. This palette should be adequate for most purposes. When a plug-in wants to output bitmaps and colored areas with dither, so as to get good approximations of 24 bit colors in 8-bit modes, it should make use of the **GPort** class. The **GPort** also provides access to 8 "animated palette slots" to allow dynamically altering system palette entries.

## Support Classes

This section discusses several class that are helpful in working with 3ds max palettes and paletted images.

[Class GPort](#) - This class has several purposes. It maintain the default 3ds max palette for doing 256 color graphics. It also provides a mechanism for allocating "animated color slots" in the default palette for use in the user interface. Additionally it provide various functions for doing dithered graphics using the default 3ds max palette.

There is one instance of **GPort** that is shared globally. It is accessed by the function:

```
GPort* GetGPort();
```

Here is an example of a typical use of this class. In this case the developer wants to put up a dialog box with two color swatches in it, and have these be painted using an "animated color slot" in the default 3ds max palette. To set things up, the following is done in the **WM\_INITDIALOG** code:

```
// Plug the standard MAX palette into the hdc  
HDC hdc = GetDC(hwndDlg);  
hOldPal = GetGPort()->PlugPalette(hdc);  
// Get one anim palette slot  
animSlot[0] = GetGPort()->GetAnimPalSlot();  
// Get another anim palette slot  
animSlot[1] = GetGPort()->GetAnimPalSlot();
```

The routine **GetAnimPalSlot()** is used to get an animatable palette slot. There are a total of 8 slots available. This method may return -1. This indicates that **GPort** has no more remaining animatable palette slots. In this case, using the routine **GPort::PaintColorSwatch()** will do the proper thing, that is dithering the swatch instead of trying to paint it with a single palette index.

Next, in the **WM\_PAINT** code before drawing the following is done:

```
// Just to be safe: if the palette is there, unchanged, this is cheap.  
GetGPort()->PlugPalette(hdc);  
GetGPort()->SetAnimPalEntry(animSlot[0], rgb1); // set an rgb  
into slot  
GetGPort()->SetAnimPalEntry(animSlot[1], rgb2); // set an rgb
```

into slot

```
GetGPort()->AnimPalette(hdc); // update the palette with these colors
```

```
PaintColorSwatch(hdc, rgb1, 0, 10,10,40,20); // see below
```

```
PaintColorSwatch(hdc, rgb2, 0, 10,30,40,40);
```

To finish up, in **WM\_DESTROY** you should do the following:

```
GetGPort()->ReleaseAnimPalSlot(animSlot[0]); // Release palette slot
```

```
GetGPort()->ReleaseAnimPalSlot(animSlot[1]); // Release palette slot
```

```
GetGPort()->RestorePalette(hdc,hOldPal); // Restore palette
```

The code for **PaintColorSwatch()** is also included below to illustrate in more detail how **GPort** works. This is a routine to draw a color swatch using an animated palette entry, if available.

```
void PaintColorSwatch(HDC hdc, DWORD col, int slot,  
int left, int top, int right, int bottom) {  
    HPEN oldPen = (HPEN) SelectObject(hdc,  
GetStockObject(NULL_PEN));  
    HBRUSH oldBrush;  
    if (slot>=0) {  
        HBRUSH brush = GetGPort()->MakeAnimBrush(slot,col);  
        oldBrush = (HBRUSH)SelectObject (hdc, brush);  
        Rectangle (hdc, left,top,right,bottom);  
        DeleteObject(brush);  
    }  
    else {  
        oldBrush = (HBRUSH)SelectObject (hdc,  
        GetStockObject( NULL_BRUSH ));  
        Rectangle (hdc, left, top, right,bottom); // Paint the border  
        Rect rect;  
        rect.left = left++;  
        rect.top = top++;  
        rect.right = right--;  
        rect.bottom = bottom--;
```

```

GetGPort()->DitherColorSwatch(hdc, rect , col);
}
SelectObject (hdc, oldPen);
SelectObject (hdc, oldBrush);
}

```

[Class Quantizer](#) - This class is used for dithering output to files. This class computes a palette when going from 24-bit to 8-bit color.

[Class ColorPacker](#) - This class is used to pack the pixels down using the **Quantizer** computed palette.

The following code shows how the methods of **Quantizer** and **ColorPacker** may be used to compute a palette for an image.

```

/* Make a palette for the given image */
int BitmapIO::CalcOutputPalette(int palsize, BMM_Color_48
*pal) {
    Quantizer *q = BMMNewQuantizer();
    if (!q->AllocHistogram()) {
        bail_out:
        q->DeleteThis();
        return(0);
    }
    PixelBuf line(map->Width());
    if (!line.Ptr())
        goto bail_out;
    int y;
    for(y=0; y<map->Height(); ++y) {
        // call GetOutputPixels to get fileOutputGamma corrected
        values.
        if (!GetOutputPixels(0,y,map->Width(),line.Ptr()))
            goto bail_out;
        q->AddToHistogram(line.Ptr(),map->Width());
    }
    // Reserve the background color (assuming the upper left pixel is
    a
    // representative) and make a palette.

```

```
BMM_Color_64 bgpix;  
GetOutputPixels(0,0,1,&bgpix);  
int n = q->Partition(pal, palsize, &bgpix);  
q->DeleteThis();  
return n;  
}
```

## Parameter Blocks

See Also: [Parameter Blocks and Maps in Release 3 and Later](#), [Parameter Maps](#), [Class IParamBlock](#), [Class ParamDimension](#), [Class GetParamName](#), [Class GetParamDim](#).

## Overview

The parameter block class provides a mechanism for storing values for a plug-ins parameters. When a parameter block is created, the developer specifies the number of parameters and the types of each parameter. Parameter types consist of a range of built in types like integer, float, 3D Point, and Color.

Parameters may be animated or constant. In order for a parameter to be animatable, it must have a controller to control the animation. Different parameter types require different controller types. For example, a floating point value, like the angle parameter for the bend modifier, requires a floating point controller. A node transformation matrix requires a transform controller. The most common controllers are interpolating or 'key frame' controllers.

One of the main purposes of parameter blocks is to manage the complexity of maintaining different controllers for different parameters. To access the values in the parameter block, the plug-in developer uses **GetValue()** and **SetValue()** methods, each take a TimeValue as a parameter. The parameter block stores values in an efficient a manner as possible. If the value hasn't yet been animated, that is, **SetValue()** hasn't been called with time not equal to 0, then a constant value is stored. When **SetValue()** is called with time not equal to 0, and MAXs 'Animate' button is on, a new instance of the default controller for the parameter type is plugged in to the parameter block and initialized with the parameter's values. Plug-ins are required to display their animated parameters in the track view. If all the plug-in's parameters are handled by a parameter block, then the parameter block will take care of this task as well.

## **Parameter Blocks in Use**

In this section we'll explore the definition and creation of parameter blocks. Then we'll look at the method of setting and retrieving values from the parameter block. Finally we'll look at how parameter blocks and references may be used together for processing animated user interface parameters. Note: This section provides only a minimal description of the reference mechanism. For an in depth discussion of references, see the Advanced Topics section [References](#).

## Creating a Parameter Block

Throughout this section an example of a plug-in with three parameters is used. An angle, a length, and an integer count parameter. A parameter block descriptor is used to describe each parameter. It is initialized by passing three arguments per parameter. (Note that another version uses four arguments. The version with four arguments is used to help maintain backward compatibility. The two descriptors are otherwise the same).

### Prototype:

```
class ParamBlockDesc {  
  public:  
    ParamType type;  
    UserType *user;  
    BOOL animatable;  
};  
// This version of the descriptor has an ID for each parameter.  
class ParamBlockDescID {  
  public:  
    ParamType type;  
    UserType *user;  
    BOOL animatable;  
    DWORD id;  
};
```

The arguments to ParamBlockDesc are:

### **ParamType type**

The Parameter Type - The following types may be used:

- **TYPE\_INT** - Integer values.
- **TYPE\_FLOAT** - Floating point value.
- **TYPE\_POINT3** - Point values.
- **TYPE\_RGBA** - Color values - Red, Green, Blue and Alpha.
- **TYPE\_BOOL** - Boolean values.

### **UserType \*user**

The next value is NOT USED - it must always be passed as NULL.

### **BOOL animatable**

This is a flag indicating if the parameter may be animated or not. Pass TRUE

if the value may be animated and FALSE if it is constant.

**DWORD id** (Second version only)

This is an ID used to identify this parameter. This provides a solution to the problem of backwards compatibility. If you alter the parameter structure of your plug-in in the future (by adding or deleting parameter for example) previously saved 3ds max files will be incompatible. You can however use a mechanism which uses these IDs to convert older versions to the current version. See the Advanced Topics section on [Parameter Maps](#) for more detail. Items in the parameter block are referred to by index. The index is derived from the order in which the descriptors appear in the ParamBlockDesc array.

```
// Parameter block indices  
#define PB_INDEX_ANGLE 0  
#define PB_INDEX_LENGTH 1  
#define PB_INDEX_COUNT 2  
ParamBlockDesc pdesc[] = {  
    { TYPE_FLOAT, NULL, TRUE }, // Angle  
    { TYPE_FLOAT, NULL, TRUE }, // Length  
    { TYPE_INT, NULL, FALSE } // Count  
};
```

A parameter block is created from this array by calling **CreateParameterBlock()**. This function requires two values, a pointer to the ParamBlockDesc array and a count of the number of parameters. (Note there is an alternate version for use with the backwards compatibility mechanism discussed above that uses three arguments. The third argument is used to indicate a version of the parameter block. See the Advanced Topics section on [Parameter Maps](#) for more information).

**Prototype:**

```
IParamBlock *CreateParameterBlock(ParamBlockDesc *pdesc,  
int count);
```

or

```
IParamBlock *CreateParameterBlock(ParamBlockDescID *pdesc,  
int count,DWORD version);
```

```
IParamBlock *pblk = CreateParameterBlock(pdesc, 3);
```

The function returns a pointer to the parameter block it creates.

## Retrieving Parameter Block Values

Whenever the developer needs to retrieve a value from the parameter block, the **GetValue()** method is used. There are overloaded functions for each type of value to retrieve (int, float, Point3, and RGBA). Each method has four parameters. Below is the float version.

```
BOOL GetValue( int i, TimeValue t, float &v,  
Interval &ivalid );
```

The **i** parameter is the integer index of the parameter to retrieve. This is the index into the ParamBlockDesc array.

If the parameter is animated it will be varying over time. The **t** parameter specifies at what time to retrieve the value.

The **v** parameter is a C++ reference to a float. The value is returned through **v**.

The **ivalid** parameter is a C++ reference to an Interval. The **GetValue()** method updates the interval passed in. This method is frequently used by developers to 'whittle' down an interval. When a parameter of a parameter block is animated, for any given time there is a interval over which the parameter is constant. If the parameter is constantly changing the interval is instantaneous. If the parameter does not change for a certain period the interval will be longer. If the parameter never changes the interval will be FOREVER. By passing an interval to the **GetValue()** method you ask the parameter block to 'intersect' the interval passed in with the interval of the parameter. Intersecting two intervals means returning a new interval whose start value is the greater of the two, and whose end value is smaller of the two. In this way, the resulting interval represents a combined period of constancy for the two intervals.

This technique is used frequently to compute a validity interval for an object. The developer starts an interval off as FOREVER, then intersects this interval with each of its animated parameters (by calling **GetValue()**). **GetValue()** 'whittles' down the interval with each call. When all the parameters have been intersected the result is the overall validity interval of an object at a specific time. For more information see the Advanced Topics section on [Intervals](#).

The return value is TRUE if a value was retrieved. Otherwise it is FALSE.

## Setting Parameter Block Values

Whenever the developer needs to store a value into the parameter block, the **SetValue()** method is used. There are overloaded functions for each type of value to set (int, float, Point3, and RGBA). Each method has three parameters. Below is the float version.

**BOOL SetValue( int i, TimeValue t, float v );**

The **i** parameter is the integer index of the parameter to set. This is the index into the ParamBlockDesc array of the parameter.

The **t** parameter specifies at what time to set the value.

The value to store is passed in **v**.

The return value is TRUE if the value was set. It is FALSE otherwise.

## Setting Values and Handling References

The **IParamBlock** class is derived from class **ReferenceTarget**. Since parameter blocks are reference targets, plug-ins may create references to them. By creating a reference to the parameter block, the plug-in may be notified whenever any of the parameters are set to new values.

The sample code below creates a parameter block from the array of descriptors and then makes a reference to the parameter block.

```
//This is the index of the Param Blk reference.  
#define PARAM_BLK_REF 0  
MakeRefByID(  
    FOREVER, // Interval - always use FOREVER.  
    PARAM_BLK_REF, // Index of the PB reference.  
    CreateParameterBlock(pdsc, 3) // Create the PB  
);
```

When values of the parameter block are changed, the plug-in needs to be informed. This notification is done by calling the ReferenceMaker method **NotifyDependents()**. The code fragment below shows how this is done. Whenever our sample plug-in needs to adjust its angle parameter it calls the function below. This function calls the **SetValue()** method of the parameter block and then calls **NotifyDependents()**.

```
void Sample::SetAngle(TimeValue t, float r)  
{  
    // pblock is a pointer to our param blk  
    pblock->SetValue(PB_INDEX_ANGLE, t, r);  
    NotifyDependents(FOREVER, PART_OBJ,  
    REFMSG_CHANGE);  
}
```

When the **NotifyDependents()** method is called after a parameter has been changed, the plug-in is notified via a **REFMSG\_CHANGE** message sent to its **NotifyRefChanged()** method. Here the developer may respond to the change in the parameter's value.

There are two other messages which may be sent to this method to handle parameter block processing. These are **REFMSG\_GET\_PARAM\_NAME** and **REFMSG\_GET\_PARAM\_DIM**.

The message **REFMSG\_GET\_PARAM\_DIM** is sent by the system when it needs a parameter dimension for the parameter. Any parameter that can be controlled by a controller has a dimension. This dimension can be considered a unit of measure. It describes its type and its order of magnitude. When a controller needs to display the parameter values (for example in the function curve editor) it converts the value using its parameter dimension **Convert()** function. It can also convert back using the **Unconvert()** function.

There are several default parameter dims implemented. See [ParamDimension](#) for details.

The **REFMSG\_GET\_PARAM\_NAME** messages is sent by the system when it needs a name for the parameter. For example, in the track view a name for the parameter needs to be displayed. When this message is received from the system the name is provided. The parameter which it needs the name for is passed in the **partID**.

The plug-in implements this method to receive messages. See below for how the messages associated with the parameter block are handled.

```
RefResult Sample::NotifyRefChanged(  
Interval changeInt,  
RefTargetHandle hTarget,  
PartID& partID,  
RefMessage message)  
{  
  switch (message) {  
    case REFMSG_CHANGE:  
      // Code to handle the changed parameter...  
      break;  
    case REFMSG_GET_PARAM_DIM:  
      // When a client of a parameter block receives the  
      // REFMSG_GET_PARAM_DIM message, the partID  
      // field is set to point at one of these structs.  
      // The client should set dim to point at its dim descriptor.  
      GetParamDim *gpd = (GetParamDim*)partID;  
      // Based on which index it wants...  
      switch (gpd->index) {
```

```

// ...provide the appropriate dim
case PB_INDEX_ANGLE:
    gpd->dim = stdAngleDim;
    break;
case PB_INDEX_LENGTH:
    gpd->dim = stdWorldDim;
    break;
case PB_INDEX_COUNT:
    gpd->dim = stdSegmentsDim;
    break;
}
return REF_STOP;
case REFMSG_GET_PARAM_NAME:
    // When a client of a parameter block receives the
    // REFMSG_GET_PARAM_NAME message, the
    // partID field is set to point at one of these
    // structures. The client should fill in the parameter name.
    GetParamName *gpn = (GetParamName*)partID;
    switch (gpn->index) {
    case PB_INDEX_ANGLE:
        gpn->name = _T("Angle");
        break;
    case PB_INDEX_LENGTH:
        gpn->name = _T("Length");
        break;
    case PB_INDEX_COUNT:
        gpn->name = _T("Count");
        break;
    }
    return REF_STOP;
}
return(REF_SUCCEED);
}

```

See also: [GetParamName](#), [GetParamDim](#).

# Parameter Editing in the Command Panel

See Also: [Class Animatable](#), [Custom Controls](#), [Parameter Maps](#).

## **Overview**

This section discusses the editing of an item's parameters in the command panel. Plug-In types such as procedural objects, modifiers, space warps, and controllers may all present their user interface in the command panel.

## Methods Called During Parameter Editing

There are two methods that are called when a user begins and ends editing an item's parameters. In the case of procedural objects, modifier, and controllers, these methods are from class **Animatable**. They are named **BeginEditParams()** and **EndEditParams()**. Other plug-in types, such as utilities, may have their own version of these methods (for instance **UtilityObj::BeginEditParams()**). The text below discusses the more common **Animatable** version.

### Beginning Parameter Editing

When the user begins to edit an item the **BeginEditParams()** method is called on it. At this point it is expected to do two things: Put up its user interface via rollup pages, and register any sub-object selection types it may have. Each of these is discussed below.

Adding rollup pages can be done using the method **Interface::AddRollupPage()** or by creating a parameter map (**CreateCPPParamMap()**) which puts up the rollup pages itself.

The **BeginEditParams()** prototype looks like this:

```
virtual void BeginEditParams(IObjParam *ip,ULONG flags,  
Animatable *prev=NULL);
```

The **flags** parameter passed is used to indicate which branch of the command panel the item is being edited in. For instance it may be the Create branch, the Modifier branch, the Hierarchy branch, or the Motion branch.

Some items may have specific parameters that are particular to the creation process. The **flags** parameter indicates whether the user is in the create branch or not. For example, procedural spheres have an option to create them by dragging out the radius or the diameter. These parameters shouldn't be displayed in the Modify branch since they are only for creating. The sphere checks the flag to see if rollup should be added. Here's the code fragment from the sphere code where that is done:

```
void SphereObject::BeginEditParams(IObjParam *ip,ULONG  
flags,Animatable *prev) {  
    // ...  
  
    if (flags&BEGIN_EDIT_CREATE) {
```

```
pmapCreate = CreateCPPParamMap(
    descCreate,CREATEDESC_LENGTH,
    this,
    ip,
    hInstance,
    MAKEINTRESOURCE(IDD_SPHEREPARAM1),
    GetString(IDS_RB_CREATIONMETHOD),
    0);
```

```
pmapTypeIn = CreateCPPParamMap(
    descTypeIn,TYPEINDESC_LENGTH,
    this,
    ip,
    hInstance,
    MAKEINTRESOURCE(IDD_SPHEREPARAM3),
    GetString(IDS_RB_KEYBOARDENTRY),
    APPENDROLL_CLOSED);
}
```

```
pmapParam = CreateCPPParamMap(
    descParam,PARAMDESC_LENGTH,
    pblock,
    ip,
    hInstance,
    MAKEINTRESOURCE(IDD_SPHEREPARAM2),
    GetString(IDS_RB_PARAMETERS),
    0);
}
```

Note above how the 'Creation Method' or 'Keyboard Entry' rollups are only added if the **BEGIN\_EDIT\_CREATE** flag is set. In either case the 'Parameters' rollup is added.

The following bit flags may be compared with the **flag** parameter to test the branch (or sub-task of the branch).

**BEGIN\_EDIT\_CREATE**

Indicates the item is being edited in the create branch.

### **BEGIN\_EDIT\_MOTION**

Indicates a controller is being edited in the motion branch.

### **BEGIN\_EDIT\_HIERARCHY**

Indicates a controller is being edited in the Pivot subtask of the hierarchy branch.

### **BEGIN\_EDIT\_IK**

Indicates a controller is being edited in the IK subtask of the hierarchy branch.

### **BEGIN\_EDIT\_LINKINFO**

Indicates a controller is being edited in the Link Info subtask of the hierarchy branch.

Inside **BeginEditParams()** the item is also expected to register any sub-object selection levels with the system. . For example, a modifier plug-in may have sub-object selection levels like 'Gizmo' and 'Center'. These need to appear in the sub-object drop down allowing the user to select them. In the case of a bend modifier for example, this is accomplished by calling **BeginEditParams()** on the base class **SimpleMod**, i.e.:

```
void BendMod::BeginEditParams(IObjParam *ip, ULONG flags, Animatable *prev) {  
    SimpleMod::BeginEditParams(ip,flags,prev);  
    // ...
```

What the base class **SimpleMod** does is the following:

```
void SimpleMod::BeginEditParams(IObjParam *ip, ULONG flags, Animatable *prev){  
    // ...  
    // Add our sub object type  
    TSTR type1(GetResString(IDS_RB_APPARATUS) );  
    TSTR type2(GetResString(IDS_RB_CENTER) );  
    const TCHAR *ptype[] = { type1, type2 };  
    ip->RegisterSubObjectTypes( ptype, 2 );  
    // ...
```

Note how it retrieves two string resources using **GetResString()** ("Gizmo" and "Center") and registers them using **Interface::RegisterSubObjectTypes()**.

## Ending Parameter Editing

When the user is finished editing the item's parameters its **EndEditParams()** method is called. For example, if the user switches branches of the command panel, or selects a different object type to be created, it's called on the item currently being edited.

The prototype looks like this:

```
virtual void EndEditParams(IObjParam *ip, ULONG flags,  
Animatable *next=NULL);
```

This method is also passed a **flag** value. If the **END\_EDIT\_REMOVEUI** flag is set to the item should remove its rollup pages from the command panel. For example, when in the create branch, procedural objects can be created one after the other so there is no need to remove the UI when one instance of a class is finished with its create stage if another instance of that same class is about to be created. In such a case, **END\_EDIT\_REMOVEUI** won't be set since the user interface should stay in place.

## User Interface Related Variables

The best way to handle user interface-related variables is to make them [= 4\) BSPSPopupOnMouseOver\(event\);">class variables](#) (by declaring them static). Class variables are **shared** by every instance of the class (rather than stored with each instantiated object of the class). This way, an object doesn't suffer the overhead of these variables that are only used when it is editing. Only one item may have its parameters being edited at any one time so the class variables work well.

For example, consider a procedural sphere. It may have class variables that contain the handles to its parameter maps in the command panel. When a sphere is first created, it is immediately edited so its **BeginEditParams()** method is called. It creates the parameter maps and stores the handles to them in its class variables. If the user chooses to make another sphere, the first sphere's **EndEditParams()** method is called, but **END\_EDIT\_REMOVEUI** flag is set to FALSE, indicating that the sphere should leave the rollup pages in the command panel. A new instance of the Sphere class is then created and its **BeginEditParams()** method is called. It notices that the parameter map handles are not NULL so it doesn't need to create them. For instance, consider how the procedural sphere checks this in the code below. If its pointer to the 'Creation Method' and 'Parameters' parameter maps are still valid (non-NULL) it

doesn't recreate the rollup pages. Rather it simply updates the parameter block pointer stored by the parameter map to point to the new sphere object.

```
if (pmapCreate && pmapParam) {  
  // Left over from last sphere ceated  
  pmapCreate->SetParamBlock(this);  
  pmapTypeIn->SetParamBlock(this);  
  pmapParam->SetParamBlock(pblock);  
} else {  
  // . . .
```

### **Interface Pointer Validity**

The interface pointer passed to **BeginEditParams()** is usually stored by the plug-in since it is likely that the plug-in will need it to call its methods. This pointer is only valid while the item is being edited. Therefore, another important thing to do when **EndEditParams()** is called is to set the object's interface pointer to NULL. This pointer is only valid between the **BeginEditParams()** and **EndEditParams()** methods. A plug-in should not call methods on this pointer outside of this interval.

## Parameter Maps

See Also: [Parameter Blocks and Maps in Release 3 and Later](#), [Class IParamMap](#), [Class IParamArray](#), [Class IParamBlock](#), [Class ParamBlockDescID](#), [Class ParamUIDesc](#).

## Overview

Parameter Maps are used to minimize the programming effort required to manage the user interface parameters of a plug-in. A simple plug-in, such as the procedural sphere, has a user interface consisting of controls like spinners, radio buttons and check boxes. Each of these controls has a one-to-one correspondence with a variable or parameter in a parameter block. Parameter maps can be used to map the UI controls to a parameter handler of the appropriate data type. The use of parameter maps has several advantages:

For standard processing, the developer will not need to write message-processing code to handle events generated by the user. For example, if a developer does not use parameter maps they must provide a dialog proc which processes the messages sent by the controls as the user works with them. For instance the spinner control sends a message when the user operates the up and down arrows or enters new values. Parameter maps handle this message processing internally.

The developer is freed from dealing with the complexity of having to manage controllers to control the animation of user interface values. The appropriate type of controller is assigned and managed by the parameter block which the parameter map controls. The developer simply uses **GetValue()** and **SetValue()** methods to retrieve and store values.

Undo and Redo are handled automatically for changes to a parameter's value. The Loading and Saving of the parameters to and from disk is handled automatically.

The following user interface controls are available for use with parameter maps:

## Spinner Custom Control



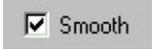
The spinner control is used to provide input of values limited to a fixed type. For example, the control may be limited to the input of only positive integers. The input options are integer, float, universe (world space units), positive integer, positive float, positive universe, and time. This control allows the user to increment or decrement a value by clicking on the up or down arrows. The user may also click and drag on the arrows to interactively adjust the value. The Ctrl key may be held to accelerate the value changing speed, while the Alt key may be held to decrease the value changing speed.

## Radio Button



Radio buttons are used to provide the user with a single boolean choice, or when used in groups, to select among several options.

## Single Check Box



Check boxes are used to provide the user with a single boolean choice.

## **Multiple Check Box**

Multiple Check boxes are also supported. This interface allows each bit of a single integer to control a different check box.

## Color Swatch Custom Control



The Color Swatch control presents the user with the standard 3ds max modeless color selector when the user clicks on the control. The color swatch control displays the currently selected color.

Note that you may use a combination of parameter maps and other techniques to manage your plug-in's user interface. If you don't specifically inform the parameter map about a UI control, it will be ignored. If your plug-in has UI controls that require special processing, a mechanism is provided to allow you to do so. If you wish to use other types of controls such as custom buttons, toolbars or image controls see the Advanced Topics section [Custom Controls](#)

## Basic Concepts of Parameter Maps

A parameter map may be considered a table of parameter handlers. The parameter map provides the processing required to map the UI controls to the variables these controls affect. The parameter map operates on a [= 4\) BSPSPopupOnMouseOver\(event\);">virtual array](#) of parameters. Each control is one element of the virtual array. Each rollup page in the command panel is treated as its own virtual array.

For processing parameter blocks, the parameter map manages everything. As a user operates the UI controls of the plug-in, the parameter map processes the messages sent by the controls and stores any values in the parameter block which need to be set.

For processing other variables, the developer needs to provide a way for the parameter map to get and set the i-th element of the virtual array. To do this, the developer assigns an integer index to each parameter. The developer also implements a set of **GetValue()** and **SetValue()** methods. One parameter to these methods is an index to which parameter to get or set. If the parameter map needs the i-th variable to be stored, the developer stores it. If the parameter map needs to retrieve the i-th value the developer supplies it. A section below discusses how this is done.

These are the primary classes involving parameter maps the developer should be aware of:

[Class ParamUIDesc](#) - These are descriptors that define the properties of the UI control such as its type (spinner, radio button, check box, etc.), which resource ID it refers to (which control in the dialog template), and which index into the virtual array to use.

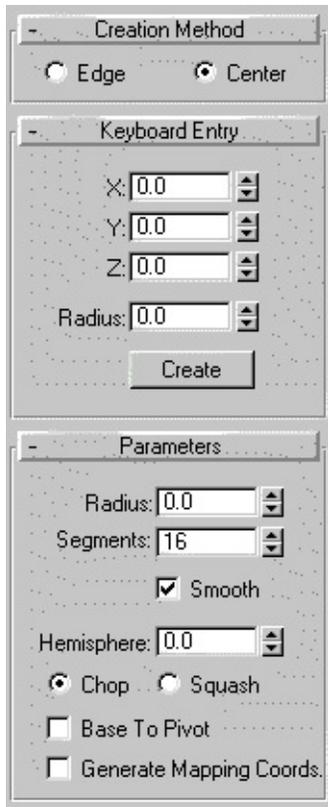
[Class ParamBlockDescID](#) - This class describes each parameter of a parameter block by its type (int, float, point, color), whether it may be animated or is constant, and a ID used to help manage backwards compatibility of parameter blocks.

[Class IParamArray](#) - This is the base class from which parameter maps and parameter blocks are derived. This class represents a virtual array of parameters.

[Class IParamMap](#) - This class provides a set of methods to work with the parameter map, for example, retrieving a pointer to the parameter block used.

## Using Parameter Maps

This section discusses how to use parameter maps to manage the user interface parameter of a plug-in. The example of the procedural sphere object is examined to see how it uses parameter maps. Parameter maps handle most of the command panel user interface processing of the procedural sphere. Shown below are the three rollup pages of the command panel added by the procedural sphere. Except for the "Create" button, all the controls are managed by the parameter map.



The sphere object has two types of parameters managed by parameter maps.

One type of parameter is a simple non-animated value which is driven by a control in the user interface. An example of this is the value associated with the creation method radio buttons in the UI of the sphere. These buttons control if the sphere is created by dragging its radius or its diameter. This is simply a variable which contains the number of the radio button selected by the user (0 or 1). The parameter map allows the variable to be updated as the user chooses radio buttons.

Another type of value is a parameter of a parameter block. Parameter blocks are the mechanism used by 3ds max to manage the animation of parameters. If a parameter is animated, it must have a controller to control the animation.

For example, a floating point value, like the radius of the sphere, uses a floating point controller. The primary purpose of parameter blocks is to manage the complexity of handling different controllers for different parameters. The parameter map updates parameter in the parameter block resulting from any user input, like adjusting the radius spinner.

The following is a description of how the procedural sphere uses parameter maps. It describes how the parameter map was set up and initialized.

The SphereObject class is sub-classed from **IParamArray**.

```
class SphereObject : public GenSphere, public IParamArray {  
    ...  
};
```

This is done so the sphere may use the virtual array mechanism to allow the parameter map to work with its variables. The details of the way the parameter map access these variables is described later in this section.

## Declare the UI Variables

The developer must declare several [= 4\)](#) [BSPSPopupOnMouseOver\(event\);">class variables](#) which are pointers to IParamMaps. Each parameter map manages a **single** rollup page in the command panel.

```
static IParamMap *pmapCreate;  
static IParamMap *pmapParam;
```

The developer must also declare class variables needed for the parameters in the user interface.

```
static int dlgSegments;  
static int dlgCreateMeth;  
static int dlgSmooth;  
static Point3 crtPos;  
static float crtRadius;
```

## Describe the Controls

The developer must define the **ParamUIDesc** arrays to establish the properties of the UI controls such as their type (spinner, radio button, check box, etc.), which resource ID they refer to, and which index into the virtual array they use. Below is the **ParamUIDesc** array for the Parameters rollup page. It calls several overloaded constructors passing the values needed to describe the controls. See the [ParamUIDesc](#) Reference section for detailed information on the constructors.

```
static int squashIDs[] = {IDC_HEMI_CHOP, IDC_HEMI_SQUASH};
static ParamUIDesc descParam[] = {
    // Radius
    ParamUIDesc(
        PB_RADIUS, // Virtual array index
        EDITTYPE_UNIVERSE, // Type of value to edit
        IDC_RADIUS, IDC_RADSPINNER, // Resource IDs
        MIN_RADIUS, MAX_RADIUS, // Upper and lower limits on the
value
        SPIN_AUTOSCALE), // Scale factor for up/down arrow clicks
    // Segments
    ParamUIDesc(
        PB_SEGS,
        EDITTYPE_INT,
        IDC_SEGMENTS, IDC_SEGSPINNER,
        (float)MIN_SEGMENTS, (float)MAX_SEGMENTS,
        0.1f),
    // Smooth
    ParamUIDesc(PB_SMOOTH, TYPE_SINGLECHECKBOX, IDC_OBSM
    // Hemisphere
    ParamUIDesc(
        PB_HEMI,
        EDITTYPE_FLOAT,
        IDC_HEMISPHERE, IDC_HEMISPHERESPINNER,
        0.0f, 1.0f,
        0.005f),
    // Chop/squash
    ParamUIDesc(PB_SQUASH, TYPE_RADIO, squashIDs, 2),
```

```
// Recenter  
ParamUIDesc(PB_RECENTER,TYPE_SINGLECHECKBOX,IDC_HE  
};  
#define PARAMDESC_LENGTH 6
```

## Set Up the Parameter Block

Several of the controls used allow animated values. For example, the radius, and segments parameters may be animated. The developer uses a parameter block to store these values. The developer must define the parameter block descriptor which describes the properties of each parameter.

### Prototype:

```
class ParamBlockDescID {  
    public:  
    ParamType type;  
    UserType *user;  
    BOOL animatable;  
    DWORD id;  
};
```

This class is initialized by passing four values per parameter. These values are:

### **ParamType type**

The Parameter Type - The following are the types which may be used:

- **TYPE\_INT** - Integers values.
- **TYPE\_FLOAT** - Floating point values.
- **TYPE\_POINT3** - Point values.
- **TYPE\_RGBA** - Colors values - Red, Green, Blue and Alpha.
- **TYPE\_BOOL** - Boolean values.

### **UserType \*user**

This value is NOT USED - it must always be passed as NULL.

### **BOOL animatable**

This is a flag indicating if the parameter may be animated or not. Pass TRUE if the value may be animated and FALSE if just a constant value should be stored.

### **DWORD id**

This is an ID assigned to each parameter. This is used for backwards compatibility if you change the parameter block structure in the future. There is a mechanism that allows older format parameter blocks to be converted to a newer format using these IDs to match corresponding parameters between the new and old format. This is described below under Backwards Compatibility.

## Create an Array of Descriptors

The developer must create an array of the descriptors. Items in the parameter block are referred to by index. The index is derived from the order in which the descriptors appear in the ParamBlockDescID array.

```
#define PB_RADIUS 0  
#define PB_SEGS 1  
#define PB_SMOOTH 2  
#define PB_HEMI 3  
#define PB_SQUASH 4  
#define PB_RECENTER 5  
static ParamBlockDescID descVer1[] = {  
  { TYPE_FLOAT, NULL, TRUE, 0 }, // Radius  
  { TYPE_INT, NULL, TRUE, 1 }, // Segs  
  { TYPE_INT, NULL, TRUE, 2 }, // Smooth  
  { TYPE_FLOAT, NULL, TRUE, 3 }, // Hemi  
  { TYPE_INT, NULL, FALSE, 4 }, // Squash  
  { TYPE_INT, NULL, FALSE, 5 } }; // Recenter  
#define PBLOCK_LENGTH 6
```

For more detailed information see the Advanced Topics section [Parameter Blocks](#).

## Create the Parameter Block and Make a Reference to it

A parameter block must be created from the parameter block descriptors. This is done in the constructor of the sphere object. A reference is made to the parameter block as well. For more information on references see the Advanced Topic section [References](#). After the parameter block is created, default values are initialized. This is done using the **SetValue()** method of the parameter block. The developer must pass the index of the parameter, the time to set the value at, and the value to set.

```
SphereObject::SphereObject()  
{  
  MakeRefByID(FOREVER, 0,  
    CreateParameterBlock(descVer1, PBLOCK_LENGTH,  
      CURRENT_VERSION));  
  assert(pblock);  
  pblock->SetValue(PB_RADIUS,0,crtRadius);  
  pblock->SetValue(PB_SMOOTH,0,dlgSmooth);  
  pblock->SetValue(PB_SEGS,0,dlgSegments);  
  pblock->SetValue(PB_SQUASH,0,0);  
}
```

## Add the Rollup Page to the Command Panel

When the user may edit the sphere's parameters the developer must add the rollup pages to the command panel. This is done using the **CreateCPPParamMap()** method. This method creates a parameter map to handle the display of parameters in the command panel. Shown below are two samples from the sphere's **BeginEditParams()** method.

This first call to **CreateCPPParamMap()** manages the variables of the SphereObject (not the parameter block).

The **CreateCPPParamMap()** method takes several arguments. The first is the array of **ParamUIDescs**, one element for each control to be managed. The second is the number of items in this array. The third parameter is a pointer to the virtual array of parameters. The example below uses the **this** pointer of the sphere object to indicate which parameter array. The **this** pointer means the **SphereObject** itself is the **IParamArray** pointer (the **SphereObject** was derived from **IParamArray**). In this way, the parameter map may access the variables of the sphere object. The fourth parameter is the interface pointer passed into the **BeginEditParams()** method. The fifth parameter is the DLL instance handle of the plug-in. The sixth parameter is the dialog template for the rollup page (created using the resource editor). The next parameter is the title displayed in the rollup page title bar. The final parameter is a set of flags to control settings of the rollup page. After this call finishes, the "Creation Method" rollup has been added to the command panel.

```
pmapCreate = CreateCPPParamMap(
    descCreate,CREATEDESC_LENGTH,
    this,
    ip,
    hInstance,
    MAKEINTRESOURCE(IDD_SPHEREPARAM1),
    _T("Creation Method"),
    0);
```

The example below is similar, however it uses the parameter block pointer **pblock** to indicate which virtual array to manage. The parameters in this case are all stored as part of the parameter block (the **IParamBlock** class is derived from **IParamArray**). After this call finishes, the "Parameters" rollup has been

added to the command panel.

```
pmapParam = CreateCPPParamMap(  
descParam,PARAMDESC_LENGTH,  
pblock,  
ip,  
hInstance,  
MAKEINTRESOURCE(IDD_SPHEREPARAM2),  
_T("Parameters"),  
0);
```

## Allowing the Parameter Map to Access plug-in Variables

The developer must implement the **GetValue()** and **SetValue()** methods of the **IParamArray** class to allow the parameter map to manage the **SphereObject** variables. The virtual array mechanism works by using an index to specify which parameters are to be retrieved and set. Shown below are samples for the integer creation method variable. Each method uses a switch statement that checks the index into the virtual array and gets or sets the appropriate variable. Note that the way the developer assigns the index to the variables is important since this is an index into the virtual array. Start at 0 and assign each one 0, 1, 2, 3...

```
BOOL SphereObject::SetValue(int i, TimeValue t, int v)
{
  switch (i) {
    case PB_CREATEMETHOD: dlgCreateMeth = v; break;
  }
  return TRUE;
}
BOOL SphereObject::GetValue(int i, TimeValue t, int &v, Interval
&ivalid)
{
  switch (i) {
    case PB_CREATEMETHOD: v = dlgCreateMeth; break;
  }
  return TRUE;
}
```

## Removing the Rollup Page from the Command Panel

When the user is done editing the sphere's parameters, the **DestroyCPParamMap()** method is used to remove the rollup page from the command panel and release the controls associated with the parameter map. It is called from the **EndEditParams()** method.

**Prototype:**

```
void DestroyCPParamMap(IParamMap *m);  
if (pmapCreate) DestroyCPParamMap(pmapCreate);  
DestroyCPParamMap(pmapParam);  
pmapCreate = NULL;  
pmapParam = NULL;
```

## Processing Controls not managed by the Parameter Map

Some controls may need to be processed by the developer directly. The sphere has a 'Create' button in the 'Keyboard Entry' rollout. Button controls are not processed by parameter maps. In order to process the messages sent when the user operates the button, the developer needs to derive a class from [ParamMapUserDlgProc](#) and set it as the parameter map's user [= 4](#)) [BSPSPopupOnMouseOver\(event\);">callback](#) using **SetUserDialogProc()**.

```
class SphereTypeInDlgProc : public ParamMapUserDlgProc {
public:
    SphereObject *so;
    SphereTypeInDlgProc(SphereObject *s) {so=s;}
    BOOL DlgProc(TimeValue t,IParamMap *map,
        HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam);
    void DeleteThis() {delete this;}
};
```

The **SetUserDlgProc()** method of [IParamMap](#) is used to set the callback to handle the messages. Note that the callback is called after the default processing is complete.

```
pmapTypeIn->SetUserDlgProc(new SphereTypeInDlgProc(this));
```

For more information on Dialog Procs see the Advanced Topics section on [Custom Controls](#).

## Providing a Name and Dimension for the Parameter Block Parameters

The developer must provide a name for each parameter used in a parameter block. This name is needed because the animated parameters show up in the track view and must be labeled. The parameter block takes care of displaying them in the track view, but has no idea of the name for each one, it only knows them by index. The developer provides the name to display by implementing the method **GetParameterName()** and returning the name of the parameter whose index into the parameter block is passed.

```
TSTR SphereObject::GetParameterName(int pbIndex)
{
  switch (pbIndex) {
    case PB_RADIUS:
      return TSTR(_T("Radius"));
    case PB_HEMI:
      return TSTR(_T("Hemisphere"));
    case PB_SEGS:
      return TSTR(_T("Segments"));
    case PB_SMOOTH:
      return TSTR(_T("Smooth"));
    default:
      return TSTR(_T(""));
  }
}
```

The developer must provide a dimension for each parameter used in a parameter block. This dimension is basically the type and magnitude of the value stored in the parameter block. Pre-defined constants are used as the return values. For more information, see the Reference section [ParamDimension](#).

```
ParamDimension *SphereObject::GetParameterDim(int pbIndex)
{
  switch (pbIndex) {
    case PB_RADIUS:
      return stdWorldDim;
    case PB_HEMI:
      return stdNormalizedDim;
    case PB_SEGS:
      return stdSegmentsDim;
  }
}
```

```
case PB_SMOOTH:  
    return stdNormalizedDim;  
default:  
    return defaultDim;  
}  
}
```

## Backwards Compatibility

As development evolves on a plug-in, the use of its parameter blocks may change. For example, more parameters may be added or some may be deleted. 3ds max files that were saved using the old version of the plug-in would not normally load properly under the new format. In order to provide backward compatibility with the older format a mechanism exists to allow the old format to be converted to the new format.

To accomplish the conversion you must define both the old and the new format of the parameter block.

```
static ParamBlockDescID descVer0[] = {
    { TYPE_FLOAT, NULL, TRUE, 0 },
    { TYPE_INT, NULL, TRUE, 1 },
    { TYPE_INT, NULL, TRUE, 2 } };
static ParamBlockDescID descVer1[] = {
    { TYPE_FLOAT, NULL, TRUE, 0 },
    { TYPE_INT, NULL, TRUE, 1 },
    { TYPE_INT, NULL, TRUE, 2 },
    { TYPE_FLOAT, NULL, TRUE, 3 },
    { TYPE_INT, NULL, FALSE, 4 },
    { TYPE_INT, NULL, FALSE, 5 } };
#define PBLOCK_LENGTH 6
```

The fourth argument of the **ParamBlockDescID** is the ID of the parameter. These are used to match the old version of the parameter to the new version. The code which does the conversion matches the IDs between the two versions, for example, it will match the ID=1 of the old version to the ID=1 of the new version.

You create an array of the class **ParamVersionDesc**, one element for each old version of the parameter block. This structure describes a version of the parameter block. You also create an instance of **ParamVersionDesc** for the new version.

```
// Array of old versions
static ParamVersionDesc versions[] = {
    ParamVersionDesc(descVer0,3,0)
};
#define NUM_OLDVERSIONS 1
```

```
// Current version
static ParamVersionDesc curVersion(descVer1,PBLOCK_LENGTH,1);
#define CURRENT_VERSION 1
```

When the plug-in is loaded, its **Load()** method is called. From within this method you call a method of the **ILoad** class which registers the parameter block post load callback object. The callback object is an instance of the class **ParamBlockPLCB**. This callback creates a new parameter block. The new parameter block inherits any parameters from the old parameter block whose parameter IDs match. In this way the older format is automatically converted to the new format by matching the corresponding IDs.

```
IOResult SphereObject::Load(ILoad *iload)
{
  iload->RegisterPostLoadCallback(
    new
    ParamBlockPLCB(versions,NUM_OLDVERSIONS,&curVersion,1)
  )
  return IO_OK;
}
```

See also: [Loading and Saving](#), [ParamBlockPLCB](#).

# Parent-Child Hierarchy

See Also: [Class INode](#).

## Overview

In 3ds max nodes in the scene may be linked together to form a hierarchy. 3ds max provides a developer with methods to work with this hierarchy via the **Inode** class.

A node that is linked to another node is referred to as a child node. A node that has children is referred to as a parent node. A node may have several children, but only a single parent. 3ds max provides access to the children of a node using an array mechanism.

## Methods

The **INode** method **NumberOfChildren()** may be used to retrieve the number of children a node has. To retrieve any of the children, use the method **GetChildNode(int i)**. This returns an **INode** pointer to the 'i-th' child. To retrieve the parent of a node use the method **GetParentNode()**. This returns an **INode** pointer to the parent node.

To attach a node to another node, use the method **AttachChild(INode\* node, int keepPos=1)**. This method attaches the specified node to the calling node.

To detach a child node use **Detach(TimeValue t, int keepPos=1)**. Both these methods allow the original position of the node prior to the hierarchy change to be maintained.

The method **IsRootNode()** determines if this node is a root node (does not have a parent node other than the world).

Note the following regarding cycles in hierarchical links:

Cycles with both hierarchical links and references can be legal or illegal, depending on the nature of the references. If a node's transform controller depends on one of the node's child nodes (or on any of its descendent nodes) this is illegal. However there is some special case code so that if a node's \*object depends on any of the node's descendent nodes, it is not considered a cycle, and is legal.

To review the full details on these methods see the reference section [Class INode](#).

# Profiling Plug-In Performance

See Also: [Debugging](#).

## **Overview**

There is a utility plug-in called the 3ds max Profiler that is available to allow developers to analyze the execution and performance of their plug-ins. The profiler supports analyzing Release build plug-ins. This allows profiling to take place with full optimization, inlining, etc. (Hybrid and Debug builds can also be profiled however this doesn't provide accurate information since the code is not optimized).

## **Two Types of Profiling**

There are two types of profiling that may be done. One is called **Polling** and the other is called **Instrumented**.

**Polling Profiling** requires no special compiler settings other than generating debug information (a PDB file). This works by setting up a separate thread and periodically looking at the context of the main thread. It looks at the instruction pointer to determine what instruction the CPU is at and stores this information. When finished, the stored addresses are used to report the module, source file and line number of the function (see below for the details on enabling these options).

**Instrumented Profiling** requires a special compiler setting but provides much more accurate profiling information.

Typically both of these are used. The polling profiler is used first to get a rough idea of where the greatest amount of execution time is being spent. Then instrumented profiling is done for those time consuming areas. This allows one to get a quick overview using polling and then collect very accurate, detailed information using instrumented profiling.

## Preparing For Profiling -- Compiler and Link Settings

There are several things that need to be done to get a plug-in ready for profiling. These are:

1. Generating debugging information.
2. For Instrumented Profiling Only -- Linking to the profiling LIB file.
3. For Instrumented Profiling Only -- Compiling using a special compiler flag.

These steps are outlined below:

1. You need to make sure debugging information is available. This can be turned on in the IDE using **Project / Settings / C++ Tab** and choosing **General** from the **Category** dropdown and choosing **Program Database** from the **Debug info** combo box. You'll also need to check the **Generate Debug Info** check box in the **General Category** of the **Link Tab** of the dialog. This will cause a PDB file to be written. The profiler needs the PDB file in order associate the addresses it collects as the program executes with actual entry points in the code.
2. For Instrumented Profiling Only -- You must add **ACAP.LIB** to the plug-ins link libraries. This is done using the **Project / Add To Project / Files** command from the pulldown menus. From the dialog presented choose **\MAXSDK\LIB\ACAP.LIB**.
3. For Instrumented Profiling Only -- to work the plug-in must be compiled with a special compiler flag: **/Gh**. This option isn't available through the IDE in a check box -- it must be manually typed into the Project Options list. You access this list from the IDE using **Project / Settings / C++ Tab / General Category**. At the beginning of the list of options add **/Gh**. What the **/Gh** option does is instruct the compiler to call a special function named **penter()** prior to doing anything for every procedure entry point. The **penter()** function is provided by the profiler and is used to collect all the profiling information.

## Using the MAX Profiler Plug-In

The profiler is a utility plug-in (**MAXSDK\PLUGIN\MAXPROF.DLU**) with an interface in both the command panel and a small floating dialog. The command panel interface is used to indicate which type of profiling is done, which information is written to the output file, and what file name it's written to. The floating dialog is used for starting, stopping, saving and clearing the profile recording.

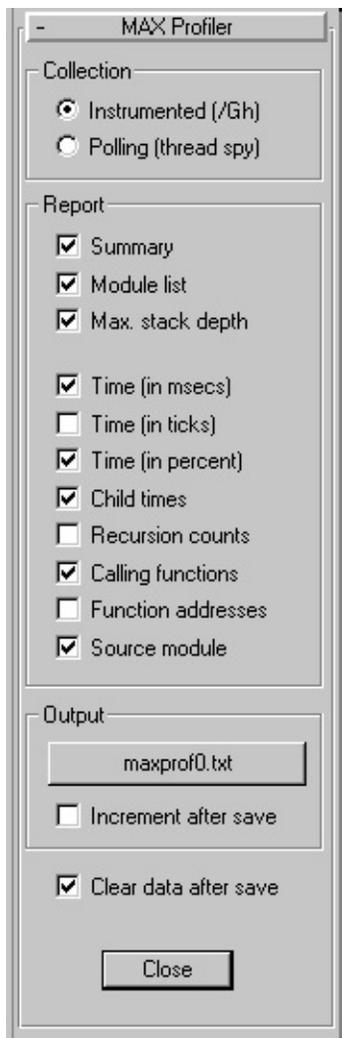
To run the profiler select **3ds max Profiler** from the list of utility plug-ins. A smaller floating dialog appears as well as the command panel interface. Via radio buttons in the Collection area choose if you want to perform Instrumented or Polling profiling.

To collect profiling information you select the options you want to record using the check boxes in the **Report** area of the command panel rollup. Next choose the desired output file name, from the **Output** rollup. By default the output file is written to **maxprof0.txt** in the same directory as the currently loaded 3ds max file.

Once these choice have been made, click on the **Record** button in the floating dialog. This puts the profiler in record mode -- ready for the collection of profiling information.

Next, simply use the plug-in you wish to profile. For instance, if your plug-in is a modifier, apply it to an object and adjust its settings. As you work with the plug-in, or 3ds max calls its methods, the profiling information is collected. The floating dialog stays active even if you leave the utility branch of the command panel so you're free to use all of 3ds max to fully test your plug-in. Note that the profiler is persistent even through a 3ds max File / Reset operation. This allows developers to track what calls are made during a Reset as well. If you want to reset the profiling information, press the Clear button on the floating dialog.

This section list each of the controls in the command panel user interface:



### Report section:

The check boxes in this section determine which portions of the output file are written. Below is a description of each of the options in this dialog. The sample output file shown later in this topic shows an example what is written for each of these.

**Summary:** This generates some basic information about the speed of the machine 3ds max is running on, the amount of time spent profiling, and the number of functions that were profiled.

**Module list:** This generates a list of all the modules that are loaded into process memory.

**Max stack depth:** This reports the maximum depth of function calls the profiler reached.

The following options are reported for every function that is called. These

items appear as columns of data in a table where each row represents one function call.

**Time (in msec):** This is the amount of time spent in the function in milliseconds. This is labeled **SelfTime** in the output table.

**Time (in ticks):** This is amount of time spent in the function in ticks. This is labeled **SelfTicks** in the output table.

**Time (in percent):** This is the time in a function as a percent of the overall time spent in all the functions that were profiled. This is labeled **SelfPercent** in the output table.

**Child times:** If this is checked, three new columns are added to the output table. These are labeled **HierTicks**, **HierTime** and **HierPercent**. These are similar to those above, but include not only the function itself, but also any functions it calls. This one is useful because you can sort the table by child times (in a spreadsheet such as Excel) and it will display the most time intensive functions first.

**Recursion counts:** This option is useful if you have recursive functions. It shows how many time the function calls back on itself. This is labeled **Recursions** in the output table.

**Calling functions:** This generates a multiline entry. For each function profiled, a list of all the functions, or places within a function, that called it are shown. This also lists how many times each of the functions called it.

**Function addresses:** This generates the address in memory for the function. This is labelled **Address** in the output.

**Source module:** This generates the name of the module (the name of the DLL that the profiled function lives in). If you are running under NT5 and have **IMAGEHLP.DLL** you'll get the source file name and line number. This is labelled **Module** in the output.

The number of calls (labeled **Count** in the output table) is always reported. This is the number of times the function was entered.

### **Output section:**

**Output file selector button:** By default, the output is written to the path where the 3ds max file was loaded. Clicking on this button brings up the file selector dialog for choosing a new output file.

**Increment after save:** This check box is enabled only when the filename has a number or numbers at the end. If checked, the file name shown on the

button is incremented each time the profiling information is saved. If unchecked, the file is overwritten on a save.

**Clear data after save:** This check box resets the profiler after each time the profiling information is saved. If this is off, profiling information continues to accumulate after a save.

**Close:** This closes the 3ds max Profiler plug-in in the user interface.

This section list the controls in the floating dialog.



The floating dialog has three buttons used for starting/stopping, saving and clearing the profile recording.

**Record / Stop:** This button is used to start and stop the collection of profiling information.

**Save:** This button is used to save the collected profiling information to disk.

**Clear:** This button is used to reset the profiling information so new recording will start from a clear state. If you want to clear out any existing profiling information generated press this button.

## Interpreting the Profiler Output File

This section presents an example of output from the profiler and describes the meaning of each section.

### The summary section:

This section is written if the 'Summary' check box is checked.

**Tue Jul 29 15:46:10 1997**

**WinNT 4.0**

**microSecsPerTick 0.12126**

This simply gives data about the timing of the CPU. This is the number of micro seconds per tick. Tick counts are used in several places in the profiler output.

**Ticks (elapsed/collected) 1860866843/305119085**

This tells how many CPU ticks went by during the run of the profiler (between when the Record and Stop buttons were pressed) over how many were actually collected (counted) by the profiler.

**Secs (elapsed/collected) 225.649/36.9988**

This tells the same information as above, only in seconds.

**Total functions 76**

This is the total number of functions that were profiled.

**Total calls 7493**

This is the total number of calls that were found while profiling.

### The module list section:

This section is written if the 'Module list' check box is checked.

**module base**

**3dsmax 0x400000**

**ntdll 0x77f60000**

**...etc...**

**GDI5 0x33600000**

**SZB5 0x33200000**

**msafd 0x77660000**

**wshtcpip 0x77690000**

This simply lists all the modules that are loaded into process memory and the

address of where they actually wound up.

### Max stack depth section.

This section is written if the 'Max stack depth' check box was checked.

### Max Stack Depth 8

This indicates how far into the procedure chain the profiler was used. That is, how many calls deep was the profiler involved in. This number is exclusive of recursion.

### The profiling data section.

The data shown below is copied from an Excel spreadsheet to make it appear in a table format. In pure text form as output by the profiler the data is tab delimited. One can simply copy this data to the clipboard, paste it into Excel, and it fills out the spreadsheet cleanly into rows and columns. The data here is just as described above under the user interface controls. Of particular note is the list of calling functions. When that option is checked the table may contain multiple lines per function. For each function profiled, a list of all the functions, or places within a function, that called it are shown. This also lists how many times each of the functions called it. For example, in the table below,

**GetString()** was called 13 times, 1 time by **EnterMode()**, 4 times by **DoPoint()**, 1 time by **DefineStroke()**, 2 times by **DefineStroke()** at a different address, 3 times by **UpdateStrokeInfo()**, and 2 times by **ReviewStrokes()**.

```
Count Recursions SelfTicks SelfTicks/Call SelfTime SelfPercent HierTicks HierTime HierPercent Address I
13 0 7502 577 1 0 7502 1 0 822087728 GetString stroke!E:\devel\3dswin\src\DLL\STROKE\stroke.cpp@4
    1 822090354 StrokeCMode::EnterMode +32 stroke!@
    4 822091673 StrokePrgm::DoPoint +69 stroke!@
    1 822101165 StrokeTable::DefineStroke +60d stroke!@
    2 822102498 StrokeTable::DefineStroke +b42 stroke!@
    3 822103004 UpdateStrokeInfo +7c stroke!@
    2 822105572 StrokeTable::ReviewStrokes +7b4 stroke!@
3 0 1050 350 0 0 1050 0 0 822087792 BuildStrokeSetPathName stroke!E:\devel\3dswin\src\DLL\STROKE\
    1 822100149 StrokeTable::DefineStroke +215 stroke!@
    2 822103881 StrokeTable::ReviewStrokes +119 stroke!@
21 0 1727 82 0 0 1727 0 0 822087968 Stroke::Stroke stroke!E:\devel\3dswin\src\DLL\STROKE\stroke.cpp@
    21 822103715 StrokeTable::ReviewStrokes +73 stroke!@
144 0 885 6 0 0 885 0 0 822088224 Stroke::GetFunctionName stroke!E:\devel\3dswin\src\DLL\STROKE\st
    60 822089505 StrokeTable::Sort +51 stroke!E:\devel\3dswin\src\DLL\STROKE\stroke.cpp@583
    60 822089514 StrokeTable::Sort +5a stroke!E:\devel\3dswin\src\DLL\STROKE\stroke.cpp@590
```

1 822101304 StrokeTable::DefineStroke +698 stroke!@  
20 822102448 StrokeTable::DefineStroke +b10 stroke!@  
3 822102962 UpdateStrokeInfo +52 stroke!@  
31 0 17268 557 2 0 17268 2 0 822088256 Stroke::Save stroke!E:\devel\3dswin\src\DLL\STROKE\stroke.cp  
31 822089776 StrokeTable::Save +80 stroke!@

## **Profiling and Processors**

This section discusses how the profiler deals with multiple processors.

The profiler forces 3ds max to run on a single processor. This is beneficial because it provides more accurate timing results than trying to collect the data from multiple processors.

## Profiling in MAX 3.0

With some combinations of system software and Visual C development environments, the profiler will fail to output symbols, even though the .pdb information is present. This is caused not by the profiler itself, but by the imagehlp.dll upon which it relies. This imagehlp.dll is supplied by Microsoft, and while it has undergone some changes, it has not stayed completely current with the rest of the OS DLL naming conventions. This is to say that they're loading some old DLLs explicitly - clearly a bug in the imagehlp.dll.

Consequently, you have two options for how you can resolve the problem. You must choose one of these two work arounds:

1. See that you have the NT4.0 version of %SYSTEMROOT%\system32\imagehlp.dll installed. It should have the file version of 4.0.1381.125 as seen in the explorer's file properties. Now, curiously, this file loads MSPDB50.DLL via hard-coded string. Of course if you're using VC6, you'll have MSPDB60.DLL rather than MSPDB50.DLL. Fortunately, the two are compatible at the interface level, so you can "copy MSPDB60.DLL MSPDB50.DLL" to provide one for imagehlp.dll to load. The drawback of using this fix is that the imagehlp.dll from NT4 does not support filename/line number information, and the most discrete info you can retrieve from the profiler's output is the names of the functions.

2. If you have access to NT5b2, you can use some of the DLLs supplied there. On that CD, in \i386 you'll find "imagehlp.dll" and "msdbi.dll". Taken together these files will work in NT4 just fine. Moreover, they'll supply the cool-ofilename and line number info that makes the profiler more valuable. Be sure that the file versions you get are 5.0.1878.1 for imagehlp.dll and 6.0.8337.0 for msdbi.dll. Again, this version info is available from the explorer's file properties dialog.

With whichever method you elect, you can put the required files either in your system32 directory so that they're available to the system as a whole, or in max's executable directory, in which case they'll affect only 3ds max .

## Read Only Plug-Ins

See Also: [Anti-Piracy Protection](#).

Plug-ins can read the 3ds max hardware lock ID number. Using the hardware lock ID, developers can provide versions of their plug-in that are slightly disabled. When the plug-in was loaded, if the current hardware lock wasn't the one it was authorized to run on, it would put itself in a disabled mode.

For example, this might mean the user interface for the plug-in was only partially enabled. Or perhaps the plug-in's parameters might not be fully editable in MAX. Possibly they are assigned fixed values, or values of a limited range, so the user can get the general idea of the plug-in but not really work with it in production.

These disabled versions could be freely distributable and contain information on how the complete versions can be purchased. This kind of information can be shown to the user via the 3ds max File/Summary Info.../Plug-In Info...

command. This is accomplished using the DLL function **LibDescription()**.

See [DLL Functions and Class Descriptors](#) for more information on this function.

A developer could implement a system where the user could phone the plug-in distributor and get a key number to unlock the disabled version. This key would allow the plug-in to be bound to the hardware lock of the new machine.

Sample code using an anti-piracy / authorization scheme can be found in **\MAXSDK\SAMPLES\MODIFIERS\TWIST.CPP**.

## References

See Also: [Class ReferenceMaker](#), [Class ReferenceTarget](#), [List of Reference Messages](#).

## Overview

In the 3ds max architecture, elements of the scene often form dependencies on one another. The typical manner these dependencies are handled in 3ds max are through References\*. A reference is a record of dependency between a reference maker and a reference target. The reference maker is said to be dependent upon the reference target. If the target changes in some way that affects the maker, the maker must be notified so it may take appropriate action.

\* Note: This use of the term reference in this section should not be confused with the term reference used in the 3ds max interface and user manuals. Nor is it to be confused with the C++ definition of reference. In this section, the term reference will always apply to the notion of a dependent relationship unless specifically stated otherwise.

Below are a few examples of dependent relationships between elements of a 3ds max scene that are managed using references:

A loft model is dependent upon its path shapes. When one of the path shapes changes, the model must be notified so it may update itself.

The path controller is dependent upon the spline path the controller is assigned to follow. When the spline path changes shape, the controller must be notified so it may realign its node to the revised path.

The procedural sphere is dependent upon its animated parameters. When the parameters are changed, the object must be notified so that it can update its cached representation to reflect the new settings.

The above examples show how references have been used in 3ds max plug-in development. Any time a developer wants to set up a dependent relationship between two elements of the scene and be notified of any changes, a reference may be used.

There are two key classes involved in the 3ds max reference system. These are the **ReferenceMaker** class, and the **ReferenceTarget** class. The **ReferenceTarget** class is derived from the **ReferenceMaker** class. Most plug-in classes are derived from **ReferenceTarget**. Thus most plug-ins may make references themselves.

There are three key methods involved when working with references. These are **MakeRefByID()**, **NotifyDependents()**, and **NotifyRefChanged()**.

## Change Notification Methods

The reference scheme allows a reference target to notify all its dependent reference makers when it changes. This section presents an overview of how this is done.

Having a reference is similar to having a pointer to an object, however when a maker references a target, that target maintains a pointer back to the maker. The reference target keeps a list of back pointers to all the reference makers which reference it. This gives the target the ability to notify each of its dependent reference makers when it has changed in some fashion. There are several methods that must be called or implemented by the plug-in when it uses references:

The reference maker must inform the reference target that it is dependent upon it. It does this by creating a reference to the target using a method called **MakeRefByID()**. The target then maintains this record of dependency via its pointer back to the reference maker.

When a reference target changes it must notify its dependent reference makers of this change. It does this by calling a method **NotifyDependents()**.

A reference maker must implement a method to receive the change notification messages sent by the target. It does this by implementing a method called **NotifyRefChanged()**.

Let's look at each of these methods and how they are handled by a specific 3ds max plug-in. We'll use the example of the path controller. The path controller governs the position of a node in the scene allowing it to track along a selected spline path. A spline path node in the scene is chosen to follow, and a parameter controls the amount of banking applied to the node as it follows along the path. (The path controller has other parameters, but concerning references the others are similar to the two covered here). The processing of these parameters involves several references.

The path controller tracks another node in the scene. This node is a spline and it is of course free to change shape and orientation at any time. By creating a reference to the spline node using **MakeRefByID()**, the controller will be notified whenever the spline changes. The path controller also must monitor its own banking parameter which may be animated and change over time. By creating a reference to this parameter the controller will be informed whenever the parameter changes.

A reference target is responsible for notifying all its dependent reference makers when it has changed. The spline path is a reference target. It is therefore responsible for informing its dependent reference makers when it changes. It does this by calling a method of ReferenceMaker **NotifyDependents()**. Any reference target maintains a list of pointers to the items that reference it. Internally, the **NotifyDependents()** function loops through all these pointers and calls a method **NotifyRefChanged()** on each one. **NotifyRefChanged()** is the method implemented by the reference maker responsible for receiving and responding to the change notification messages.

The path controller also maintains a banking parameter and has created a reference to it, so the parameter has become a reference target. Whenever the user alters the value of the banking parameter, a message must be sent to notify the path controller it has changed. This is done by calling **NotifyDependents()**. As part of this function call it passes the message **REFMSG\_CHANGE**. The **NotifyDependents()** call broadcasts this message to all the items which reference it. The bank parameter has one reference to it -- the path controller.

As a reference maker, the path controller must have a way to receive and respond to messages sent to it. The path controller must respond to the notification of the banking parameter changing and the spline path changing. It does this by implementing a method of the ReferenceMaker class called **NotifyRefChanged()**. The usual implementation of this method has a switch statement where each case is one of the messages the plug-in must respond to. The messages the plug-in must respond to depend upon the types of references it makes. There are additional messages which the system itself may need the plug-in to respond to. For example, say the spline which the path controller is following is deleted from the scene. The system needs to inform the plug-in that this has occurred. The plug-in developer must handle this possible condition by providing a case in **NotifyRefChanged()** to respond to the message **REFMSG\_TARGET\_DELETED**. See [List of Reference Messages](#).

## **Details of the Change Notification Methods**

This section looks in detail at the methods used by reference makers and targets for change notification.

## Making References

To create a dependency upon an item, the developer creates a reference to the item. This is done by calling a method of ReferenceMaker called **MakeRefByID()**.

```
RefResult MakeRefByID(  
    Interval refInterval,  
    int which,  
    RefTargetHandle htarget  
);
```

This method creates a reference between the object which calls the method, and the ReferenceTarget specified by the **htarget** parameter.

The **refInterval** parameter indicates the interval of time over which this reference is active. Outside this interval, the reference is not considered to be a dependency. This allows the plug-in to have dependent relationship over only portions of an entire animation time range. If a plug-in has a dependency over the entire animation it may use the pre-defined interval **FOREVER** for this parameter. In the current implementation all plug-ins must use **FOREVER** for this interval.

The **which** parameter indicates which reference index this newly created reference is assigned to. The system uses a virtual array mechanism to access the references an item has. The developer simply assigns an integer index to each reference. For example, the path controller might use an index of 0 for the bank amount parameter reference, and an index of 1 for the node to follow reference. The path controller would then pass either 0 or 1 as the **which** parameter depending upon which reference it was making. This is discussed in greater detail below under Reference Access Methods.

The **hTarget** parameter is the handle of the item to which a reference is being made.

The return value from this method is of type **RefResult**. This is usually **REF\_SUCCEED** indicating the reference was created and is registered by the reference target.

## Sending Change Notification Messages

When a reference target changes it must notify its dependent reference makers of this change. It does this by calling **NotifyDependents()**.

```
RefResult NotifyDependents(  
    Interval changeInt,  
    PartID partID,  
    RefMessage message,  
    SClass_ID sclass=NOTIFY_ALL,  
    BOOL propagate=TRUE,  
    RefTargetHandle hTarg=NULL  
);
```

This method broadcasts the message specified by the **message** parameter to all the items which reference the caller.

The **partID** parameter is used to pass message specific information to the items which will receive the message. See the Reference section [Class ReferenceMaker NotifyRefChanged\(\)](#) method for more details.

The **changeInt** parameter indicates the interval of time over which the change reported by the message is in effect. Currently all plug-ins must pass **FOREVER** for this interval.

The **sclass** parameter defaults to **NOTIFY\_ALL**. If this value is passed to **NotifyDependents()** all dependents will be notified. Other super class values may be passed to only send the message to certain items whose SuperClassID matches the one passed.

The **propagate** parameter defaults to **TRUE**. This indicates that the message should be sent to all 'nested' dependencies. If passed as **FALSE**, this parameter indicates the message should only be sent to first level dependents. Normally this should be left to default to **TRUE**.

The **hTarg** parameter defaults to **NULL**. A plug-in developer should never pass anything for this parameter. It must always default to **NULL**.

## Responding to Change Notification Message

A plug-in which makes references must implement a method to receive and respond to messages broadcast by its dependents. This is done by implementing the **NotifyRefChanged()** method of **ReferenceMaker**.

```
virtual RefResult NotifyRefChanged(  
    Interval changeInt,  
    RefTargetHandle hTarget,  
    PartID& partID,  
    RefMessage message  
);
```

The plug-in developer usually implements this method as a switch statement where each case is one of the messages the plug-in needs to respond to. The **message** parameters passed into this method is the specific message which needs to be handled.

The **changeInt** interval is the interval of time over which the message is active. Currently, all plug-ins will receive **FOREVER** for this interval.

The **hTarget** parameter is the handle of the reference target the message was sent by. The reference maker uses this handle to know specifically which reference target sent the message.

The **partID** parameter contains information specific to the message passed in. Some messages don't use the **partID** at all. See the section below on Reference Messages and PartIDs for more information about the meaning of the **partID** for some common messages.

The return value from this method is of type **RefResult**. This is usually **REF\_SUCCEED** indicating the message was processed. Sometimes, the return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

Below is an example of the code structure usually used to implement **NotifyRefChanged()**. It is a simplified version taken from the path controller:

```
RefResult PathPosition::NotifyRefChanged(  
    Interval changeInt,  
    RefTargetHandle hTarget,  
    PartID& partID,
```

```
RefMessage message)
{
switch (message) {
case REFMSG_CHANGE:
// Code to handle the target changing...
break;
case REFMSG_TARGET_DELETED:
if (hTarget == pathNode) {
// Code to handle to path node being deleted...
}
break;
};
return REF_SUCCEED;
}
```

Note: A plug-in should NOT normally call **NotifyDependents()** from its **NotifyRefChanged()** method. All the appropriate dependents are notified automatically and thus doing so is unnecessary.

## Reference Access Methods

The system manages the access to an item's references by using a virtual array. If the plug-in makes references, it must implement three methods of

**ReferenceMaker** to handle access to its references. These methods are:

### **int NumRefs();**

The plug-in implements this method to return the total number of references it makes.

### **RefTargetHandle GetReference(int i);**

The plug-in implements this method to return a reference handle to its 'i th' reference. The plug-in keeps track of its references using an integer array index for each one. When the system calls this method, the plug-in returns its 'i th' reference.

### **void SetReference(int i, RefTargetHandle rtarg);**

The plug-in implements this method to store the reference handle passed into its 'i-th' reference. The plug-in simply keeps track of its references using an integer array index for each one. When the system calls this method, the plug-in stores its 'i-th' reference.

Below is an example of how these methods might be implemented

```
#define PATHPOS_BANK_REF 0
#define PATHPOS_PATH_REF 1
int NumRefs() { return 2; }
RefTargetHandle PathPosition::GetReference(int i)
{
    if (i==PATHPOS_BANK_REF) {
        return bankAmount;
    } else {
        return pathNode;
    }
}
void PathPosition::SetReference(
    int i,
    RefTargetHandle rtarg)
{
    if (i==PATHPOS_BANK_REF) {
```

```
bankAmount = (Control*)rtarg;  
} else {  
  pathNode = (INode*)rtarg;  
}  
}
```

## Maintenance Methods

The following are methods that may be used to delete references when they are no longer needed:

### **DeleteAllRefFromMe();**

This deletes all references from the calling reference maker.

### **DeleteAllRefs();**

Deletes all references to and from the calling reference maker.

### **DeleteMe();**

This method deletes all references to and from the calling reference maker, sends the **REFMSG\_TARGET\_DELETED** message, handles Undo, and deletes the object.

## Deleting and Replacing References as the Reference Structure Changes

The number of references maintained by a plug-in may change over time. This section presents information on the different approaches developers may use to allow the reference structure to change.

Basically, a developer can have two kinds of reference structures. One approach is to have the number fixed, where some references may go to NULL occasionally, but the number stays the same. Alternatively, a developer can have a variable number of references, and move them around as they are added and deleted.

The first option, where the number stays fixed, is accomplished by keeping the number of references returned by **NumRefs()** constant. An example of this from 3ds max is the 3D Texmap Marble (**\MAXSDK\SAMPLES\MATERIALS\MARBLE.CPP**). It has 4 references: the pblock, the xyzGen instance, and two sub-texmaps. When the texmap starts up fresh it doesn't have sub-maps. Yet it still returns 4 from **NumRefs()**. When the user clicks on a user interface button to add a sub-map, the texmap calls **ReplaceReference()** on the sub-texmap index and a reference to the new sub-texmap gets plugged-in. Thus the pointer to it is no longer NULL. Again, **NumRefs()** doesn't change, it is just that reference is no longer NULL. A user can also set the sub-texmap back to 'None' to stop using it. In this case **DeleteReference()** will be called (**DeleteReference()** breaks the connection between the pointer stored and the system -- this is done so 3ds max will not send change notification messages any longer). Again, even though the reference was deleted **NumRefs()** still returns 4.

**DeleteReference()** will set the pointer to NULL.

The other approach to structuring references involves altering the number of references returned from **NumRefs()**. A developer can, after deleting a reference, change **NumRefs()** to one less and move around the pointers so **GetReference()** and **SetReference()** still return the proper values. An example of when this might be used is if the data structure for the plug-in had a variable number of references that the user could directly alter. The Multi-SubObject material is structured like this (**\MAXSDK\SAMPLES\MATERIALS\MULTI.CPP**). It maintains a table of the sub-materials and the user can alter the number of these on the fly via the user interface. In its implementation of **NumRefs()** it returns the

number of items in the table at that moment. When the user asks for fewer sub-materials (and thus fewer references) **ReplaceReference()** is called passing NULL and **NumRefs()** returns a smaller value. When the user adds additional sub-materials, new materials are created and **ReplaceReference()** is called passing pointers to these new materials.

## Pointers and References

When a plug-in has a pointer to an item it should usually make a reference to the item as well. This will ensure the item is not deleted by the system (which would make the pointer invalid).

In 3ds max when the last reference to an item is deleted, the item itself is deleted.

For example, say your plug-in has a pointer to a sub-anim, but you have not created a reference to it. If this sub-anim appears in track view, the system will create a reference to it because track view needs a reference to the items it displays. When track view is finished displaying this sub-anim, it will delete the reference. If this is the last reference to the sub-anim, the sub-anim itself will be deleted. This would invalidate the pointer you are maintaining to the sub-anim. The way to prevent this from happening is to create a reference to the sub-anim. The system does not delete items that still have references to them. Therefore by making a reference to the item you ensure that the system will not delete it.

The one exception to this is nodes in the scene. Nodes may be deleted by the user by simply selecting the node and pressing the delete key. If an item has a reference to the node the user may still delete it. However the item that referenced the node will receive the message

**REFMSG\_TARGET\_DELETED** via **NotifyRefChanged()**. It can then respond appropriately to the deletion.

At certain times a plug-in may need to have a two-way reference. For example, if there is an item that references another item, the referenced item cannot have a reference back to the thing that referenced it, because this would create a loop (a cyclic reference). Cyclic references are illegal. In other words if A has a reference to B, then B **cannot** have a reference to A.

The Ring Array plug-in has a situation where A has a reference to B, but B has a **pointer** to A (but doesn't have a reference to A since that would cause a loop). Again, whenever you have a pointer to something you should have a reference to the item as well. But B cannot reference A.

What is done in this case, where you want a two-way reference, is as follows. A has a reference to B, and A can give B a pointer to itself. A is then responsible for informing B if it gets deleted so B can set its pointer to A to NULL. So A is responsible for making sure B's pointer to A doesn't end up invalid. This is the way you can set up a two-way dependency yet not create a cyclic reference. The

item that is being pointed at must manage the pointer for the item that maintains the pointer.

When this setup is saved to disk another problem arises. How does one save a pointer, since the value will of course be different when loaded. The solution to this is to use methods of the **ISave** and **ILoad** classes that let a developer save and restore pointers. This must be a pointer to one of the objects that the scene saves with the reference hierarchy, but it is not a pointer that itself is a reference. When the pointer is saved the method **ISave::GetRefID()** is used. This returns an integer ID that can be saved to disk. When loading, a method **ILoad::RecordBackpatch()** is used. This takes the ID and a pointer to a pointer and sets the pointer to point back at the item that was pointed at when things were saved. See [Class ISave](#), [Class ILoad](#) for more details.

## The Loading and Saving of References

The system takes care of loading and saving references when an object is saved to disk. An object does not need to explicitly save its references, nor does an object need to load its references. After a scene is loaded, an object's references will automatically be restored. The system does this by using the **NumRefs()**, **GetReference()** and **SetReference()** methods of the plug-in.

For example, the path controller plug-in has a reference to the spline shape node in the scene it follows. It does not save this node explicitly in its implementation of **ReferenceMaker::Save()**. Yet this reference will be restored whenever the file is loaded from disk. This happens because the reference is saved and restored automatically by the system.

For anything that you have a reference to that you want saved and reloaded, you need to have a class descriptor registered with the system. This is because the system needs to call **ClassDesc::Create()** on the item to create it upon reloading. See [Class ClassDesc](#).

Note that since references are restored automatically at loading time 3ds max needs to call **ReplaceReference()**. This **ReplaceReference()** will cause a call to **DeleteReference()**, which checks if the return value of **GetReference()** is NULL. Therefore developers have to initialize their references to NULL, if the **ClassDesc::Create()** method is called with **loading==TRUE**. If they don't and **GetReference()** is called with a non initialized return value at loading time, 3ds max will crash.

While loading, that is in a plug-in's implementation of its **Load()** method, the references are NOT in place yet. The references are not in place until everything in the 3ds max file is loaded. If a developer needs to do something with the references when loading, a post load callback may be used. This is done using the method **ILoad::RegisterPostLoadCallback()**. The callback is called after all loading is complete. Inside the callback the plug-in's references will be in place and they may be processed by the plug-in.

Another way a post load callback is used is to restructure references during a **Load()**. For example, say a plug-in references a static number of objects followed by any number of additional objects. Later, as development progresses on the plug-in, another static object is added to the reference structure. It would be nice if old files that still used the older reference structure could be loaded. The way to solve this is as follows: After the plug-ins **Load()** method is called

its references are put in place using **SetReference()**. What needs to happen is that the plug-in should check for older versions of files (by checking a saved version number) and set a flag to indicate the old version is being loaded. Then a post load callback is registered to turn off this flag once all the references have been established. With this mechanism in place, the **SetReference()** method can check this flag and integrate the old reference structure or the new reference structure accordingly.

For users of the r2 or later version of the SDK a new method is available to handle the above. See the method

**ReferenceMaker::RemapRefsOnLoad()**.

## Referencing a Global Instance of a Non-Plugin Class

If a plug-in needs to have some sort of global instance that its other objects reference, but is itself not a specific plug-in type, then the super class ID of **REF\_TARGET\_CLASS\_ID** should be used. The class of the global instance should be derived from **ReferenceTarget**.

## Viewing Reference Messages

This section discusses a utility plug-in called the Reference Watcher. This plug-in may be used to help understand the reference structure of a chosen item and to monitor the reference messages it sends. This provides a quick, visual way to examine the use of references in MAX.

To run this plug-in you'll first need to build it. The project is in **\MAXSDK\SAMPLES\HOWTO\REFCHECK**. Load this project into VC++ and press F7 to build it. Place **REFCHECK.DLU** into your 3ds max DLL search path (for example in the **STDPLUGS** directory). You can then run the program by choosing How To/Reference Watcher in the utility panel.

This plug-in allows you to create a reference to a chosen Reference Target and then watch the messages sent as you work with the target. For instance, follow the steps below to analyze the reference messages sent by a node in the scene.

Reset 3ds max and create a single Box in the scene called **Box01**. Leave it selected after creation.

Launch the **Reference Watcher** from the **Utility** branch of the command panel. Click on the **Pick Reference Target to Watch** button and choose the **Box01** node from the dialog. This is done by opening the **Objects** branch and selecting the **Box01 label** then pressing **OK**.

Result: The Reference Watcher utility creates a reference to the box node in the scene and displays information about it. The upper most list box shows this list of reference messages. Since we haven't done anything to the node yet the message list is blank.

The second list box in the dialog shows the items that are referencing the node we've selected. This provides a way to get a general idea of who is referencing an item. In the case of the Box01 node these are:

- 0: Node
- 1: (This Reference Watcher)
- 2: Node Selection
- 3: Material Editor
- 4: Scene
- 5: Material Editor
- 6: Scene
- 7: Animatable.

The bottom most list box shows the references that the node itself has. For the **Box01** node it has six reference targets. These are:

- 0: The Transform Controller
- 1: The Object Reference
- 2: The Pin Node for IK (not currently assigned -- NULL)
- 3: The Material Reference (not currently assigned -- NULL)
- 4: The Visibility Controller (not currently assigned -- NULL)
- 5: The Image Blur Controller (not currently assigned -- NULL)

As you work with the node (select it/deselect it, assign modifiers, bind to space warps, assign materials, change controllers, drag the item in the viewports, etc.), you can monitor the reference messages sent. They'll show up using the **#define** name from **REF.H** (for example **REFMSG\_CHANGE**) in the upper most window. If you want to clear the message list so you can see what messages are sent for a particular action press the **Clear Message List** button first.

Developers can use this utility to help study the use of references inside MAX.

## Additional Global Functions Related to the Reference Hierarchy

The following global functions are available for working with references:

### Function:

```
void EnumRefHierarchy(ReferenceMaker *rm, RefEnumProc  
&proc);
```

### Remarks:

This function provides a general purpose reference enumerator. It simply calls the **RefEnumProc::proc()** on each element in the reference hierarchy. See [Class RefEnumProc](#).

### Parameters:

**ReferenceMaker \*rm**

The reference maker whose dependents will be enumerated.

**RefEnumProc &proc**

The callback object whose **proc()** method is called for each element.

### Function:

```
ReferenceTarget *CloneRefHierarchy(ReferenceTarget *rm);
```

### Remarks:

A new global function has been added for release 2.0 to clone a reference target and the hierarchy emanating from it. This function encapsulates the code necessary to clone a **ReferenceTarget** and the reference hierarchy emanating from it. It handles multiple instances correctly, using a **RemapDir**.

### Parameters:

**ReferenceTarget \*rm**

The reference target to clone.

### Return Value:

A duplicate of the entire reference hierarchy of the item passed.

### Sample Code:

In this example, a material is being copied from one **MtlBase** to another (**source** to **dest**). To copy all the sub-materials this function is used:

```
dest[i] = (MtlBase *)CloneRefHierarchy(source[i]);
```

**Function:**

**BOOL DependsOn(RefMakerHandle mkr, RefMakerHandle targ);**

**Remarks:**

This global function is available in release 2.0 and later only.

Returns TRUE if there is a path of references from **mk**r to **ta**rg; otherwise FALSE. Note that this returns TRUE if **mk**r == **ta**rg.

**Parameters:**

**RefMakerHandle mkr**

The reference maker to check.

**RefMakerHandle targ**

The reference target to check.

**Function:**

**void ClearAFlagInHierarchy(ReferenceMaker \*rm, int flag);**

**Remarks:**

This function is available in release 2.0 and later only.

Clears the specified Animatable flag(s) from each item in the reference hierarchy.

**Parameters:**

**ReferenceMaker \*rm**

The reference maker whose dependents will be enumerated and have their flags cleared.

**int flag**

See [List of Animatable Flags](#).

## Summary

References are used to handle dependencies between items in the scene. A plug-in making references is responsible for implementing several methods. The method **NotifyRefChanged()** is used to receive messages that something it references has changed. The plug-in must also implement a method **NumRef()** to return the number of references it makes. Methods **GetReference()** and **SetReference()** must be implemented by the plug-in to allow the system to access its references. If an item is a reference target, it must call **NotifyDependents()** to broadcast a message whenever it has changed in a way which affects other elements in the scene.

# Render Elements

See Also: [Class IRenderElementMgr](#) , [Class IRenderElement](#), [Class IRenderElementCompatible](#), [Class MaxRenderElement](#)

## Overview

Render Elements allow renderers to output to separate bitmaps portions of the final shaded image so that they can be manipulated/composited at a later time, without re-rendering.

Render Elements are specific to a given renderer. 3ds max defines both a general interface that all render elements must support & an interface specific to elements supporting the max default renderer. Mental Ray specific elements will support the general interface, but probably define their own specific interface. Part of the general interface allows the system to figure out which installed render elements belong to which renderer, & to only list those compatible elements.

3ds max render elements are supported by code in each material. After shading each active render element's PostIllum method is called in turn. It is provided with;

- · the full shade context used to do the shading
- · pointers to the material & (if there is one) shader used to do the computation
- · textured inputs to the shader (e.g. diffuse color blended w/ diffuse texture color by the amount spinner)
- · component-wise output from the shading process (diffuse term, reflection term...)
- · there is provision for extending the component-wise outputs to include arbitrary name-matched channels (this allows materials w/ special components, e.g. fluorescence in ray material, to be communicated to an element. The element must of course be looking for the special component. The component-wise outputs maybe computed either with or without shadows, depending on a query on the element interface.

Essentially, this is all of the information that is passed to 3ds max Shaders, plus

the shadowed or unshadowed component-wise output from the materials shading process. This means that each render element may in fact be a complete, alternate, special-purpose shader. This is intentional, of course, so that any wacky thing might be created. Most current render elements don't take advantage of this, tho, they merely process the component-wise output from the materials shading process, but the potential is very powerful.

3ds max Render elements then store their computed shade in an extended ShadeOutput class (imtl.h) that is part of the ShadeContext. The ShadeOutput has an array of element output values, one per element & each max render element is given an index into the array to use. The ShadeOutput will follow the fragments being shaded all through the compositing process, so that transparency & partial coverage is properly resolved.

Atmosphere cannot be properly computed at shading time since the object may be covered by a transparent object, dividing the needed calculation into two: atmosphere from the eye to the transparent object, composited over atmosphere between the two objects & the final objects color. Hence there is a second, optional call to each render element after the atmosphere is computed. The general interface tells the system whether to apply atmosphere to the render elements color, does so if desired & calls the PostAtmosphere call on the max render element.

Finally, a last query on the general interface determines whether to apply the AA filter when outputting the element to its bitmap, or merely blending with other elements in the pixel.

A sample plugin of a Render Element can be found in the SDK samples;  
**\\MAXSDK\\SAMPLES\\RENDER\\RENDERELEMENTS.**

# Rotation Concepts

See Also: [Class Quat](#), [Class AngAxis](#), [Class Matrix3](#).

## Overview

This section describes the different ways rotation is represented in the 3ds max SDK and the various classes used in storing and interpolating rotation.

There are several class used in 3ds max that describe rotations in 3D space.

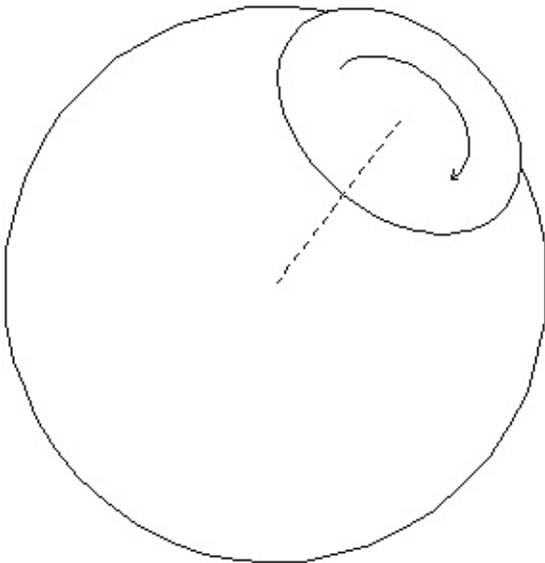
These classes are **Quat**, **AngAxis**, and **Matrix3**. There are also methods that convert between these classes.

## The Visualization of Rotation Space

The set of **all** rotations in space may be visualized as:

- 1) The set of all unit length axis vectors with one endpoint centered at the sphere center  $(0,0,0)$  and the other endpoint on the surface of the unit sphere.
- 2) The set of rotations about each vector axis (rotations about the radius vectors that point to each endpoint on the sphere).

One way to visualize this is to imagine that you could nail a dinner plate to the surface of a ball that is fixed in space (with the top surface of the plate facing the center of the ball, and the nail going from the center of the plate to the center of the sphere -- see the figure below). For each nailed position on the ball, imagine you can rotate the plate about the nail (axis). The point of placement of the nail on the ball, and the plate's rotation about it describes each of the possible rotations of the plate in a unique way.



Unit sphere with axis and plate shown.

## Quaternion Representation of Rotation

A quaternion may be visualized in a similar way:

- 1) An arbitrary unit length axis vector (one of the endpoints on the unit sphere...or the nail position).
- 2) A rotation about the axis (one on the rotations about the radius vectors that point to each endpoint on the sphere).

In the simplest terms, the rotational space described by each quaternion is a 3D unit vector plus a rotation. The **Quat** class has four public data members that store this information:

**float x, y, z, w;**

A normalized quaternion can represent  $-\pi$  to  $+\pi$  rotation -- that is -180 to 180 degrees of rotation. Quaternions use the left-hand rule for determining positive rotation.

## **AngAxis Representation of Rotation**

This class is similar to a quaternion, except that a normalized quaternion only represents  $-\pi$  to  $+\pi$  rotation (that is  $-180$  to  $180$  degrees of rotation). The **AngAxis** class can instead have any number of revolutions stored. It has two public data members:

**Point3 axis;**

**float angle;**

The **axis** is similar to the values in the **Quat** class. The **angle** value can represent any rotation amount in radians.

## **Matrix Representation of Rotation**

A matrix may be used to describe rotation. See the Advanced Topics section on [Matrix Representations of 3D Transformations](#) for more information. That section describes not only rotation, but translation and scaling as represented by matrices.

## Euler Angles

Multiple rotations are not independent. When you rotate something twice, the second rotation rotates the first rotation. For this reason, the order of rotation is important. The 3ds max Euler angle controller, by default, has an XYZ ordering. That is, the first rotation is done along the X axis, then along the Y axis, and finally along the Z axis. In other words, the default Euler angle controller has a type of EULERTYPE\_XYZ (described below). The 3ds max local Euler angle controller is similar, but specifies rotations around an object's local coordinates instead of the world coordinates. It has a type of (EULERTYPE\_XYZ | EULERTYPE\_RF).

The SDK provides functions for converting from a Matrix3 to Euler angles using various rotation orderings. You need the different types because you get different interpolation of the float values with the each type.

### Prototype:

```
void MatrixToEuler(const Matrix3 &mat, float *ang, int type);
```

### Remarks:

This function is available in release 2.0 and later only.

Converts the specified matrix to Euler angles using the **type** parameter to determine the order.

### Parameters:

**const Matrix3 &mat**

The rotation matrix to convert to Euler angles.

**float \*ang**

The result is stored here. The array ordering is based on the **type** parameter below.

**int type**

This parameter specifies the order of application of the angles. One of the following values:

```
EULERTYPE_XYZ  
EULERTYPE_XZZ  
EULERTYPE_YZX  
EULERTYPE_YXZ  
EULERTYPE_ZXY
```

**EULERTYPE\_ZYX**  
**EULERTYPE\_XYX**  
**EULERTYPE\_YZY**  
**EULERTYPE\_ZXZ**

Note: Internally, 3ds max uses static coordinate frames. OR in the **EULERTYPE\_RF** bit if you have a special need to use a rotating coordinate frame. This is described in Ken Shoemake's paper in Graphic Gems IV, pp. 222-229.

**Prototype:**

**void EulerToMatrix(float \*ang, Matrix3 &mat, int type);**

**Remarks:**

This function is available in release 2.0 and later only.

Converts from the specified Euler angle format (specifying rotation with a triple of angles) to a Matrix3 format.

**Parameters:**

**float \*ang**

The Euler angles to convert. The expected format is based on the **type** parameter below.

**Matrix3 &mat**

The matrix result is stored here.

**int type**

This parameter specifies the order of application of the angles. One of the following values:

**EULERTYPE\_XYZ**  
**EULERTYPE\_XZZ**  
**EULERTYPE\_YZX**  
**EULERTYPE\_YXZ**  
**EULERTYPE\_ZXY**  
**EULERTYPE\_ZYX**  
**EULERTYPE\_XYX**  
**EULERTYPE\_YZY**  
**EULERTYPE\_ZXZ**

Note: Internally, 3ds max uses static coordinate frames. OR in the

**EULERTYPE\_RF** bit if you have a special need to use a rotating coordinate frame. This is described in Ken Shoemake's paper in Graphic Gems IV, pp. 222-229.

**Function:**

**void QuatToEuler(Quat &q, float \*ang, int type, BOOL b=FALSE);**

**Remarks:**

Converts the quaternion to Euler angles. When converting a quaternion to Euler angles using this method, the correct order of application of the resulting three rotations is specified by the **type** parameter.

**Parameters:**

**Quat &q**

The quaternion to convert.

**float \*ang**

The angles are returned here in the same order as specified by **type** (for example **EULERTYPE\_XYZ** would be **ang[0]=x, ang[1]=y, ang[2]=z.**)

**int type**

This parameter specifies the order of application of the angles. One of the following values:

**EULERTYPE\_XYZ**

**EULERTYPE\_XZZ**

**EULERTYPE\_YZX**

**EULERTYPE\_YXZ**

**EULERTYPE\_ZXY**

**EULERTYPE\_ZYX**

**EULERTYPE\_XYX**

**EULERTYPE\_YZY**

**EULERTYPE\_ZXZ**

Note: Internally, 3ds max uses static coordinate frames. OR in the **EULERTYPE\_RF** bit if you have a special need to use a rotating coordinate frame. This is described in Ken Shoemake's paper in Graphic Gems IV, pp. 222-229.

**BOOL b=FALSE**

This parameter is available in release 4.0 and later only.

When this argument is set to false (or omitted), each function performs as it did before version 4.0. When the boolean is TRUE, the matrix is made with its terms transposed. When this transposition is specified, **EulerToQuat()** and **QuatToEuler()** are consistent with one another. (In Shiva, they have opposite handedness).

**Function:**

**void EulerToQuat(float \*ang, Quat &q, int type);**

**Remarks:**

Converts Euler angles to a quaternion.

**Parameters:**

**Quat &q**

The quaternion result is returned here.

**float \*ang**

The angles are specified in the same order as specified in the application (for example **EULERTYPE\_XYZ** would be **ang[0]=x, ang[1]=y, ang[2]=z.**)

**int type**

This parameter specifies the order of application of the angles. One of the following values:

**EULERTYPE\_XYZ**  
**EULERTYPE\_XZZ**  
**EULERTYPE\_YZX**  
**EULERTYPE\_YXZ**  
**EULERTYPE\_ZXY**  
**EULERTYPE\_ZYX**  
**EULERTYPE\_XYX**  
**EULERTYPE\_YZY**  
**EULERTYPE\_ZXZ**

Note: Internally, 3ds max uses static coordinate frames. OR in the **EULERTYPE\_RF** bit if you have a special need to use a rotating coordinate frame. This is described in Ken Shoemake's paper in Graphic Gems IV, pp. 222-229.

**Handling Sign Flips when Converting a Rotation Controller to an**

## Euler Controller

When converting any rotation controller to an Euler controller it is possible for sign flips to occur in the resulting animation. This is due to the fact that one single rotation matrix can be expressed through many different triplets of Euler angles. Sometimes the produced Euler angles are not interpolatable. There is code in the SDK which solves this problem. It samples the whole animation range and produces Euler angles that result in the same animation as the original, if interpolated. This code is generally valuable for developers who want to convert keys of any type of 3ds max controller that uses quaternions internally into a Euler angle keys (for example an export plug-in).

The code can be found in

**\MAXSDK\SAMPLES\CONTROLLERS\EULRCTRL.CPP** in the method **void EulerRotation::Copy(Control \*from)**. The algorithm works as follows:

The code samples the entire animation range and incrementally adds up the angles for each time step. During resampling it detects possible sign flips by comparing the Euler/Quat ratio of two successive time steps. The Euler/Quat ratio is the relation of the angle difference in Euler space to the angle difference in Quat space. If this ratio is bigger than PI the rotation between the two time steps contains a flip. The Euler/Quat ratio can be determined using the SDK functions **GetEulerQuatAngleRatio()** or **GetEulerMatAngleRatio()** (see below). If a flip is detected the algorithm calculates the actual angle not containing a flip and adds it to the incremental angle.

The actual detection of the flip is dependent on the amount of rotation in between the time steps. It is obvious, that the smaller the time step, the more accurate the detection is. In the code described above a time step of 1 tick is used, which is most accurate, but also most time intensive. The THRESHHOLD value describes the maximum intended rotation allowed between two time increments. In the code described above the THRESHHOLD is 1.0 which is equal to 57 degrees of rotation (a lot of rotation for one tick).

### Function:

```
float GetEulerQuatAngleRatio(Quat &quat1,Quat &quat2, float *euler1, float *euler2, int type = EULERTYPE_XYZ);
```

### Remarks:

This global function is available in release 3.0 and later only.

The Euler/Quat ratio is the relation of the angle difference in Euler space to the angle difference in Quat space. If this ratio is bigger than PI the rotation between the two time steps contains a flip. See the description above for details.

**Parameters:**

**Quat &quat1**

The 'previous' rotation.

**Quat &quat2**

The 'current' rotation.

**float \*euler1**

The 'previous' rotation as an euler angle.

**float \*euler2**

The 'current' rotation as an euler angle.

**int type = EULERTYPE\_XYZ**

This parameter specifies the order of application of the angles. One of the following values:

**EULERTYPE\_XYZ**  
**EULERTYPE\_XZZ**  
**EULERTYPE\_YZX**  
**EULERTYPE\_YXZ**  
**EULERTYPE\_ZXY**  
**EULERTYPE\_ZYX**  
**EULERTYPE\_XYX**  
**EULERTYPE\_YZY**  
**EULERTYPE\_ZXZ**

**Return Value:**

The Euler/Quat ratio between the time steps.

**Function:**

**float GetEulerMatAngleRatio(Matrix3 &mat1,Matrix3 &mat2,  
float \*euler1, float \*euler2, int type = EULERTYPE\_XYZ);**

**Remarks:**

This global function is available in release 3.0 and later only.

The Euler/Matrix Angle ratio is the relation of the angle difference in Euler space to the angle difference in Matrix space. If this ratio is bigger than PI the

rotation between the two time steps contains a flip. See the description above for details.

**Parameters:**

**Matrix3 &mat1**

The 'previous' rotation.

**Matrix3 &mat2**

The 'current' rotation.

**float \*euler1**

The 'previous' rotation as an euler angle.

**float \*euler2**

The 'current' rotation as an euler angle.

**int type = EULERTYPE\_XYZ**

This parameter specifies the order of application of the angles. One of the following values:

**EULERTYPE\_XYZ**

**EULERTYPE\_XZZ**

**EULERTYPE\_YZX**

**EULERTYPE\_YXZ**

**EULERTYPE\_ZXY**

**EULERTYPE\_ZYX**

**EULERTYPE\_XYX**

**EULERTYPE\_YZY**

**EULERTYPE\_ZXZ**

**Return Value:**

The Euler/Matrix Angle ratio between the time steps.

## Schematic View

See Also: [Class IGraphObjectManager](#), [Class IGraphNode](#), [Class Animatable](#).  
[Class SubClassList](#).

## Overview

The Schematic View window allows 3ds max users to browse, and perform certain operations on, many of the objects that compose a 3ds max scene. Like Track Views, multiple Schematic Views can be opened within MAX, both in modeless windows and docked in viewports.

Within a Schematic View window, a subset of the set of all Animatable objects within the 3ds max scene is represented by a collection of rectangular "nodes." These nodes are organized in tree form (or directed acyclic graph). The set of objects displayed depends on the current filter settings, the hide/show state of the nodes, and which objects were added to the Schematic View during the traversal phase. Schematic View is a **GlobalReferenceMaker** and any time it detects a change in the structure of the 3ds max scene it performs a traversal of the scene's object network and updates the Schematic View graph. During the traversal, the actual Animatable objects (object's derived from Animatable) are responsible for adding themselves to the Schematic View and continuing the traversal by recursively calling **SvTraverseAnimGraph(...)**. In practice, many objects in the scene are not added to the schematic view. In general, these omitted objects are either internal data structures that might confuse the user if displayed or they are objects whose sheer quantity might overwhelm the user (vertices, for example). Furthermore, although the Schematic View requires that the represented objects be derived from Animatable, there are very few restrictions on the reference pointers between objects (called "node pointers" in this section to distinguish them from 3ds max references). In particular, node pointers need not necessarily correspond to actual 3ds max references (though in practice they usually do). This flexibility allows the schematic view to represent object relationships in a manner different from the actual internal reference relationships. This can be useful in places where the internal structure does not agree with the user's notion of how the objects are structured (e.g., a linked list displayed as siblings under a parent node, or vice-versa).

## **Topology**

The topology of the objects added to the schematic view must be a directed acyclic graph (DAG). Cyclic references are currently not allowed.

Currently, nodes in the schematic view are arranged in tree form even if the underlying topology is a DAG. This is accomplished by duplicating shared nodes in the DAG. The DAG is essentially "flattened" into a tree. This is also how the track view works but, as will be explained in detail later, the schematic view stores additional information about the context of "shared" nodes which allows greater insight into instance relationships in the tree (multiple objects referencing a common node).

## The Schematic View API

There currently are three interfaces into the schematic view objects.

**IGraphObjectManager**, **IGraphNode**, and **IGraphRef**.

**IGraphObjectManager** essentially represents an instance of a schematic view window and provides methods for adding nodes and node pointers, refreshing the schematic view, accessing filter bits and updating and controlling the various editors within 3ds max in ways that are not surfaced in the general interface. **IGraphNode** represents a node in the schematic view graph and provides a few methods for querying information about the node. Finally, **IGraphRef** represents a node pointer and, currently, has no methods. Each of these three interfaces is described in greater detail below.

In addition to the three schematic view interfaces, there is a set of methods in **Animatable** that can be overridden, all or in part, to specialize the behavior of the schematic view node(s) which represents the Animatable object. Most of these methods are appropriately implemented for the various abstract base classes from which plug-in developers derive their classes. In most cases, there is no need for the plug-in developer to override any of these methods. All of the schematic view **Animatable** methods are prefixed by "Sv". In addition to the "Sv" methods, there are a half dozen or so "Sv" prefixed functions in **Animatable** which are called to perform various schematic view related tasks. They are implemented in **Animatable** instead of **IGraphNode** because they are more closely associated with the **Animatable** than with the graph node or because they are called in places where the **IGraphNode** and **IGraphObjectManager** interfaces are not available.

Taken as a whole, the three interfaces (**IGraphObjectManager**, **IGraphNode**, and **IGraphRef**) along with the "Sv" **Animatable** methods and functions represent about 95% of the complete schematic view API and 100% of the public (plug-in developer) API. The other 5% or so represents a small collection of functions scoped to the application and an interface, **ISchematicView**, which is used by the application code to control the schematic view window.

In general, the **IGraphObjectManager**, **IGraphNode**, and **IGraphRef** interfaces are not accessible to plug-ins outside of the scope of the Animatable "Sv" methods and should not be cached by the plugin.

## **Overview of the Chain of Events in Schematic View**

The events that occur in the schematic view are either generated by the user (clicking on a node, etc.) or transmitted to the schematic view via a **GlobalReferenceMaker::NotifyRefChanged(...)** message.

In response to these events, the schematic view performs one or more of the "Sv" methods on the effected nodes. The **IGraphObjectManager**, **IGraphNode**, and **IGraphRef** interfaces are passed to the "Sv" methods where appropriate.

The "Sv" methods perform some action and/or return some value and/or call one or more methods in the passed schematic view interface(s).

For example, the user selects a modifier in the schematic view and hits the "Delete" key. In response to the delete request, the schematic view calls the "**bool SvRemoveThis(IGraphObjectManager \*gom, IGraphNode \*gNode)**" method in Modifier which, in turn, calls the "**DeleteModifier(IGraphNode \*gNode)**" method in the **IGraphObjectManager** interface.

# Snapping

See Also: [Class Osnap](#), [Class IOSnapManager](#), [Class HitMesh](#), [Class OsnapHit](#), [Class OsnapMarker](#), [Structure SnapInfo](#), [Class Interface](#), [Class ViewExp](#).

## Overview

This topic presents information on object snapping in MAX. Object snaps are a plug-in type that allow objects to provide the 3ds max snapping system with additional points that may be snapped to. For example, the built in snap system provides the ability to snap to points such as vertices, endpoints, midpoints, etc. A developer of a procedural sphere object might want to provide additional snaps for center point and quadrant points. By deriving a class from **Osnap** and implement its methods a developer can seamlessly provide this functionality in MAX.

The 3ds max scene has a single object called the **Osnap Manager**. At 3ds max startup it will load any object snap plug-ins (those with a **DLS** extension). Object snap plug-ins are derived from class **Osnap**. This class has various methods that allow the system to query it about potential objects to be snapped to. That is, when 3ds max is traversing the scene doing a hit test the Osnap Manger will call each of the plug-ins to snap against a particular node. The **Osnap** class has a method that allows it to tell the Osnap Manager if it wants to snap to the specified object type based on its Super Class ID and Class ID. For example, NURBS snaps only respond to NURBS objects. Tangent and Perpendicular snaps only respond to Spline objects. If the object snap plug-ins do want to snap to the object type, they can register a set of points to snap to. For instance, the sphere above would register the center and quadrant points. This system allows developers to implement very specific snapping behaviors. For example, a door object might have a special snap mode that always snaps to the hinge point.

## Principal Classes

The following are the main classes associated with creating object snap plug-ins.

### [Class Osnap](#)

This is the base class for creating osnap plug-ins. Conceptually, the osnap class represents a "rule" for locating points in an object's local space. Typically, an instance of this class will only make sense for certain object types. It's the job of the **ValidInput()** method to filter out uninteresting nodes in the scene. When the scene is traversed, each object which passes the input test will be passed into the **Snap()** method. This method is the workhorse of object snap plug-ins and is responsible for computing, allocating and recording its hits.

### [Class IOsnapManager](#)

This class provides an interface to the Osnap Manager. Developers who implement osnaps need to record hits with the osnap manager.

### [Class OsnapHit](#)

This class encapsulates the data required to record a snapped point. Typically a plug-in creates instances of this class and records them with the Osnap Manager. If a snap plug-in needs to record additional data for its hits, it should derive from this class and provide a clone method which copies this additional data and calls the base classes clone method.

### [Class OsnapMarker](#)

This class is used for drawing osnap markers in the viewports. The marker is drawn as a polyline. The Osnap class must implement the **GetMarkers()** method which typically returns pointers to these static instances.

### [Class HitMesh](#)

This is a class to hold a list of object space points for highlighting the geometry associated with a hit. In practice, developers need only implement two methods of this class.

### [Class Interface](#)

There are several methods in this class related to object snap. The most important one returns a pointer to the Osnap Manager. See Interface Osnap Methods.

## Handling Snap Preview and Point Snap in Creation Procedures

The **ViewExp::SnapPoint()** method has been enhanced in 3ds max 2.0 to work with the new object snap system. A change that developers will need to make is that any mouse procs which will be calling **ViewExp::SnapPoint()** will need to add a call to **ViewExp::SnapPreview()** as in the following code fragment from `\MAXSDK\SAMPLES\OBJECTS\CYL.CPP`.

```
int CylinderObjCreateCallBack::proc(ViewExp *vpt,int msg, int
point,
    int flags, IPoint2 m, Matrix3& mat ) {
float r;
if (msg == MOUSE_FREEMOVE) {
vpt->SnapPreview(m,m,NULL, SNAP_IN_3D);
}
if (msg==MOUSE_POINT||msg==MOUSE_MOVE) {
switch(point) {
case 0:
ob->suspendSnap = TRUE;
sp0 = m;
p[0] = vpt->SnapPoint(m,m,NULL,SNAP_IN_3D);
...

```

## **Sample Code**

Some simple sample code provided in the SDK for demonstrating object snap is the sphere object snap plug-in. It provides two additional snaps to MAX. These are the center of the sphere and the quadrant points on the sphere. This code is available in **\MAXSDK\SAMPLES\SNAPS\SPHERE\SPHERE.CPP**.

The main code where snapping is handled for Geometric Objects, Shape Objects, Cameras, Lights and Helpers is available in **\MAXSDK\SAMLES\SNAPS\XMESH\XMESH.CPP**.

## Space Warp Plug-Ins

See Also: [Class Modifier](#), [Class ModContext](#), [Class WSMObject](#), [Class WSMModifier](#).

In addition to Object Space Modifiers (such as Bend, Taper, Skew, etc.), 3ds max supports another type of modifier referred to as a Space Warp. Another name for this type of modifier is a World Space Modifier. Example space warps are Ripple, Wave and Bomb.

A World Space Modifier (WSM) has two components: A WSM Object and a WSM Modifier. A WSM object is just another type of procedural object (derived from **WSMObject**). It exists in the scene and can have modifiers (including other WSMs) applied to it. When an item is bound to a WSM object, a WSM modifier (derived from **Modifier** or **WSMModifier**) is created and inserted in the object's history.

World space modifiers are very similar to regular object space modifiers -- in fact, they are derived from the same base class: **Modifier**. They have a mod app that is actually in the pipeline that refers to the modifier, although WSM modifiers aren't instanced. When a user binds more than one object to a WSM object at once, a unique WSM modifier is created for each object so there is a one to one correspondence between WSM modifiers and their mod apps. This is done because a WSM modifier may contain parameters specific to that application of the modifier. For example, the Ripple world space modifier has a 'Flexibility' parameter which allows the affect of the Ripple to be scaled on a per application basis.

The main difference between a regular object space modifier and a WSM modifier is that WSM modifiers usually have a reference to the WSM object's node. Also, WSM modifiers are applied after the object has been transformed into world space.

A WSM modifier references its corresponding WSM object's node so it can get the node's world space transformation matrix. It uses this matrix to transform the points of the object it is deforming into the space of the WSM object where it actually performs the deformation. After the deformation is applied, the points are transformed back to world space. This is similar to the way object space modifiers transform their points into the space defined by the ModContext transformation matrix -- however this space is defined by the WSM object's node. So in general, if an object's node is moved, the object appears to move

through the deformation field defined by the WSM object it is bound to. If the WSM object moves, a similar effect is apparent. Of course, if an object and the WSM object it is bound to move together (if one is a child of the other, for example) then the affect appears to be more like an object space modifier.

Also of note is that WSM modifiers (actually their mod apps) live in special world space derived objects. These are similar to the derived objects that hold object space modifier mod apps, except that these are contained in a specific node. Since they are associated with a specific node, they cannot be instanced.

There are two places where WSM user interface parameters can appear:

In the parameters of the WSM object. Any centralized parameters should be contained in the WSM object. Since WSM modifiers are never instanced, any parameters that the user would want to change and affect all objects bound to the WSM would belong here. For example, the Ripple WSM has its

Amplitude, Phase, WaveLength and Decay parameters in the WSM Object.

In the parameters of the WSM modifier. These parameters are specific to the object bound to the WSM. For example, Ripple has its flexibility parameter here. Often time these modifiers won't need to have any parameters at all. An example might be a dynamics WSM that has modifier parameters for the objects mass and surface characteristics.

# Style Guidelines for Creating Pipeline-Friendly Meshes

See Also: [Working with Meshes](#), [Class MNMesh](#).

3ds max meshes allow the developer a great deal of flexibility. There are very few rules. Meshes can have holes in them, such as you get when deleting any face from a box or a sphere. They may be composed of separate, disconnected units, which may even self-intersect. A 'fan' of 10 faces can use the same edge, simply by using the same two vertices. A grid may be created which is simply a 2-D collection of points with one set of faces pointing up and another pointing down.

However, not all of the possible meshes are appropriate for operations that may be done to them later in the 3ds max pipeline. For instance, the 3ds max Boolean, like any other Boolean, regards meshes as the outside boundary of solid objects. A Boolean intersection, for instance, is defined by the intersection of the volumes contained by the two operands. A mesh with a hole in it, such as the teapot, contains no volume, and therefore produces surprising results in Boolean.

Furthermore, Boolean, like several other compound objects and modifiers, converts meshes internally into MNMeshes. The **MNMesh** winged-edge structure can only handle 2 faces on a single edge, so the 'fan' described above gives it some difficulty. A mesh that doesn't gracefully convert to an **MNMesh** won't always behave as expected in MeshSmooth, Tesselate, ShapeMerge, Connect, Boolean, Slice, and possibly many more to come.

Also, if a mesh is not constructed in a straightforward manner, the user may experience some difficulty working with it in modifiers such as Edit Mesh. Of course, the user can construct very strange meshes themselves in Edit or Editable Mesh, but it's best if parametric objects and modifiers avoid introducing this sort of complexity when it isn't needed.

To that end, the following style guidelines for creating pipeline-friendly meshes in 3ds max are presented. None of these are hard-and-fast rules, but the more you are able to follow, the more flexible your mesh will be within the 3ds max environment.

## 1. Reference each edge at most once in each direction.

3ds max meshes don't really have edges, so what this means is that if you use

the vertices (a,b) in a face, you should only have one other face that uses both of these vertices, and it should reference them in the other direction (b,a). This other face is the 'other side' of the edge (a,b). If the same edge is used more than twice, or more than once in the same direction, MeshSmooth and other MNMesh-based effects have to compensate by duplicating the vertices used in the edge, once for each extra face using this edge.

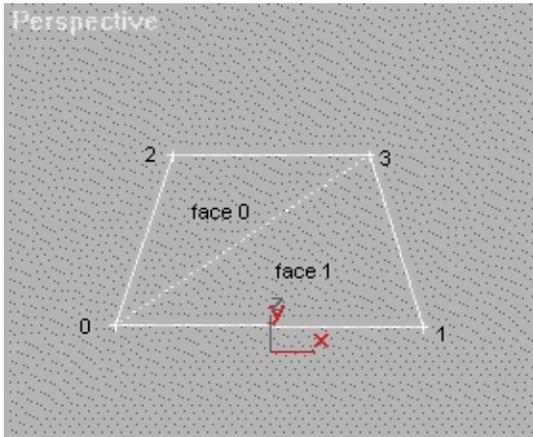


Figure 1A: Typical use of an edge

Face 0, which has vertices (0,3,2), uses the hidden edge in the (0,3) direction. Face 1, with vertices (0,1,3), uses the same edge in the (3,0) direction. The other edges are '1-sided', since there's no face on their other side.

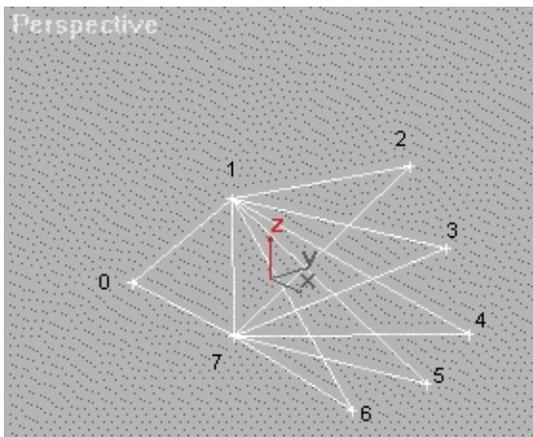
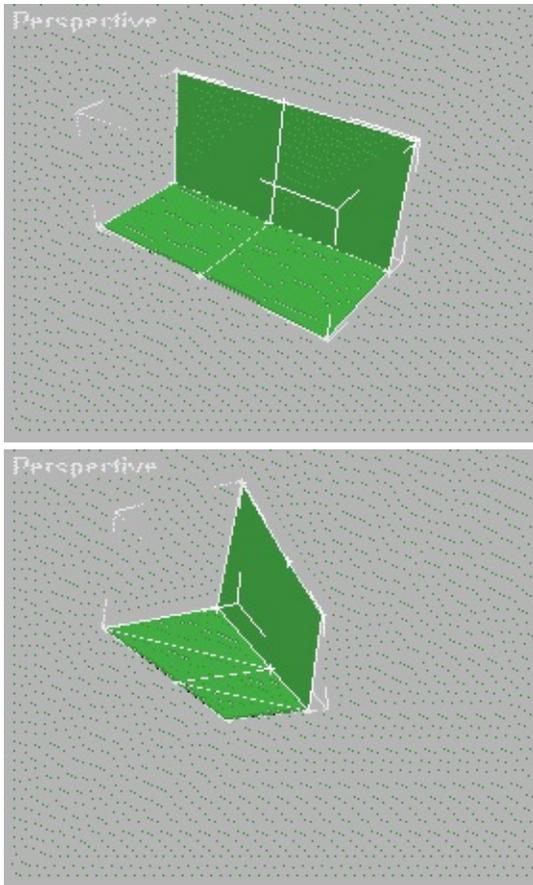


Figure 1B: Overuse of an edge

The face on the left uses the central edge in the (7,1) direction; the other 5 use it in the (1,7) direction. This can be inconvenient for the user and for many pipeline modifiers. Also, it generally doesn't respond well to smoothing groups.

Another situation in which edges are referenced too many times is when a zero-thickness mesh is constructed, with faces on both sides.



The above two views show the same zero-thickness mesh, with 9 points, that has faces representing both the front and back. The edges connecting the outside to the central point are used twice by front-facing faces and twice by back-facing edges. This also can cause problems in the pipeline.

Additionally, some systems may not display edges properly, as shown above at right: the edge connecting the top middle and base middle vertices should be visible, but doesn't show up because both the front-facing and back-facing faces have the same Z-depth.

## 2. Avoid self-intersection

For an example of self-intersecting meshes, we need look no further than the standard 3ds max primitive Teapot. Each of the four Teapot components is a separate mesh element, but the handle and spout elements both intersect the main body. Even if this mesh is closed with the CapHoles modifier, it causes trouble for Booleans because once again, what is 'inside' and what is 'outside' the teapot isn't very well defined.

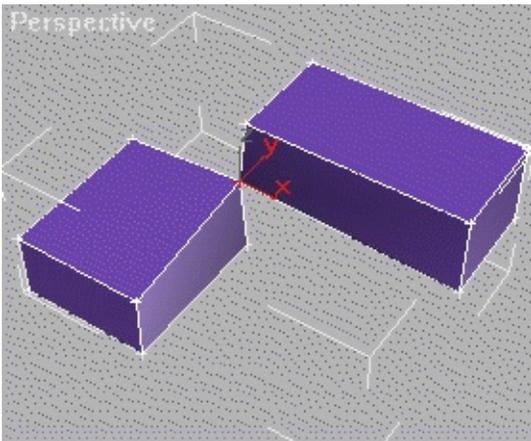
For an alternative to self-intersection, when (as in the teapot) computing the proper, connected result would be taxing, you may want to use the Boolean

itself within the SDK. If separate mesh elements are closed and otherwise acceptable to Boolean, you can just Union them together. Boolean is a bit slow, however, and there are still some known cases where it actually produces an open result. See the **MakeBoolean** method in class **MNMesh** ([MNMesh Boolean Operations](#)).

**3. Avoid creating faces with vertices located at the same place.**

A face referencing the same vertex twice, or two different vertices located in the same place, makes no contribution to your mesh and can interfere with display and rendering. Also, it confuses algorithms that traverse faces, such as ShapeMerge or Editable Mesh's new Cut function. Finally, Edit Mesh users can find it very frustrating to have to sort through multiple vertices at the same location in order to move the one they want.

**4. Try not to 'bridge' separate mesh components with a single vertex.**



The above two boxes share a single vertex, which is selected. Such vertices are split by **MNMesh** conversions. **MNMesh** effects deal with such vertices adequately, but the duplication is occasionally a surprise.

**5. If a mesh looks like a single element, it probably should be.**

A user shouldn't be able to take what looks like a single object apart by moving 'element' selections in Edit Mesh. If a model looks like a box, it shouldn't behave like a collection of 6 separate faces. This will give the user more reasonable results not only in modifiers like MeshSmooth, but also when they edit the object in Edit Mesh.

**6. Whenever convenient, close your meshes.**

A 'closed' mesh is one in which every edge has exactly two faces: one on each side. Closed meshes represent 3-dimensional objects. Open meshes are mere shells. Closed objects respond well to Boolean operations; open ones

can't, since there's no way for the Boolean to know which parts of operand A are supposed to be 'inside' operand B, and vice versa. Closed meshes have measurable volume; open ones do not.

This style point is less important than the others, because users are used to dealing with it. Any user who has models in Editable Mesh, for instance, deletes faces and constructs new ones all the time. However, avoid catching users off-guard: try not to leave tiny holes that can't be seen, say by putting open edges very close to each other. Users may not see them, but any hole will cause problems in pipeline effects that depend on solid, closed meshes, and may affect rendering adversely.

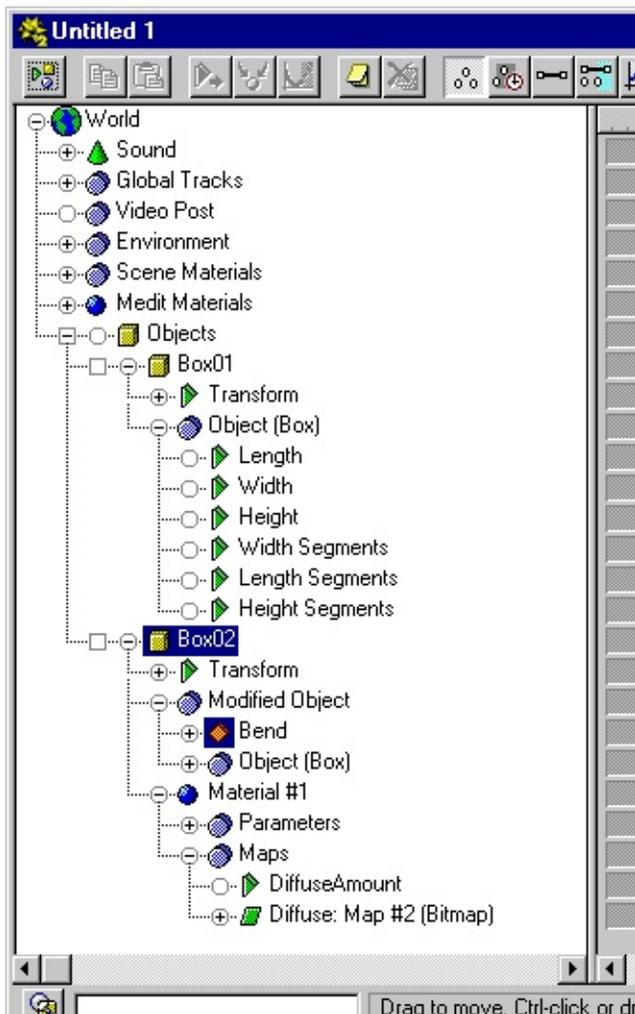
## Sub-Anims

See Also: [Track View](#), [Class Animatable](#), [Parameter Blocks](#).

## Overview

In the SDK the term **anim** refers to something derived from class **Animatable**. This anim can be anything such as a node, object, controller, material, texmap, parameter block, etc. Further, any item that has sub-items has what are referred to as **sub-anims**. This hierarchy of items, the parent item and it's sub-items, is referred to as the sub-anim hierarchy. The sub-anim hierarchy can be thought of as the Track View hierarchy.

Below is a partial screen capture of Track View showing a few anims and their sub-anims.



For example, a node has six sub-anims. These are: Space Warp Bindings, the Transform Controller, the Object Reference of the node, the Material, the Visibility Controller, the Image Motion Blur Controller. If a node does not have some of these assigned (for example the material and visibility controller) these sub-anims are NULL. Any sub-anims that are NULL don't appear in Track View.

In the example above, the node **Box01** has two sub-anim s that are non-NULL: the transform controller (labeled **Transform**) and the object reference (labeled **Object(Box)**). The other node, **Box02**, has three non-NULL sub-anim s. These are the transform controller (**Transform**), the derived object (**Modified Object**), and the Material (**Material01**).

Note that sub-anim s may have sub-anim s themselves. This can be seen in the parameters of **Box01** sub-anim **Object(Box)**. Procedural objects derived from **SimpleObject** (like **Object(Box)**) have a single sub-anim and that is the parameter block. The box's parameter block has six sub-anim s (Length, Width, etc.). Note also that the node **Box02** sub-anim **Material #1** has sub-anim s of its own. These are its parameters and its texture maps.

Sub-Anim s are often the animatable parameters of a plug-in. In 3ds max any animated parameter has a controller to control the animation. The controller for a parameter appears in the track view. This allows the user to view the animation associated with a parameter in several formats such as in key frames, as a range of time over which the animation takes place, and graphically as a function curve.

3ds max accesses the sub-anim s of a plug-in using a virtual array mechanism. That is, each sub-anim is given an index and is accessed by a method that returns a pointer to the 'i-th' sub-anim. These methods are **Animatable::NumSubs()**, **Animatable::SubAnim(i)** and **Animatable::SubAnimName(i)**.

## Methods

The plug-in can store its parameters in any way it chooses. When the system needs to access a parameter, it passes the plug-in an index and the plug-in returns the parameter it has assigned to that index. Below is a description of the three methods associated with sub-anims.

### **virtual int NumSubs();**

This method returns the total number of sub-anims maintained by the plug-in. If a plug-in is using a parameter block to manage its parameters it should just return 1 for all those parameters. Below is the **SimpleMod** implementation of this method. It returns 3 as it has a controller for the Center, the Gizmo, and a parameter block.

```
int NumSubs() {return 3;}
```

### **virtual Animatable\* SubAnim(int i);**

This method returns a pointer to the 'i-th' sub-anim. If a plug-in is using a parameter block to manage all of its parameters it should just return a pointer to the parameter block itself from this method. Below is the **SimpleMod** implementation of this method.

### **Animatable\* SimpleMod::SubAnim(int i)**

```
{  
  switch (i) {  
    case 0: return posControl; // Center  
    case 1: return tmControl; // Gizmo  
    case 2: return pblock; // Parameter block  
    default: return NULL;  
  }  
}
```

### **virtual TSTR SubAnimName(int i);**

This method returns the name of the 'i-th' sub-anim to appear in track view. The system has no idea what name to assign to the parameter (it only knows it by the array index), so this method is called to retrieve one to display. Below is the **SimpleMod** implementation of this method. Note that it returns a descriptive name for each controller, i.e. "Center" for the center mark position controller, and "Gizmo" for the gizmo controller. However it just returns "Parameters" for the parameter block. The parameter block name itself does

not show up in track view, only its sub-controllers. This is because the parameter block implements a method that tells the system to not show it in track view. This prevents the user from having to navigate another nested level to simply get to the parameters.

```
TSTR SimpleMod::SubAnimName(int i)  
{  
  switch (i) {  
    case 0: return TSTR(_T("Center"));  
    case 1: return TSTR(_T("Gizmo"));  
    case 2: return TSTR(_T("Parameters"));  
    default: return TSTR(_T(""));  
  }  
}
```

## Summary

The sub-anim hierarchy of a plug-in corresponds to the Track View hierarchy. Any non-NULL sub-anim that a plug-in has appear in Track View as branches under the parent item. A developer implements three methods of the **Animatable** class, **NumSubs()**, **SubAnim(i)** and **SubAnimName(i)** to provide 3ds max with access to its sub-anim.

## Sub-Object Selection

See Also: [Class CommandMode](#), [Class Interface](#), [Class XFormModes](#), [Class BaseObject](#), [Class Modifier](#).

Certain modifiers may wish to provide the system with different levels of sub-object selection. There are two general types of sub-object selection:

Selection of the modifier's gizmo. A modifier's gizmo is a visual representation in the scene the user can manipulate. This can be something as simple as the 3D box used by the bend modifier or something more complex like a FFD lattice. The Bend modifier provides sub-object levels of Center and Gizmo.

Selection of components of the object flowing through the geometry pipeline. If the object in the pipeline is a TriObject, for example, the sub-object selection levels may be things such as vertices, faces, and edges. The Edit Spline modifier supplies sub-object levels for Spline, Segments and Vertex.

A modifier registers its sub-object selection levels when its

**BeginEditParams()** method is called. A method of the Interface class named **RegisterSubObjectTypes()** handles this. It provides the system with a list of strings. These strings are the names of the sub-object selection levels that will appear in the Sub-Object drop down list. For example, the following code would register two sub-object levels, "Gizmo" and "Center".

```
const TCHAR *ptype[] = { "Gizmo", "Center" };  
ip->RegisterSubObjectTypes( ptype, 2 );
```

When the user presses the Sub-Object button or changes the selection in the sub-object drop down list the modifier is notified. The system calls the Modifier's

**ActivateSubobjSel()** method. In its implementation of this method the modifier is expected to provide an instance of a class derived from

**CommandMode** to support Move, Rotate, Uniform scale, Non-uniform scale, and Squash. These modes replace their object level counterparts although the user still uses the move/rotate/scale toolbar buttons to activate these modes.

Certain sub-object levels may not support all of these modes. For example, the Bend modifier (derived from **SimpleMod**) has two sub-object levels: "Gizmo" and "Center". The Gizmo level supports Move, Rotate, Non-uniform scale, Uniform scale and Squash while the Center level only supports Move. Any modes that the modifier does not support will have the corresponding button grayed out in the toolbar.

In addition to move/rotate/scale, a plug-in may provide other modes. For instance the Edit Mesh modifier has an extrude mode where the user can click and drag to interactively extrude faces of a mesh.

Users will expect sub-object move, rotate and scale transformations to behave like their object level counterparts. To make this very easy to implement, there are a set of standard modes defined by the system that a developer may use. These modes handle all this logic internally -- the developer simply uses them. Most modifiers will be able to use these modes. The sample code below demonstrates how the modes are used. This example is from the implementation of class **SimpleMod**.

In the class declaration the mode pointers are defined:

```
class SimpleMod : public Modifier {  
    ...  
    protected:  
    static MoveModBoxCMode *moveMode;  
    static RotateModBoxCMode *rotMode;  
    static UScaleModBoxCMode *uscaleMode;  
    static NUScaleModBoxCMode *nuscaleMode;  
    static SquashModBoxCMode *squashMode;  
    ...
```

In the **BeginEditParams()** method the modes are allocated:

```
void SimpleMod::BeginEditParams( IObjParam *ip, ULONG  
flags,  
    Animatable *prev )  
    {  
        ...  
        // Create sub object editing modes.  
        moveMode = new MoveModBoxCMode(this,ip);  
        rotMode = new RotateModBoxCMode(this,ip);  
        uscaleMode = new UScaleModBoxCMode(this,ip);  
        nuscaleMode = new NUScaleModBoxCMode(this,ip);  
        squashMode = new SquashModBoxCMode(this,ip);  
        ...  
    }
```

In the method **ActivateSubobjSel()** the modes are activated. This method is defined in [BaseObject](#).

```
void SimpleMod::ActivateSubobjSel(int level, XFormModes&
modes )
{
switch ( level ) {
  case 1: // Modifier box
    modes = XFormModes(moveMode,rotMode,nuscaleMode,
      uscaleMode,squashMode,NULL);
    break;
  case 2: // Modifier Center
    modes =
XFormModes(moveMode,NULL,NULL,NULL,NULL,NULL);
    break;
  }
  ...
}
```

When the item is finished being edited the modes can be deleted:

```
void SimpleMod::EndEditParams( IObjParam *ip, ULONG flags,
Animatable *next)
{
  ...
ip->DeleteMode(moveMode);
ip->DeleteMode(rotMode);
ip->DeleteMode(uscaleMode);
ip->DeleteMode(nuscaleMode);
ip->DeleteMode(squashMode);
if ( moveMode ) delete moveMode;
moveMode = NULL;
if ( rotMode ) delete rotMode;
rotMode = NULL;
if ( uscaleMode ) delete uscaleMode;
uscaleMode = NULL;
if ( nuscaleMode ) delete nuscaleMode;
```

```
nuscaleMode = NULL;  
if ( squashMode ) delete squashMode;  
squashMode = NULL;  
}
```

For a complete list of the classes that may be used see [List of Standard Sub-Object Modes](#).

Both the standard transformation modes and custom modes are expected to treat sub-object selection in the same way 3ds max handles object selection. This means all the keypress options, such as Ctrl to add to a selection and Alt to subtract from a selection, should work in the same manner. To avoid forcing plug-in developers to implement all this logic, a **SelectionProcessor** class is provided. This class represents a mouse proc derived from the **MouseCallback** class. When you create an instance of the **SelectionProcessor** class you give its constructor a pointer to another **MouseCallback** -- the mouse proc to handle the mode being implemented. The **SelectionProcessor** receives the mouse input first. It processes the mouse input using a standard selection logic and then calls the **MouseCallback** it was given.

The sample code below is from `\MAXSDK\SAMPLES\MESH`. It shows how the extrude command mode of the Edit Mesh modifier uses the **SelectionProcessor** class to handle selection yet performs its own work after the selection process has been handled. The extrude command mode allows the user to select items with the mouse using all the standard logic such as box selection, and using the Ctrl and Alt keys. If the user drags the mouse however, the extrude operation is performed. By using the selection processor all the logic for selection is handled automatically.

The overall manner this is set up is as follows:

A command mode has a method **MouseProc()** that returns an instance of **MouseCallback** to handle the user / mouse interaction. In the extrude command mode's implementation of **MouseProc()** it returns an instance of a selection processor (which is derived from **MouseCallback**). The constructor for the selection processor takes a pointer to another **MouseCallback**. This is the extrude mouse callback. When the command mode is processing mouse input, the selection processor mouse callback is called first. It processes all the logic for the selection. If the user drags the mouse however the selection processor calls the extrude mouse callback to handle the extrude operation.

Below are the class definitions showing how this is set up.

The mouse callback below implements the user / mouse interaction for the extrude operation.

```
class ExtrudeMouseProc : public MouseCallBack {  
  private:  
    MoveTransformer moveTrans;  
    EditMeshMod *em;  
    IObjParam *ip;  
    IPoint2 om;  
  public:  
    ExtrudeMouseProc(EditMeshMod* mod, IObjParam *i)  
      : moveTrans(i) {em=mod;ip=i;}  
    int proc(  
      HWND hwnd,  
      int msg,  
      int point,  
      int flags,  
      IPoint2 m );  
  };
```

The code below defines a class from **GenModSelectionProcessor**. The constructor passes on the mouse proc for the extrusion interaction to the base class **GenModSelectionProcessor**. This allows the selection processor to call the extrusion mouse proc.

```
class ExtrudeSelectionProcessor : public  
GenModSelectionProcessor {  
  protected:  
    HCURSOR GetTransformCursor();  
  public:  
    ExtrudeSelectionProcessor(ExtrudeMouseProc *mc,  
      Modifier *m, IObjParam *i)  
      : GenModSelectionProcessor(mc,m,i) {}  
  };
```

The command mode declares an instance of the selection processor and extrude mouse proc. The extrude mouse proc is passed into the selection processor.

```

class ExtrudeCMode : public CommandMode {
private:
    ChangeFGObject fgProc;
    ExtrudeSelectionProcessor mouseProc;
    ExtrudeMouseProc eproc;
    EditMeshMod* em;
public:
    ExtrudeCMode(EditMeshMod* mod, IObjParam *i) :
        fgProc(mod), mouseProc(&eproc,mod,i), eproc(mod,i)
    {em=mod;}
    int Class() { return MODIFY_COMMAND; }
    int ID() { return CID_EXTRUDE; }
    MouseCallback *MouseProc(int *numPoints)
        { *numPoints=2; return &mouseProc; }
    ChangeForegroundCallback *ChangeFGProc()
        { return &fgProc; }
    BOOL ChangeFG( CommandMode *oldMode )
        { return oldMode->ChangeFGProc() != &fgProc; }
    void EnterMode();
    void ExitMode();
};

```

See the source code in **EDITMOPS.CPP** for the full implementation of the methods of these classes.

## Modifier Methods

The overall process of sub-object selection relies upon the implementation of several methods in the modifier. These methods are from class **BaseObject**. These involve selecting, identifying, moving, rotating and scaling sub-object components. These methods are:

**virtual void ClearSelection(int selLevel);**

This method is called to clear the selection for the given sub-object level. All sub-object elements of this type should be deselected.

**virtual void SelectSubComponent(HitRecord \*hitRec, BOOL selected, BOOL all, BOOL invert=FALSE)**

This method is called to change the selection state of the component identified by **hitRec**.

**virtual int SubObjectIndex(HitRecord \*hitRec);**

Returns the index of the sub-object element identified by the HitRecord **hitRec**. The sub-object index identifies a sub-object component. The relationship between the index and the component is established by the modifier. For example, an edit modifier may allow the user to select a group of faces and these groups of faces may be identified as group 0, group 1, group 2, etc. Given a hit record that identifies a face, the edit modifier's implementation of this method would return the group index that the face belonged to.

**virtual void CloneSelSubComponents(TimeValue t);**

This method is called to make a copy of the selected sub-object components.

**virtual void Move(TimeValue t, Matrix3& partm, Matrix3& tmAxis,**

**Point3& val, BOOL localOrigin=FALSE )**

When this method is called the plug-in should respond by moving its selected sub-object components.

**virtual void Rotate(TimeValue t, Matrix3& partm, Matrix3& tmAxis,**

**Quat& val, BOOL localOrigin=FALSE )**

When this method is called the plug-in should respond by rotating its selected sub-object components.

**virtual void Scale(TimeValue t, Matrix3& partm, Matrix3& tmAxis,**

**Point3& val, BOOL localOrigin=FALSE )**

When this method is called the plug-in should respond by scaling its selected sub-object components.

## Sub-Object Selection in Edit Modifiers

A certain class of modifiers, referred to as edit modifiers, allow specific object types to be edited. For example, the Edit Mesh modifier allows TriObjects to be edited while the Edit Patch modifier allows patch objects to be edited. A generic Edit Point modifier would edit any object that supports the deformable interface. The sub-object selection levels for these edit modifiers reflect the sub-object types for the object type that the edit modifier is designed to edit. For example, the Edit Mesh modifier supports three sub-object selection levels: vertex, face and edge. These correspond to actual elements of a TriObject. In contrast, the sub-object levels of the Bend modifier, are gizmo and center. These have no correspondence with the object that the bend modifier is modifying.

Edit modifiers typically allow the user to select sub-object elements of the object and perform at least the standard move/rotate/scale transformations. They may also support other operations such as the Edit Mesh modifier's extrude operation.

When the user creates sub-object selections with an edit modifier, they are typically stored in the object because they correspond to actual parts of the object. The sub-object selection is actually part of the object's state that flows up the pipeline. For example, the Edit Mesh modifier allows the user to select vertices, faces or edges. If a Bend modifier is then applied after the Edit Mesh modifier, it will operate on the selected vertices or faces. If the user goes back in the history to the Edit Mesh modifier and changes the selection, the set of vertices that the bend affects will change. Essentially just editing the selection is a form of editing an object because the selection is part of the object.

Edit modifiers have to store the selection themselves. If the input to an edit modifier changes, the modifier will need to reapply its modifications. Since changing the selection set is a modification to the object, the edit modifier will need to recreate the selection set.

Selection sets are not the only thing edit modifiers need to store. If an edit modifier moves points on a modifier, then it needs to store, in some form, the changes it made to the points. For example, the Edit Mesh modifier stores changes made to the vertices of a mesh by maintaining an array of 3D vectors that represent a delta for each vertex. This delta is applied to an object by simply adding each delta to its corresponding vertex. This brings up a couple of issues: What happens if the number of vertices of the input object change? Since the vertex index is used to find its corresponding delta, if the topology of an object changes, a vertex's index might change. If this is the case the user is given a

warning message that a modifier exists in the stack that depends on topology (see the method **Modifier::DependOnTopology()** for the details on how an edit modifier informs the system it is dependent on topology). If the user selects OK from the warning message dialog the topology may be changed. The least that an edit modifier should do is not reference a vertex out of the range of an object's vertex array. The Edit Mesh modifier continually scales the length of its delta array to match the length of the vertex array of the incoming object. This ensures that the deltas are all applied to legitimate vertices, but if the topology changes, the effect is rarely useable.

The user may use the volumetric selector modifier to help eliminate this problem. This modifier uses spheres, 3D boxes, and cylinders to define a 3D region. Points inside the region become selected while points outside are not. This method of selection is independent of topology and therefore does not exhibit the previously-described problems.

Modifiers can be instanced and therefore a modifier, in general, isn't just applied to a single object. This raises the question of what happens when an edit modifier is instanced. It must store its changes to a particular object somewhere. One of the fields of the [ModContext](#) is a pointer to a LocalModData derived class. The most convenient place to store this data is here in the ModContext. When an object is copied, the ModContext (and therefore the LocalModData) is also copied making it possible to copy objects with edit modifiers applied to them.

## Named Sub-Object Selection Sets

A modifier that supports sub-object selection can choose to support named sub-object selection sets. This allows the user to save and restore certain sub-object selection sets and not have to pick them over and over. The modifier indicates it wishes to do so by returning TRUE from the following method of BaseObject.

### **virtual BOOL SupportsNamedSubSels();**

Returns TRUE if the plug-in supports named sub-object selection sets; otherwise FALSE.

A modifier that wishes to support this capability maintains its list of named sub-object selections. When the user enters sub-object selection mode the modifier should add its named selection sets into the drop down using the following method of Class Interface:

### **virtual void AppendSubObjectNamedSelSet(const TCHAR \*set)=0;**

A modifier may call this method to add sub-object named selection sets to the named selection set drop down list in the 3ds max toolbar. This should be done whenever the selection level changes (in the Modifiers

**BaseObject::ActivateSubobjSel()** method).

The following methods of BaseObject are called when the user picks items from the drop down list:

### **virtual void ActivateSubSelSet(TSTR &setName);**

When the user chooses a name from the drop down list this method is called. The plug-in should respond by selecting the set identified by the name passed.

### **virtual void NewSetFromCurSel(TSTR &setName);**

If the user types a new name into the named selection set drop down then this method is called. The plug-in should respond by creating a new set and give it the specified name.

### **virtual void RemoveSubSelSet(TSTR &setName);**

If the user selects a set from the drop down and then chooses Remove Named Selections from the Edit menu this method is called. The plug-in should respond by removing the specified selection set.

The following methods from Class Interface also deal with named sub-object selection sets:

### **virtual void ClearSubObjectNamedSelSets()=0;**

This method clears the named selections from the drop down.

**virtual void ClearCurNamedSelSet()=0;**

This method clears the current edit field of the named selection set drop down.

For sample code that deals with named sub-object selection sets see

**\\MAXSDK\\SAMPLES\\MODIFIERS\\EDITMESH.CPP.**

## Sub-Object Coordinate Systems:

See Also: [Class BaseObject](#), [Class Control](#), [Class SubObjAxisCallback](#).

When a modifier or controller is in a sub-object selection level, it has the ability to define the meaning of some of MAX's reference coordinate systems. MAX's reference coordinate system has two parts: The reference coordinate system drop down list and the center button. These are described below:

**1** The reference coordinate system drop down list. This lets the user select the orientation of the reference coordinate system. While the user is in a sub-object selection state, 'local' refers to the local coordinate system of the currently selected sub-object element and 'parent' refers to the node's local coordinate system.

What a sub-object's local coordinate system means is up to the modifier or controller. For example, Bend (and in fact, any modifier using **SimpleMod** as a base class) define local to mean the coordinate system of the gizmo. The Edit Mesh modifier defines local to mean different things for face and vertex selection levels. For face, the modifier constructs a coordinate system where the Z axis is parallel to the face's normal and X lies somewhere in the world XY plane perpendicular to Z. The Y axis is then given by X and Z. For vertices, a similar thing is done -- the vertex normal used for smoothing is used to define the Z axis. X and Y are constructed the same way as above.

**2** The center button. This button allows the user to choose the position of the center of the reference coordinate system. It has 3 options: Center of the selection set, Center of the reference coordinate system, and Pivot point. If the reference drop down list is set to local and the center button is set to center of the reference coordinate system, then the modifier or controller defines the position of the center when it constructs the local coordinate system. For center of selection and pivot point, it is up to the modifier or controller to decide what these mean. The Edit Mesh modifier defines center of the selection set to be the average of all vertices referenced by selected faces and the pivot point to be the center of a face cluster.

When the user is in a sub-object selection level, the system needs to get the reference coordinate system definition from the current modifier or controller being edited so that it can display the axes. Two methods (from **BaseObject** or **Control**) allow the system to do this:

**GetSubObjectCenters(SubObjAxisCallback \*cb, TimeValue**

**t,INode \*node)**

This method specifies the position of the center. The plug-in enumerates its centers and calls the callback **cb** once for each.

**GetSubObjectTMs(SubObjAxisCallback \*cb,TimeValue t,INode \*node)**

This method returns the axis system of the reference coordinate system. The plug-in enumerates its TMs and calls the callback **cb** once for each.

## Thread Safe Plug-Ins

See Also: [Main Plug-In Classes](#).

## Overview

Plug-ins that relate to the renderer have specific methods that need to be thread safe. This is because the renderer launches several threads of execution when rendering an image. Therefore the same method could get called from several threads at the same time. If one of the methods is writing a value that the other is reading or writing this can interfere with valid values being accessed.

Plug-in materials, texture maps and atmospheric effects are the plug-in types that have methods that need to be `= 4)`

`BSPSPopupOnMouseOver(event);">reentrant`. Any method of these plug-ins that is called while rendering, like **Texmap::EvalColor()**, **Mtl::Shade()**, **Atmospheric::Shade()**, etc. must be written in a way that is thread safe.

The methods listed above can be re-entered from another thread while the renderer is processing. For example, while **Texmap::EvalColor()** was executing it could get called again, totally asynchronously. At a certain time this method could be changing a variable, and another thread could be changing that same variable at the same time. Or it could be setting the value while another thread is attempting to read it. This can interfere with valid values being set or read.

The main rule for class variables is if a method changes a variable it won't be thread safe unless it takes specific steps to ensure it is. Any variables in the class must not be written to without taking steps. This is because both threads can see the same variable and could write to it at the same time.

There are conditions where a plug-in does NOT have to do anything to be thread safe. If the method just reads variables, it will be thread safe. Any local variables (those on the stack) can be read and written without concern. Any global variables can also be read without concern. If a method does all of its work on the stack then it will be thread safe.

## How To Ensure a Function is Thread Save

The code fragments below are from the 3ds max fog atmospheric effect. The code for this plug-in can be found in the SDK in `\MAXSDK\SAMPLES\HOWTO\MISC\FOG.CPP`. The synchronization object used to make the method thread safe here is called **CRITICAL\_SECTION**. Developers can look in the Win32 API for documentation for this and other synchronization objects.

First a developer declares a **CRITICAL\_SECTION**. This is a data type that is declared, initialized, and deleted when one is finished with it. When the class is created, like in the class constructor this object is initialized. Below are a pair of code fragment from the Fog plug-in where the **CRITICAL\_SECTION** is declared and initialized:

This creates the **CRITICAL\_SECTION** object.

```
class FogAtmos : public StdFog {
    public:
        CRITICAL_SECTION csect;
        ...
}
```

Here it is initialized in the constructor.

```
FogAtmos::FogAtmos() {
    ...
    InitializeCriticalSection(&csect);
    ...
}
```

In code where data that is accessible to several threads is to be modified, you call a Win32 API named **EnterCriticalSection()** and pass the **CRITICAL\_SECTION** object.

For example, thread A enters the code and **EnterCriticalSection()** is called. This sets a state in the **CRITICAL\_SECTION** data structure. Now thread B enters the same section of code. It also calls **EnterCriticalSection()** and passes in the **CRITICAL\_SECTION** object. The **CRITICAL\_SECTION** object knows that another thread is already in that code. Therefore it will sit and wait until thread A finishes and calls **LeaveCriticalSection()**. When thread A calls **LeaveCriticalSection()** this will trigger thread B that it may return from

**EnterCriticalSection()**. At this point it can then enter the code. Thus bracketing code with **EnterCriticalSection()** and **LeaveCriticalSection()** will make the code thread safe.

Below is a fragment from a method of the fog plug-in that is made thread safe using this technique.

```
void FogAtmos::UpdateCaches(TimeValue t)  
{  
  EnterCriticalSection(&csect);  
  if (!valid.InInterval(t)) {  
    valid = FOREVER;  
    pblock->GetValue(PB_COLOR,t,fogColor,valid);  
    ...  
  }  
  LeaveCriticalSection(&csect);  
}
```

Note that at the beginning of the method **EnterCriticalSection()** is called. Just before it exits **LeaveCriticalSection()** is called.

When a plug-in is done with the **CRITICAL\_SECTION** object, it can call **DeleteCriticalSection()**. Here this is done in the destructor for the Fog plug-in:

```
~FogAtmos() {  
  DeleteCriticalSection(&csect);  
}
```

Developers should also be aware of the following two points:

**1** Critical sections, or any other synchronization objects, should only surround the part of the code that requires synchronization. Otherwise, if the critical section surrounds a large block of computationally-expensive code unnecessarily, performance will suffer on multiprocessor machines.

As an example, if an algorithm needs to write to a class variable, do a long computation, then write to another class variable, it would be bad to structure the code like this:

```
Enter Critical Section  
write to var1  
do long computation...
```

write to var2

Leave Critical Section

This structure stops any other processor from accessing this routine until the long computation is done. It would be better to structure the code like this:

Enter Critical Section

write to var1

Leave Critical Section

do long computation...

Enter Critical Section

write to var2

Leave Critical Section

- 2 Synchronization problems often don't show up until you are running on a multi-processor system. All plug-in developers should test their code on multi-processor systems as much as possible. Intermittent crashes or erratic behavior is often a symptom of non-reentrancy.

## **Summary**

3ds max is a multi-threaded application and plug-ins that run inside 3ds max where multiple threads are used need to be thread safe. This section has presented the data structures and calls from the Windows Win32 API used to create thread safe code.

# Time

See Also: [Time Function Reference](#).

This section discusses the concept of time as used by 3ds max.

Time is stored internally in 3ds max as an integer number of **ticks**. Each second of an animation is divided into 4800 ticks. This value is chosen in part because it is evenly divisible by the standard frame per second settings in common use (24 -- Film, 25 -- PAL, and 30 -- NTSC).

The data type used to store a specific instance of time is the **TimeValue**. A **TimeValue** stores the number of ticks the time represents. When a developer specifies time to almost all the functions in the SDK they use **TimeValues**.

In MAX, time may be displayed in one of four formats. These are just display formats (the internal representation never changes). The formats are:

**Frames** - Integer number of frames.

**SMPTE** - SMPTE time code. This format uses

**hours:minutes:seconds:frames**. 3ds max abbreviates this by not including the hours designation to **minutes:seconds.frames**.

**FRAME:TICKS** - Integer number of frames:number of ticks.

**MM:SS:TICKS** - Minutes:Seconds:Ticks.

A developer may get and set the current time displayed format. The functions to accomplish this are **GetTimeDisplayMode()** and **SetTimeDisplayMode()**. When these settings are changed a message is sent throughout 3ds max to handle updating any UI controls that reflect time settings (including plug-in custom controls).

A developer can retrieve the number of ticks per frame using the function **GetTicksPerFrame()**, and can set this number using **SetTicksPerFrame()**.

A developer can also get and set the current frame rate. This is the number of frames per second. The functions for this are **GetFrameRate()** and **SetFrameRate()**. This is just an alternate way to specify the number of ticks per frame.

There are also functions to convert between the numeric and ASCII representations of time. The functions for this are **TimeToString()** and **StringToTime()**.

To review the methods discussed above see [Time Function Reference](#).

Note that some methods related to time are part of various interface classes. For example [Class Interface](#) provides methods to retrieve the current frame slider setting and the current animation range. These methods are **GetTime()** and **GetAnimRange()**. You can also set these using **SetTime()** and **SetAnimRange()**. There is also a method **RegisterTimeChangeCallback()** that allows a plug-in to be notified every time the user changes the frame slider.

## Track View

See Also: [Class Animatable](#).

## Overview

The track view of 3ds max provides a view into time. Track view is organized as a nested hierarchy showing both parent/child relationships, and sub-anim (parameter) relationships.

Many different plug-in types may participate in track view. For example, system plug-ins, sound object plug-ins, and many controller types all may have interfaces in track view. The methods of the **Animatable** class allow a developer to participate. This section is an overview of the concepts of track view, and an outline of the specific methods involved in providing an interface.

The track view provides a hierarchical view of the scene. All controllers appear in the track view as children of the objects they control. For example, every node has a transform controller and this controller appears under the node in the track view. Also, procedural objects and modifiers appear in the track view. If these have animatable parameters, the controllers plugged in to those parameters appear underneath the object or modifier. If a controller itself has animatable parameters, then any controllers plugged into those parameters would be found underneath it in the track view.

On the right side of each item in the track view there is a long horizontal track window. In this window, a controller can represent the track it controls. For example, a keyframe controller draws marks in the track representing points in time that have keyframes. Other procedural controllers may only present a time line displaying the duration of the animation.

Each controller is also able to display a function curve. This may or may not be editable by the user depending on the type of controller. For example, the standard keyframe controllers allow the user to edit the function curve while the expression controller simply displays a graph of the function that the user specified. The user has the ability to overlay function curves for different controllers on top of each other, regardless of the types of the controllers. For example, the user could place a function curve that represents the red component of a spotlight on top of the function curve that represents the position along a path of one of the nodes in the scene and edit the two.

In addition to function curves and tracks, a modal dialog may be brought up by the controller that contains additional parameters which the user can edit.

## Appearing in Track View

By default an anim (controller, parameter, etc.) will appear in track view. If the anim should not, the following method may be used:

### **virtual BOOL BypassTreeView()**

This method indicates to the system that this item should not appear in the track view. Note: Track View was formally referred to as Tree View. The default implementation returns FALSE, so items will appear unless this method is implemented to return TRUE.

## Copy and Paste Operations

There are two types of copy and paste in track view. These are available using two buttons in track view dialog. There is the Copy Controller button, and in Edit Time Mode, there is the Copy Tracks button.

Copy Controller lets the user copy the entire controller plug-in. A method of **Animatable** is available to prevent an item from being copied.

### **virtual BOOL CanCopyAnim()**

If an animatable doesn't want to be copied in track view it can return FALSE from this method, otherwise it can return TRUE.

Copy Tracks lets the user select a block of time and copy keys. An item (usually a controller) can participate in the copy and pasting of tracks. To indicate that it will participate in copying, a controller implements the following method and returns TRUE:

### **virtual BOOL CanCopyTrack(Interval iv, DWORD flags)**

Returns TRUE if this item can copy its data over the specified range; otherwise returns FALSE.

If a plug-in can copy its track data, the following method will be called to allow the item to do so:

### **virtual TrackClipObject \*CopyTrack(Interval iv, DWORD flags)**

This method is called to copy the item's track data over the specified interval. A plug-in should derive a class from the **TrackClipObject** base class to store the data associated with the objects tracks, and implement the methods that identify the creator of the clip object.

When the user has asked to paste the track, the system will call the following method to see if the track data can be pasted to the item.

### **virtual BOOL CanPasteTrack(TrackClipObject \*cobj, Interval iv, DWORD flags)**

Returns TRUE if this item can paste its data over the specified range; otherwise returns FALSE. The item can check the **TrackClipObject** creator and see if the data is appropriate for it to accept.

If the data is okay, the following method will be called to paste the data from the **TrackClipObject** to the item.

### **virtual void PasteTrack(TrackClipObject \*cobj, Interval iv, DWORD flags)**

This method is called to paste the specified clip object to this track. This method will not be called unless **CanPasteTrack()** returned TRUE.

## Operations to Time

This section shows several of the methods of Animatable that relate to the manipulation of time in track view:

### **virtual BOOL SupportTimeOperations()**

If an item supports time operations in the track view (cut, copy, paste, etc.), it should implement this method to return TRUE. When it is FALSE the user cannot select blocks of time in the item's track.

### **virtual void DeleteTime(Interval iv, DWORD flags);**

This method is called to delete the specified interval of time (or the keys within the interval).

### **virtual void ReverseTime(Interval iv, DWORD flags);**

This method is called to reverse the data within the specified interval.

### **virtual void ScaleTime(Interval iv, float s);**

This method is called to scale an interval of time by the specified scale factor.

### **virtual void InsertTime(TimeValue ins, TimeValue amount)**

This method is called to insert the specified amount of time at the specified insertion point.

## Drawing and Hit Testing Tracks and Function Curves

This section discusses how an item may display and hit test itself in the track and function curve modes of track view.

To indicate the amount of space it needs to draw itself, the item implements the following method:

### **virtual int GetTrackVSpace(int lineHeight)**

Returns the vertical space occupied by the track in units of one line.

An item may have a custom appearance for itself in the track view. It does this custom drawing in its implementation of the following method:

### **virtual int PaintTrack(HDC hdc, Rect& rcTrack, Rect& rcPaint, float zoom, int scroll, DWORD flags)**

This method is called to display the item in the track view. If an item needs to draw itself in a special fashion, it implements this method to do so. For example, a sound plug-in may draw its waveform using this method. If an item does not need to draw itself, the default implementation may be used. This draws the range bar for the item in the track instead.

Note: When drawing something to appear in Track View, a developer should not do any clipping of their own. 3ds max will take care of all clipping itself.

The next method is used to hit test a track. Hit testing is checking to see which keys have been selected by the user. This selection may be a single pick point or a dragged rectangular region. This method is passed a table of **TrackHitRecords** to update. Each key that lies within the hit rectangle (is hit) should be added to this table.

### **virtual int HitTestTrack(TrackHitTab& hits, Rect& rcHit, Rect& rcTrack, float zoom, int scroll, DWORD flags)**

This method is called to determine which keys lie within the **rcHit** rectangle. Keys that are hit are added to the **hits** table.

When the user has selected Function Curve mode, 3ds max displays the item in a graph with time along the horizontal axis and the value of the item along the vertical axis. An item can draw itself in track view in any way it wishes. It does so by implementing the following method:

### **virtual int PaintFCurves(ParamDimensionBase \*dim, HDC hdc, Rect& rcGraph,**

**Rect& rcPaint, float tzoom,int tscroll,  
float vzoom, int vscroll, DWORD flags)**

This method is called to draw the function curve of the item.

There are a set of macros that are handy for scaling time and values into and out of screen space. These are often used in the implementations of **PaintFCurve()** and **HitTestFCurves()**. See [List of Screen-Time-Value Macros](#).

When the user executes zoom extents on the item, this method is called:

**virtual int GetFCurveExtents(ParamDimensionBase \*dim,  
float &min, float &max, DWORD flags)**

This method is called to calculate the largest and smallest values of the item.

The next method is used to hit test the keys of a function curve. This selection may be a single pick point or a dragged rectangular region. This method is passed a table of **TrackHitRecords** to update. Each key that lies within the hit rectangle (is hit) should be added to this table.

**virtual int HitTestFCurves(ParamDimensionBase \*dim,  
TrackHitTab& hits, Rect& rcHit, Rect& rcGraph,  
float tzoom, int tscroll,  
float vzoom,int vscroll, DWORD flags)**

This method is called to hit test the item's function curves. It is called to determine which keys on the curve lie within the **rcHit** rectangle. Keys that are hit are added to the **hits** table.

## Operations to Keys

This section discusses some of the various Animatable method for dealing with keys in track view. These methods perform operations such as adding, deleting and scaling keys.

### **virtual void AddNewKey(TimeValue t,DWORD flags);**

This method is called to add a new key at the specified time. The value of the key is set to the value of the previous key, or interpolated between keys, based on the flags passed.

### **virtual void DeleteKeys(DWORD flags)**

This method is called to delete keys, as specified by the flags passed.

### **virtual void SelectKeys(TrackHitTab& sel, DWORD flags);**

This method is called to select or deselect a set of keys identified by the **TrackHitTab** and the specified flags.

### **virtual void MoveKeys(ParamDimensionBase \*dim,float delta,DWORD flags);**

This method is called to move selected keys vertically in the function curve editor. This moves the key values but does not alter the key times.

### **virtual void ScaleKeyValues(ParamDimensionBase \*dim, float origin, float scale, DWORD flags)**

This method is called to scale selected keys values. This scales the key values but does not alter the key times.

### **virtual void MapKeys(TimeMap \*map, DWORD flags);**

This method is called to update the keys specified by the flags, using the TimeMap passed. The plug-in should go through the specified keys and change their time to **TimeMap::map(time)**.

## **Track View and References**

3ds max maintains a reference to every sub-anim in the track view. This is how track view monitors if the sub-anim is changing. This reference is created when the sub-anim appears in track view, and is deleted when the sub-anim is no longer shown.

## Summary

This section has provided an overview of many of the methods related to participating in track view. The main methods developers must use to work with keys, time ranges, function curves, and copying/pasting operations were discussed. The primary methods associated with track view are from class **Animatable**. See this class for more detailed information on the methods presented above as well as the others associated with track view.

## UI Customization

See Also: [Class ActionTable](#), [Class ActionCallback](#), [Class ActionContext](#), [Class IActionManager](#), [Class DynamicMenu](#), [Structure ActionDescription](#), [Class DynamicMenuCallback](#), [Class ClassDesc](#), [Class Interface](#).

## **Overview**

This section describes the various classes and functions used to customize the user interface in 3ds max R4. This includes the ability to customize the 3ds max main pulldown menus, the toolbars, quad menus and keyboard shortcuts.

Discussed below is the Action System. This is a new system used for customizing the user interface. Plug-ins may use this to customize the 3ds max main menu, toolbars, quad menus and keyboard shortcuts. This system supercedes the system used by previous version of 3ds max for keyboard accelerators.

The system relies on plug-ins creating Action Tables. These are tables of UI related operations. These operations are called Action Items. When a UI operation is performed the code that carries out that operation is done by code in an Action Callback object.

## Overview of the Principal Classes of the Action System

### [Class ActionTable](#)

The class ActionTable is a generalization of the ShortcutTable class from R3. An ActionTable holds a set of ActionItems, which are operations that can be tied to various UI elements, such as keyboard shortcuts, Custom User Interface buttons, the 3ds max main menu and the Quad menu. MAX's core code exports several ActionTables for built-in operations in MAX. Plug-ins can also export their own action tables via methods available in Class ClassDesc.

### [Class ActionItem](#)

The class ActionItem is used to represent a single operation that lives in an ActionTable. ActionItem is an abstract class with operations to support various UI operations. The system provides a default implementation of this class that works when the table is build with the **ActionTable::BuildActionTable()** method. However, developers may want to specialize this class for more specific-purpose applications. For example, MAXScript does this to export macroScripts to an ActionTable.

### [Structure ActionDescription](#)

This structure is used when creating Action Tables. An array of these structures is created with one element for each action in the table to be built. This array of structures is then passed as a parameter to the ActionTable constructor.

### [Class ActionCallback](#)

This is the callback class that actually execute the action when a user requests it. The **ExecuteAction()** method of this class is called and performs the work. Developers create a sub-class of this class and implement **ExecuteAction()**. This usually consists of a case statement which switches on the ID of the command being executed.

### [Class ActionContext](#)

This class functions like an identifier for a group of actions. Several Action Tables can share a single ActionContext. The class maintains an ID, a name, and an active state for the context and has methods to retrieve these. When a Action Context is active all the Action Tables that use that context are active as well.

### [Class IActionManager](#)

This class provides methods to work with the available Action Tables. There are methods to get the number of and a pointer to specific action tables, activate and

deactivate action tables, work with action contexts and determine if they are active. A pointer which may be used to call the methods of this class is returned from **Interface::GetActionManager()**.

### [Class DynamicMenu](#)

This is a helper class used in putting up additional quad or toolbar menus 'on the fly'. This class has a few methods related to adding new menu items, beginning and ending sub-menus, checking menu flags, and getting a pointer to the menu after it has been created. The dynamic menu created by this class is provided to the method **ActionItem::GetDynamicMenu()**.

### [Class DynamicMenuCallback](#)

This is the callback used when creating dynamic menus. An instance of this class is passed to the constructor of the helper class DynamicMenu. This class has a single method called **MenuItemSelected()** which is called when the user has chosen an item from the menu.

### [Class MAXIcon](#)

### [Class ClassDesc](#)

There are two new method of this class related to ActionTables. When a plug-in wants to make its keyboard accelerators available it uses the following new methods:

**virtual int NumActionTables();**

**virtual ActionTable\* GetActionTable(int i);**

### [Class Interface](#)

There is a new method to access the Action Manager interface. This is:

**virtual IActionManager\* GetActionManager() = 0;**

See [Class IActionManager](#) for details.

## **Building Action Tables**

This section discusses the approach developers may use to build action tables and make them available to the system. In most cases building an ActionTable is fairly easy. It is a bit more work if you choose to implement your own custom sub-class of ActionItem, but in most cases that isn't needed (see next section). The system provides a helper class called ActionDescription that helps in building tables.

```
struct ActionDescription {  
// A unique identifier for the command (must be unique per table)  
int mCmdID;  
// A string resource id that describes the command  
int mDescriptionResourceID;  
// A string resource ID for a short name for the action  
int mShortNameResourceID;  
// A string resource for the category of an operation  
int mCategoryResourceID;  
};
```

An ActionTable is built by making a static table of action descriptions and passing it to the constructor for ActionTable. For example, here is the code that builds the action table for the FFD modifier:

```
#define NumElements(array) (sizeof(array) / sizeof(array[0]))  
static ActionDescription spActions[] = {  
ID_SUBOBJ_TOP,  
IDS_SWITCH_TOP,  
IDS_SWITCH_TOP,  
IDS_RB_FFDGEN,  
ID_SUBOBJ_CP,  
IDS_SWITCH_CP,  
IDS_SWITCH_CP,  
IDS_RB_FFDGEN,  
ID_SUBOBJ_LATTICE,  
IDS_SWITCH_LATTICE,  
IDS_SWITCH_LATTICE,  
IDS_RB_FFDGEN,
```

```

ID_SUBOBJ_SETVOLUME,
IDS_SWITCH_SETVOLUME,
IDS_SWITCH_SETVOLUME,
IDS_RB_FFDGEN,
};
ActionTable* BuildActionTable()
{
TSTR name = GetString(IDS_RB_FFDGEN);
HACCEL hAccel = LoadAccelerators(hInstance,
MAKEINTRESOURCE(IDR_FFD_SHORTCUTS));
int numOps = NumElements(spActions);
ActionTable* pTab;
pTab = new ActionTable(kFFDActions, kFFDContext, name, hAccel,
numOps, spActions, hInstance);
GetCOREInterface()->GetActionManager()-
>RegisterActionContext (kFFDContext, name.data());
return pTab;
}

```

The constructor for ActionTable takes the ID of the table, the context id, a name for the table, a windows accelerator table that gives default keyboard assignments for the operations, the number of items, the table of operation descriptions, and the instance of the module where the string resources in the table are stored.

At the same time the table is built, you also need to register the action context ID with the system. This is done with the

**IActionManager::RegisterActionContext()** method.

The other part of implementing an ActionTable is implementing an ActionCallback class. This is an abstract class with a virtual method called **ExecuteAction(int id)**. You need to sub-class this and pass an instance of it to the system when you activate the ActionTable. Then when the system wants to execute an action, you will get a callback to **ActionCallback::ExecuteAction()**.

For the FFD modifier, this looks like:

```

template <class T>

```

```

class FFDActionCB : public ActionCallback
{
public:
T* ffd;
FFDActionCB(T *ffd) { this->ffd = ffd; }
BOOL ExecuteAction(int id);
};
template <class T>
BOOL FFDActionCB<T>::ExecuteAction(int id) {
switch (id) {
case ID_SUBOBJ_TOP:
ffd->ip->SetSubObjectLevel(SEL_OBJECT);
ffd->ip->RedrawViews(ffd->ip->GetTime());
return TRUE;
case ID_SUBOBJ_CP:
ffd->ip->SetSubObjectLevel(SEL_POINTS);
return TRUE;
case ID_SUBOBJ_LATTICE:
ffd->ip->SetSubObjectLevel(SEL_LATTICE);
return TRUE;
case ID_SUBOBJ_SETVOLUME:
ffd->ip->SetSubObjectLevel(SEL_SETVOLUME);
return TRUE;
}
return FALSE;
}

```

FFD uses a template class to implement several versions of this callback, but this is not required.

Finally, the system needs to activate and deactivate the table at the appropriate time. When to do this depends on the scope of applicability of the table. If your ActionTable is exported from an editable object of modifier plug-in, then you typically want it only to be active when editing the object or modifier. This is done by activating it in the BeginEditParams() method, and deactivating it in EndEditParams().

For FFD, this looks like this:

```
ffdActionCB = new FFDActionCB<FFDMod >(this);  
ip->GetActionManager()->ActivateActionTable(ffdActionCB,  
kFFDActions);
```

The first parameter is the ID of the table to activate, and the second is an instance of the ActionCallback class that is responsible for executing actions. In EndEditParams(), we deactivate the table:

```
ip->GetActionManager()->DeactivateActionTable(ffdActionCB,  
kFFDActions);  
delete ffdActionCB;
```

For other types of plug-ins the table can be activated at different times. For example, you could write a GUP plug-in that activates the table when the plug-in is loaded to provide actions that are always available.

### **To Sub-Class or not to Sub-Class?**

Developers have the option of sub-classing both the ActionTable class and the ActionItem class, but are not required to do either. Only the ActionCallback is required to be sub-classed in all cases.

For the ActionItem class, developers only need to sub-class in rare cases. The default implementation of this class stores internal strings for its name, description, category, icon, etc. In some cases this might lead to duplicate storage if that information is stored elsewhere. In that case, you might want to provided a specialization of this class for memory efficiency.

The ActionTable class also has several virtual methods that the developer might want to implement. All these methods have default implementations, so sub-classing is not required unless special behavior is needed.

The default handlers for some of the handlers may not be appropriate. For example IsEnabled(int cmdId) returns TRUE in every case in the default implementation. If you want command to be disabled under some conditions, then you will need to build a specialization of ActionTable and implement this method. Other methods that you might want to implement are:

```
virtual BOOL GetButtonText(int cmdId, TSTR& buttonText);  
virtual BOOL GetMenuText(int cmdId, TSTR& menuText)  
virtual BOOL GetDescriptionText(int cmdId, TSTR& descText);  
virtual BOOL IsChecked(int cmdId);
```

```
virtual BOOL IsItemVisible(int cmdId);  
virtual BOOL IsEnabled(int cmdId);  
virtual void WritePersistentActionId(int cmdId, TSTR& idString);  
virtual int ReadPersistentActionId(TSTR& idString);
```

You only need to implement the last two methods if your table uses command identifiers that are not persistent from session to session. An example of this is the ActionTable that exports macroScripts from MAXScript. This system reads all the macroScripts in your system and assigns command IDs that are based on the order the macroScripts are read from the file. This might change from session to session, so when we want a persistent ID to write out to our keyboard or CUI files, we need one that doesn't change. For macroScripts, it writes the name of the script concatenated with the category of the script. The ReadPersistentID method takes that string and returns its integer command ID for the current session.

For tables that use constant integer identifiers that don't change from session to session, like the FDD example above, there is no need to implement this method.

### **Registering Action Tables**

In order for the system to use an action table, it need to be registered. For most plug-ins, this is done by returning it's action table in the following methods in ClassDesc:

```
virtual int NumActionTables();  
virtual ActionTable* GetActionTable(int i);
```

The system will call these methods on start-up, so if your plug-in exports action tables, it cannot be demand loaded. This is required because the ActionItems need to be displayed in the customization dialogs even if your plug-in is not in use.

### **Action Context IDs used by MAX**

The following action context IDs are used internally in MAX.

```
const ActionContextId kActionMainUIContext = 0;  
const ActionContextId kActionTrackViewContext = 1;  
const ActionContextId kActionMaterialEditorContext = 2;  
const ActionContextId kActionVideoPostContext = 3;  
const ActionContextId kActionSchematicViewContext = 5;  
const ActionContextId kActionIReshadeContext = 6;
```

## **Menu Manager**

The menu manager API lets plug-ins register menus and menu contexts that are saved in MAX's menus customization file, and can be configured by the user.

Menus are populated with ActionItems published from ActionTables. Menu contexts are places where menus can appear in MAX's UI. Menu contexts can be either for the main menu bar, or for places where a Quad menu can be displayed. 3ds max ships with a few pre-defined menu contexts, including the main menu bar, the viewport Quad menu and the ActiveShade Quad menu. Only a single menu-bar context is allowed, but plug-ins may register new Quad menu context as appropriate. For example, the UVW Unwrap plug-in in the 3ds max SDK defines a Quad menu context that is used when the user right-clicks in the UVW Unwrap window.

Menu creation is handled with the IMenu API defined in the iMenus.h header in the 3ds max SDK.

### **Adding to MAX's default menus**

A plug-in can use a MenuContext to register new menu items on to MAX's default menu bar or Quad menus. Since 3ds max gets its menu configuration from a file (in UIMaxMenus.mnu by default), plug-ins should only register their extensions a single time. To determine if the menu extensions have been registered yet, the plug-in should register a menu context with the ImenuManager::RegisterMenuContext() method. This method will return false if the context has not been registered, and true if it has. After the context is registered, it is saved in the menu file, and the next time 3ds max starts, the call to RegisterMenuContext() will return false.

To add items to MAX's main menu, the plug-in should check the return value of RegisterMenuContext(), and if it is true, that means that this is the first time it has been registered, and the plug-in can then create new menus, add items to MAX's main menu and Quad menus. In this case the MenuContext is used only as a place holder for determining when to add items to menus, not as a place where menus can appear.

Plug-ins can also register Quad menu contexts to be used as places Quad menus can appear in their UI. Typically this only applies if the plug-in creates its own floating window, such as UVW Unwrap. Normally, plug-in just need to register ActionItems, and possibly add items to MAX's main menu or default menus. Once items have been added to menus, and new menus have been registered,

they will appear in MAX's menu customization dialogs. Users can then move the items around, or remove them from menus.

If a plug-in wants to add menus and items to MAX's existing menus, or create its own Quad menu context, it should be done when 3ds max starts. This is best done with a GUP-style plug-in. These plug-ins have a Start() method that is called when MAX's first starts up. It is in that method that plug-ins should register new contexts and add menus and items.

### **Menu Context Ids**

Every menu context needs a unique 32-bit integer identifier. The ids for MAX's built-in contexts are defined in the iMenuMan.h header file. New plug-ins should use a random, fixed, integer value for the context. This can be generated using the "gencid.exe" program in MAX's SDK. Just use one of the 2 32-bit values generated.

### **Class IMenuBarContext**

3ds max comes with one pre-defined MenuBarContext, which is used to obtain the menu used on MAX's main menu bar. Its context id is

**kMenuContextMenuBar**. When a plug-in wants to add items or sub-menus to the main menu bar, it should get the menu from the main menu bar context as follows:

```
IMenuBarContext* pMenuBarContext = (IMenuBarContext*)  
GetCOREInterface()->GetMenuManager-  
>GetContext(kMenuContextMenuBar);  
IMenu* pMenu = pMenuBarContext->GetMenu();
```

The program can then use the IMenu class methods to add items or sub-menus to the main menu. It is recommended that plug-ins add their own top-level menu rather than adding items to existing menus.

The other use of the IMenuBar context is mentioned above. Plug-ins can register a menu bar context with its own ID to determine whether it should extend the main menu or not.

## **Sample Code**

Examples of Action Table APIs in use are found in many places in the SDK. A simple example is `\MAXSDK\MODIFIERS\FFD`.

## Units of Measurement

See Also: [Units of Measurement Reference](#).

This section discusses how developers work with the units of measurement used by 3ds max.

3ds max allows users to work with units that they are most comfortable with. Users may specify that they wish to work in US Standard units (Inches, Feet, Miles) or Metric units (Millimeters, Centimeters, Meters or Kilometers). Or they may specify a custom setting (for example perhaps they want to work in fathoms). Users may also work with generic units (these are the same as the default 3ds max master units).

Internally 3ds max keeps track of all measurements in its own internal unit called the master unit. The program converts measurements from user-defined units into the master defined unit for storage and computation. The master unit is defined as 1.0 inch. The user is able to change this master unit using Customize/Preferences, selecting the General page, and adjusting the System Unit Scale controls.

To retrieve the current unit display information one can use

**GetUnitDisplayInfo()**. This returns the information describing MAX's current system of measurement. This includes the type of units used, how they are displayed and custom unit name and scale. A developer may also use

**SetUnitDisplayInfo()**. This allows the developer to control the unit settings.

Two functions are available for converting units in and out of ASCII form. These are **FormatUniverseValue()** and **DecodeUniverseValue()**.

**FormatUniverseValue()** converts the specified value to its ASCII representation using the current unit display settings. **DecodeUniverseValue()** parses the specified measurement string and converts it to a floating point value.

Developers can use the method **GetMasterScale(int type)** to retrieve the master scale in terms of the specified unit type. For example:

**GetMasterScale(UNITS\_INCHES)** returns the number of inches per unit (unless altered by the user this would be 1.0).

To review the methods discussed above in detail see the section [Units of Measurement Reference](#).

## Undo/Redo

See Also: [Class RestoreObj](#), [Class Hold](#).

## Overview

3ds max has a built-in system for handling undo and redo. This allows users to undo their previous modifications to the database. The user may undo several operations, and also redo several undo operations.

The undo/redo system uses a global object call **theHold**. This is an instance of the class **Hold**. The developer calls methods of **theHold** to participate in undo/redo. Another class involved is **RestoreObj**. The developer creates instances of a class derived from **RestoreObj** to save the data needed to undo and redo the operation of the plug-in and to implement several methods required by MAX. The process works as follows:

Any operation that will modify the database checks to see if **theHold** is 'holding'. This means the 3ds max undo system has had the **Begin()** method called. If **theHold** is not 'holding' the developer should call **theHold.Begin()**. This signals the start of a potential undo operation. If **theHold** is holding, any operation that modifies the database must register a restore object with the system before it modifies the database. A restore object is an instance of the class **RestoreObj**. The restore object saves just enough data to restore the database to the state it was in when **theHold.Begin()** was called. It also saves data to allow a Redo operation to occur.

As a simple example consider a utility plug-in that allows the user to change the wireframe color of a node in the scene. The restore object would need to save the previous color and a pointer to the node being changed. It would also need to have storage to save the current color prior to an undo. This would allow it to undo the undo, or redo the operation.

In order to register the restore object with the system the developer calls **theHold.Put()**. This methods passes a pointer to the restore object. For example:

```
if ( theHold.Holding() ) {  
    theHold.Put(new ColorRestoreObj(currentColor,this));  
}
```

In the example above the **ColorRestoreObj** is a developer defined class derived from **RestoreObj**.

Once the restore object has been registered with **theHold** there are two potential cases to terminate the **Begin()**. The user can complete the operation or they may

cancel it.

If the user completes the operation, the developer calls **theHold.Accept()**. This registers an undo object with the undo system and leaves the database in its modified state.

If the user cancels the operation, the developer calls **theHold.Cancel()**. This restores the database to its previous state and throws out the restore object.

In addition to saving the data it needs to restore the database, the restore object has three main methods it must implement:

### **Restore(int isUndo)**

This is called to restore the database to the state it was in when **theHold.Begin()** was called. In the earlier example, the **Restore()** method would reset the wireframe color to the previous color. A flag, **isUndo**, is passed into this method to indicate if it was called in response to the Undo command. In the above example, if the flag is TRUE the developer must save the current color before restoring the previous color so if the user selects the Redo command the color could be reset to the state before the undo.

### **Redo()**

This is called by the system if the user has selected the Redo command. This means the user wants to undo the undo. The developer restores the database to the state it was in before the last undo was called on the restore object.

### **Size()**

This is called by the system to retrieve the size in bytes of the restore object. This is used to make sure all the accumulated restore objects do not grow beyond a manageable size.

Sometimes the **Restore()** method of the restore object may be called but not in response to the user selecting the Undo command. The undo system also allows values to be held in a buffer and then restored programmatically. For example, consider the operation of moving a node in the scene using the mouse. When the user first clicks the mouse button down to begin the operation, the state of the node's [= 4\) BSPSPopupOnMouseOver\(event\);">transform controller](#) is saved as part of a restore object. As the user moves the mouse, the plug-in tracks its position and updates the position of the node in the scene. Over and over the plug-in receives a message that the mouse has moved and the node in the scene must be re-positioned. Rather than storing incremental moves from the previously calculated position (which may accumulate error) it is more accurate

to restore the mouse position to where it started and recalculate the transformation based on the distance from the original mouse point to the new mouse point. In order to reset the original position, the system calls **Restore()**. This puts the node back into its original position. Then when the new position is calculated this is applied to the node. This happens over and over until the user releases the mouse. When the mouse is released **theHold.Accept()** is called to register an undo object with the system.

In iterative operations such as this it is often useful to set one of the flags of Animatable to indicate that a restore object is being held. In the example above, when the user first clicks down on the mouse the developer checks if **theHold** is holding and if it is calls **theHold.Put()** to register a restore object. Then the developer calls a method of Animatable **SetAFlag(A\_HELD)**. This sets the **A\_HELD** bit of the Animatable **aflag** data member to indicate the restore object is held. Then on each iteration the bit is tested to see if it is set and if so another restore object is not registered. A single restore object can be restored over and over again.

When **theHold.Accept()** or **theHold.Cancel()** is called, the system calls a method of the restore object called **EndHold()**. The developer may then clear the **A\_HELD** bit to indicate the restore object is no longer being held.

## Database Changes that are not Undoable

When 3ds max is exited, reset, etc., the save requester is only brought up if there is something on the undo stack. If a plug-in makes a change that is not undoable, the 'save dirty bit' must be set. This will indicate to the system that the save requester needs to be brought up. There are three APIs related to this:

### Function:

**BOOL GetSaveRequiredFlag();**

### Remarks:

Implemented by the System.

Returns TRUE if the 'save required' flag is set; otherwise FALSE. Note: this method does not tell you if saving is required, it just returns the value of the 'save required' flag. To really know if saving is required, you have to check the undo buffer as well as this flag. This is the purpose **IsSaveRequired()** below.

### Function:

**void SetSaveRequiredFlag(BOOL b=TRUE);**

### Remarks:

Implemented by the System.

Sets the 'save required flag'. Note that calling

**SetSaveRequiredFlag(TRUE)** will cause the 'Save Changes' prompt to appear, but **SetSaveRequiredFlag(FALSE)** will not prevent it from coming up unless you also reset the undo buffer.

### Parameters:

**BOOL b=TRUE**

TRUE to set the save dirty bit; FALSE to clear it.

### Function:

**BOOL IsSaveRequired();**

### Remarks:

Implemented by the System.

Returns TRUE if a change was made to the 3ds max database that would require the file to be saved. In other words, it returns TRUE if the save requester will be brought up when the user exits, resets, etc.; otherwise

FALSE.

## Flushing the Undo Buffer

If a plug-in developer needs to clear the undo buffer, two APIs are available to do so. A method of class **Interface** will flush the undo buffer:

**void FlushUndoBuffer();**

Developers may also use:

**int GetSystemSetting(int id);**

If the ID **SYSSET\_CLEAR\_UNDO** is passed to this method, the undo buffer is flushed. This function will return 0. Note that this will only work with version 1.1 of 3ds max or later.

An example of when to do this is when a creation object is deleted after some undo items have been put on the stack that refer to that object, for instance when a plug-in is doing a custom creation process. See the sample code in **\MAXSDK\SAMPLES\OBJECTS\LIGHT.CPP**.

If the creation object is deleted without being attached to a node, or if the creation is canceled, and if some undo objects have been logged since the creation object was created, then one should flush the entire undo stack. This brute force way of doing it is, unfortunately, the only safe way to ensure that the 3ds max undo stack is not corrupt with restore objects that point to deleted objects.

## Summary

In summary, plug-In developers are encouraged to have their plug-ins fully participate in the undo redo system of MAX. The developer works with two classes. **RestoreObj** is used to store data for undoing and redoing modifications to the database. **Hold** is used to call methods of the system and the restore object to manage undo and redo.

For reference information see [Class Hold](#) and [Class RestoreObj](#).

## Working with Action Tables

See Also: [Class ActionTable](#), [Class ActionItem](#), [Structure ActionDescription](#), [Class ActionCallback](#), [Class ActionContext](#), [Class IActionManager](#).

## Working with Bitmaps

See Also: [BitmapManager](#), [BitmapInfo](#), [Bitmap](#), [BitmapIO](#), [BitmapStorage](#).

## Overview

This section presents information about working with bitmap images. It discusses the main classes used, and presents information about concepts such as creating, loading and saving bitmaps, memory management when working with bitmaps, palettes, alpha, gamma, the geometry/graphics buffer (G-buffer), working with multi-frame images, handling errors that occur, and pixel storage formats. This section also documents a few utility functions in the API that are not part of any class.

## Overview of the Principal Classes

The following three classes are the main ones used when working with bitmaps:

### [BitmapManager](#)

There is a global object defined by 3ds max called **TheManager**. This is an instance of the class **BitmapManager**. This object manages and enables developers to work with bitmaps in MAX. For example, this class provides methods for creating and loading bitmaps. It also has methods for displaying some common dialogs that let users interactively specify files and devices to work with, and set options for bitmaps.

### [BitmapInfo](#)

**BitmapInfo** is the class used to describe the properties of a bitmap. The developer can declare an instance of this class and use its methods such as **SetWidth()**, **SetHeight()**, **SetGamma()**, and **SetName()** to describe the bitmap properties. The other classes related to bitmaps then use this information. Thus **BitmapInfo** is the heart of all image input/output. For example, all but a few methods in the **BitmapManager** use a **BitmapInfo** object as the main argument. For instance, if you wish to create a bitmap, you declare an instance of **BitmapInfo** and use its methods to establish the bitmap properties. Then when you call the **BitmapManager** method **Create()**, you pass the **BitmapInfo** object. The **BitmapManager** uses this information to determine how much memory to allocate based on the width, height and color depth.

This class also has methods to get and set the number of frames used in multi-frame bitmaps, and to define 'custom' properties of the bitmap. Custom properties let you specify only a portion of the main bitmap, such as a smaller, sub-region of the original, or fewer frames than the original (different begin and end settings or a different frame increment step size for multi-frame bitmaps). These custom properties are read and used by bitmap copying operations for example.

### [Bitmap](#)

The **Bitmap** class is the bitmap itself. All image access is done through this class. This class provides standard methods to retrieve and store pixels from the image. The **Bitmap** class has methods to retrieve parameters of the bitmap such as its width, height, whether it is dithered, or has an alpha

channel. Additional methods allow developers to open bitmaps for output, write multi-frame images, and to copy pixel data between bitmaps.

There are other bitmap related classes used only by developers who create 3ds max plug-ins used to load and save new bitmap file formats. The methods of these classes are called by the **BitmapManager** and **Bitmap** classes and not by developers themselves. The primary classes used for creating image loading and saving plug-ins are:

### [BitmapIO](#)

This is the main plug-in class used by developers creating image loader / saver plug-ins. For example, a developer creating a plug-in to support the PCX file format would derive their plug-in class from **BitmapIO**. This class is used for both files and devices (devices are items such as digital disk recorders). Developers implement pure virtual methods of this class to provide information about the image loader / saver they are creating. This is information such as the plug-in author name, copyright data, image format description, filename extension(s) used, and other capabilities of the image loader / saver. The developer also implements methods to load the image from disk, prepare it for output, write image data to it, and close it. Image loader / saver plug-ins use the **BitmapStorage** class described below to load and save their actual pixel data.

### [BitmapStorage](#)

When an image is loaded or created, the buffer that will hold the image data is an instance of the **BitmapStorage** class. This class allows developers to access the image in a uniform manner even though the underlying storage might be 1, 8, 16, 32 or 48 bit. For example, a paletted 8-bit format is perfect for loading GIF files but not for loading 32-bit Targa files. The inverse is also true. There is no point in creating a true color 64-bit storage to load a GIF file that only has 8-bit color information.

Thus the **BitmapStorage** mechanism was created so images could be kept in memory in a more efficient way. Instead of loading everything in MAX's internal 64-bit format, bitmaps are loaded and/or created using the most efficient storage for their type. The **BitmapStorage** class provides an uniform set of pixel access methods to hide these different formats from the developer using bitmaps. For example, standard methods are available for getting and putting pixels at various color depths. In this way, even though an image may be a 1-bit monochrome line art bitmap, pixels may be retrieved or

stored at other color depths. For example the **BitmapStorage** methods **Get16Gray()/Put16Gray()**, **GetTruePixels()/PutTruePixels()**, and **GetIndexPixels()/PutIndexPixels()** provide access to pixel data at various color depths.

Again, note that the **BitmapStorage** class is a low level access mechanism used by image loader / saver (**BitmapIO**) plug-ins. Developers wanting to access an image use the methods in the **Bitmap** class instead.

## Needed #include and LIB files

When working with bitmaps, make sure you add the bitmap include file to your source file, i.e. use the statement:

```
#include "bmmlib.h"
```

Also be sure to set your MS VC++ project settings to link to  
**\MAXSDK\LIB\BMM.LIB**

Using **bmmlib.h** and **bmm.lib** will ensure you have access to all the bitmap classes, methods, and functions.

## Creating, Opening, Writing, and Closing Bitmaps

This section shows how the **BitmapInfo** and **BitmapManager** classes may be used to create, open, and write to bitmaps. Following this is code that demonstrates how to open an existing bitmap using the standard 3ds max Open File dialog box. Both these examples show how to properly delete the bitmaps after they are used.

To create a bitmap from scratch, we need to declare a pointer to point to it:

```
Bitmap *bmap;
```

Remember that the **Bitmap** class represents the bitmap itself. Next, we need to declare an instance of the **BitmapInfo** class to describe the properties of the bitmap to create:

```
BitmapInfo bi;
```

Then we initialize the **BitmapInfo** with the properties of the bitmap we wish to create. This is done using the methods of **BitmapInfo** such as **SetType()**, **SetWidth()**, etc.:

```
// Initialize the BitmapInfo instance  
bi.SetType(BMM_TRUE_64);  
bi.SetWidth(320);  
bi.SetHeight(200);  
bi.SetFlags(MAP_HAS_ALPHA);  
bi.SetCustomFlag(0);
```

Note in the above code how the type is set. The type describes the number of bits per pixel used to describe the image, whether the image is paletted (has a color map), and if it has an alpha channel. To see a list of the types of bitmaps that may be created see [List of Bitmap Types](#). In this example, we use **BMM\_TRUE\_64**. This format has 16-bits for each color (RGB) and alpha component. This is the format that 3ds max uses internally in the renderer.

Once the **BitmapInfo** is initialized, we can call a method of the bitmap manager to create the bitmap. A global instance of the **BitmapManager** class exists called **TheManager**. This is what we use to call methods of **BitmapManager** as shown below:

```
// Create a new bitmap  
bmap = TheManager->Create(&bi);
```

Note that we pass it our **BitmapInfo** instance. This is where the bitmap manager gets the information on the bitmap to create. If the pointer returned is NULL, an error has occurred in creating the bitmap. If the pointer is non-NULL, it's valid and we can use it to call methods of the **Bitmap** class to work with the bitmap. The code below shows how the **PutPixels()** method is used to write data to the bitmap. This code sets every pixel of the image to the color and opacity specified by **r**, **g**, **b** and **alpha**.

```
...
BMM_Color_64 *line, *lp, color = {r, g, b, alpha};
int bmapWidth = bmap->Width(), bmapHeight = bmap->Height();

if ((line = (BMM_Color_64 *)calloc(bmapWidth,
sizeof(BMM_Color_64)))) {
    int ix, iy;
    for(ix = 0, lp = line; ix < bmapWidth; ix++, lp++)
        *lp = color;
    for(iy = 0; iy < bmapHeight; iy++)
        int res = bmap->PutPixels(0, iy, bmapWidth, line);
    free(line);
}
```

When we are done using the bitmap, we need to delete it. This is done by simply using the delete operator:

```
if (bmap) bmap->DeleteThis();
```

That's all that's required to create a bitmap from scratch.

The following code demonstrates two other methods of the bitmap manager -- **SelectFileInput()** to allow the user to choose a file to load (and initialize the **BitmapInfo** instance passed to it) and **Load()** to create a bitmap to hold the image. Again, we declare a **Bitmap** pointer and a **BitmapInfo** instance.

```
Bitmap *bmap;
BitmapInfo bi;
```

Next, we use a method of the bitmap manager to allow the user to choose an image file. This method returns FALSE if the user cancels and TRUE if they select an image. If you want to directly indicate which file to load, just set the fields of **BitmapInfo** yourself (for example

```

bi.SetName(_T("TEST.JPG"));
// Let the user choose the file to open
BOOL res = TheManager->SelectFileInput(&bi, ip-
>GetMAXHwnd(),
  _T("Open File"));
if (!res) return; // User cancelled...

```

After **SelectFileInput()** returns, the **BitmapInfo** instance passed to it contains the necessary information about the bitmap to load. To load the bitmap, the **Load()** method of the bitmap manager is used.

```

// Load the selected image
BMMRES status;
bmap = TheManager->Load(&bi, &status);
if (status != BMMRES_SUCCESS) {
  MessageBox(ip->GetMAXHwnd(), _T("Error loading bitmap."),
  _T("Error"), MB_ICONSTOP);
}

```

Once the image is loaded, you can work with it like any other bitmap. For example to display the bitmap in a window, use the **Bitmap::Display()** method.

```

// Display the opened bitmap
bmap->Display(title, BMM_CN, FALSE, TRUE);

```

A few more notes on the bitmap manager **Load()** method -- Additional options may be set by calling **BitmapManager::ImageInputOptions()** before calling **Load()**. This method will ask the user for special details such as custom positioning of smaller/larger images, etc. This method sets the proper fields in **BitmapInfo**.

The examples above show how to create and load a bitmap. Once you have the bitmap what if you want to save it to disk? Also, how do you deal with multi-frame files (like a FLC or AVI)? The example below demonstrates both these things for a multi-frame file. Again, we declare a **Bitmap** pointer and a **BitmapInfo** instance.

```

Bitmap *bmap;
BitmapInfo bi;

```

To allow the user to choose an output file type, use the **BitmapManager** method **SelectFileOutput()**. This brings up the 'Browse Images for Output' dialog box. This method returns TRUE if the user selected a file; it returns FALSE if they cancel the dialog box. This method also handles checking if the filename chosen already exists, and if so provides an overwrite question dialog. This method will fill in the proper fields of the **BitmapInfo** passed.

```
BOOL gotIt = TheManager->SelectFileOutput(&bi, ip-  
>GetMAXHwnd());  
if (!gotIt) return; // User cancelled...
```

Next we need to set the image size and the properties of **BitmapInfo** related to multi-frame images. Below we do this for a 30 frame sequence.

```
bi.SetWidth(640)  
bi.SetHeight(480)  
bi.SetFirstFrame(0)  
bi.SetLastFrame(29)
```

With the **BitmapInfo** setup, we can call the **BitmapManager** to create the sequence:

```
bmap = TheManager->Create(&bi);
```

The next code is used to open the bitmap for output. This indicates to the system that the bitmap is open for output and we can write to.

```
bmap->OpenOutput(&bi);
```

Next we simply write the images for each frame:

```
for (frame = 0; frame < 30; frame++) {  
    // Do something to the image  
    // ...  
    // Write the image  
    bmap->Write(&bi, frame);  
}
```

When we are done we need to close the image:

```
bmap->Close(&bi)
```

Note: You can add any number of outputs to a bitmap. Just keep calling **bmap->OpenOutput()** with different outputs (for instance a TGA file and Frame Buffer). To write or close a specific output, use **Write()** and **Close()**. To write

and close them all at once, use the **Bitmap** methods **WriteAll()** and **CloseAll()**. It is okay to use **WriteAll()** and **CloseAll()** if you have just one output defined.

## Multi-frame Files

Certain file types (and most devices) are multi-frame. **AVI**, **IFL**, **FLI**, and **FLC** are all formats that support multiple frames. The **Bitmap** class has methods for dealing with these multi-frame bitmaps. For example, you can set the current frame to load or save. Also, the **BitmapInfo** class has methods that will let you work with only a subset of frames of a multi-frame image, for example, **SetStartFrame()**, **SetEndFrame()**, and **SetCustomStep()** let you specify a different start, end and frame increment to be used.

When loading files, **BitmapInfo** defaults to frame 0. For multi-frame files you should specify the frame number you want to load. This is done by using **bi.SetCurrentFrame(f)** before calling **Load()**.

## High Dynamic Range Bitmaps

Newly added to the bitmap system in R4 are High Dynamic Range bitmaps. This was accomplished by adding methods to get and put floating point color values into the Bitmap and BitmapStorage classes. Conversions between floating point and fixed point representations are handled by the BitmapStorage, include clamping and scaling of floating point values.

There are four high dynamic range BitmapStorage formats:

- LogLUV32: This format uses a logarithmic encoding of luminance and U' and V' in the CIE perceptively uniform space. It spans 38 orders of magnitude from  $5.43571 \cdot 10^{-20}$  to  $1.84467 \cdot 10^{19}$  in steps of about 0.3% luminance steps. It includes both positive and negative colors. A separate 16 bit channel is kept for alpha values.
- LogLUV24: This format is similar to LogLUV32 except it uses smaller values to give a span of 5 order of magnitude from 1/4096 to 16 in 1.1% luminance steps. A separate 8 bit channel is kept for alpha values.
- LogLUV24A: This format is identical to LogLUV24, except the 8 bit alpha value is kept with the 24 bit color value in a single 32 bit word.
- RealPixel: This format encodes the exponent, e, of the largest rgb component of the pixel, and the ratio of each component with  $2^e$ .

[Structure BMM\\_Color\\_fl](#)

Added BMM\_Color\_fl for floating point access to bitmaps.

[Structure LogLUV32Pixel](#)

[Structure LogLUV24Pixel](#)

[Class BitmapStorage](#)

There are new methods in Bitmap and BitmapStorage that allow data to be retrieved in floating point values rather than integer values so 3ds max can get the high dynamic range data without clamping.

**BMM\_Color\_fl** uses floats to hold the RGBA color components, rather than 16 bit integers which are used in **BMM\_Color\_64**. To convert from **BMM\_Color\_64** to **BMM\_Color\_fl** you divide each component by 65535.0 and to go back you multiply each component by 65535.0. High Dynamic Range bitmaps are not restricted to the range 0.0 to 1.0. Some formats allow negative values and all of the formats can go well above 1.0.

There is a problem is when converting from **BMM\_Color\_fl** to

**BMM\_Color\_64**, since the value may exceed 65535. The bitmap will either clamp the values to 0 to 65535, or scale the values by the largest component value so all of the components are in the range 0 to 65535.

[Class BitmapStorageLDR](#)

[Class BitmapStorageHDR](#)

The class hierarchy of BitmapStorage was changed a little. Two new classes BitmapStorageLDR and BitmapStorageHDR are derived from BitmapStorage and should be used as the base class for BitmapStorage implementations. BitmapStorageLDR provides default implementations of the new floating point BitmapStorage methods and BitmapStorageHDR provides default implementations of the 64 bit pixel BitmapStorage methods.

[Class AColor](#)

Implemented converters between AColor and BMM\_Color\_fl.

[Class Color](#)

Implemented converter between Color and high dynamic range pixel formats.

[Class Texmap](#)

Added method IsHighDynamicRange to Texmap so we can determine when a texmap is returning high dynamic range data.

PreShade and PostShade store high dynamic range data when a Texmap indicates that it returns high dynamic range data.

[Class StdCubic](#)

[Class StdMirror](#)

Added interface method UseHighDynamicRange to allow StdMirror and StdCubic to use high dynamic range bitmaps.

[Structure BMM\\_Color\\_24](#)

[Structure BMM\\_Color\\_32](#)

[Structure BMM\\_Color\\_48](#)

[Structure BMM\\_Color\\_64](#)

[Structure BMM\\_Color\\_fl](#)

[Structure RealPixel](#).

## Custom Bitmap Properties

You may occasionally want to work with only a portion of a bitmap or series of images. For example, when copying one file to another, if you had a 640x480, 30 frame FLC file, you might want to work with the lower right corner 160x100 portion, using every other frame, perhaps starting at frame 5 and ending at frame 15. The 'Custom' methods of the **BitmapInfo** class let you specify these options. Methods such as **SetCustomWidth()**, **SetCustomX()**, and **SetCustomStep()** let you specify the part of the source image that should be manipulated. The method that copied the image would see that these custom properties were set and would act accordingly. See the [BitmapInfo](#) class for details on these methods.

You may use the **BitmapManager::ImageInputOptions()** method to allow the user to specify these options via the standard 3ds max Input Image Options dialog. This dialog simply sets the appropriate data members in **BitmapInfo** based on the user's choices.

## Memory Management for Plug-Ins that work with Bitmaps

Memory is allocated and de-allocated by the bitmap classes in various ways. It is important to understand when memory needs to be freed by the developer, and when the system will take care of freeing it. This section discusses this issue.

The bitmap manager methods **Create()** and **Load()** both return pointers to bitmaps that need to be freed when the developer is done working with them. This is accomplished by simply using the delete operator on the pointer returned from the methods. The pseudo code below shows both of these cases.

This is the **BitmapManager::Create()** case:

```
Bitmap *bmap;  
BitmapInfo bi;  
bmap = TheManager->Create(&bi);  
// Work with the bitmap...  
// ...  
// Free the bitmap when done  
if (bmap) bmap->DeleteThis();
```

This is the **BitmapManager::Load()** case:

```
BMMRES status;  
Bitmap *bmap;  
BitmapInfo bi;  
BOOL res = TheManager->SelectFileInput(&bi, ip-  
>GetMAXHWnd(),  
_T("Open File"));  
bmap = TheManager->Load(&bi, &status);  
// Work with the bitmap...  
// ...  
// Free the bitmap when done  
if (bmap) bmap->DeleteThis();
```

As the code above shows, the developer is responsible for freeing memory from both these methods.

Another example of getting a pointer to a bitmap and needing to free it is the pointer returned from the **Bitmap** method **ToDIB()**. This method creates a new Windows Device Independent Bitmap (DIB) and returns a pointer to it. The DIB bitmap is created from the bitmap whose method is called. The DIB is allocated

internally using **LocalAlloc()** and must be freed by the developer using **LocalFree()**. The pseudo-code below show how this is done.

```
PBITMAPINFO pDib;  
pDib = bmap->ToDib();  
// Work with the bitmap...  
// ...  
// Free the bitmap when done  
LocalFree(pDib);
```

There is a different case where the developer receives a pointer to a bitmap, but is NOT responsible for deleting it when done. If you have a video post Image Filter plug-in, it receives a pointer to the video post bitmap queue. This bitmap should not be deleted as video post is using it internally. For example, in the **ImageFilter::Render()** method, the filter plug-in has access to a source bitmap named **srcmap**. The methods of this pointer may be called, but the bitmap itself should not be deleted by the plug-in. Video post will take care of it.

## Memory Management for Image Loader/Saver Plug-Ins

This section discusses how memory is managed internally by image loader / saver plug-ins. These are the plug-ins derived from class **BitmapIO**. Examples of this type of plug-in are the GIF, FLC and JPG IO modules. Developers who are not creating these types of plug-ins do not need to be concerned with these details.

The memory allocated to a bitmap is managed internally by an instance of **BitmapStorage**. This class provides access to the pixels through a uniform interface.

When a developer creates an image, the memory is allocated by the system. As long as the bitmap is being used within the system, the bitmap remains allocated. This is handled internally by a usage counter that tracks if a bitmap is still being used. If another use of the bitmap takes place, the usage count is incremented. If the use of a bitmap ends, the usage count is decremented. When the usage count for the bitmap goes to zero, the system frees the memory. This happens automatically without intervention from the plug-in.

For example, once a bitmap is loaded (this is never the case when a bitmap is created), a storage for the image is created to hold the actual bitmap. If a second attempt to open this same bitmap is made (the same file and the same frame), instead of creating a new storage, the bitmap is created pointing to the existing storage. In the storage a counter is incremented to tell how many bitmaps are using it. When a developer deletes the bitmap (i.e. **delete MyMap;**) the destructor calls the storage and asks to be unlinked from the storage. The storage decrements the usage count and if it reaches 0, the storage itself is also deleted.

Note that this is only the case when a bitmap is loaded because when you create a bitmap, it doesn't yet exist in the file system. There is no way for someone else to "open" it. This is why when you create a bitmap, it is said this is a "WRITE ONLY" bitmap, and when you load a bitmap it is said this is a "READ ONLY" bitmap. In order to read and write a bitmap you must load the original bitmap, create a second, copy the data (doing whatever processing in between), and then write the newly created bitmap.

## **Bitmap Adjustment - Changes to Resolution and Color Depth**

There are several ways to change the resolution of a bitmap. In each case this involves creating a new bitmap and copying the existing bitmap to the new. To change the image size without any scaling of the pixel data, use

**Bitmap::CopyImage()** and specify the **COPY\_IMAGE\_CROP** operation.

To resize the bitmap you use **Bitmap::CopyImage()** as well. This method lets you specify either a low quality (faster) or a higher quality (slower) copy. See [List of Copy Image Operations](#) for examples of each.

To change the color depth (number of bits per pixel), you must create a new bitmap of the desired color depth, and copy the original to the new using **Bitmap::CopyImage()**.

## Palettes

Some color bitmaps use only 8-bits per pixel. These images, unlike true color images where every pixel can be a unique color, are limited by the colors stored in a palette. For example, an 8-bit GIF file has a 256 color palette. The color values stored by the palette can be any color, but the actual bitmap image can only be comprised of colors from the palette. And since the palette is limited to 256 colors, the image has a maximum of 256 unique colors. For paletted bitmaps, each pixel in the image is actually an index into the palette (sometimes called a color lookup table). So the pixel value tells the system which palette slot to look in, and the value in that palette slot determines the exact color.

In contrast, true color images, for example a 24-bit TGA file, store the color of the pixel directly in the pixel value. There is no palette used.

In MAX, every bitmap has storage for a palette even if it is not used. There are methods of the `Bitmap` class used to work with palettes. A developer may use **`Bitmap::IsPaletted()`** to determine if an image is indeed paletted. To access the palette of a bitmap, **`Bitmap::GetPalette()`** and **`Bitmap::SetPalette()`** may be used.

Palettes are primarily a concern for image loader plug-in derived from **`BitmapIO`**. When loading an 8 bit image, the loader would create an 8-bit storage and set the palette through **`BitmapStorage::SetPalette()`**. Sample code is available showing how this is done in `\MAXSDK\SAMPLES\IO\BMP\BMP.CPP` in the **`Load()`** method.

## Pixel Storage

There are several in-memory storage formats for pixel data. For a list of the available formats see the section [Pixel Storage Types](#). Also see [Class PixelBuf](#), and [Template Class PixelBufT](#) for some useful utility classes that manage single scanline buffers.

## Pre-Multiplied Alpha

The following is a discussion of the concept of pre-multiplied alpha as used by MAX. A 32-bit bitmap file contains four channels of data: red, green, blue, and alpha. The first three provide color information to the pixels, while the alpha channel provides transparency information. There are two methods of storing alpha in a bitmap -- pre-multiplied and non-pre-multiplied.

To composite an image that is in non-pre-multiplied format, the alpha must be multiplied by each of the R,G, and B channels before adding it in to the color of the background image. This provides the correct transparency effect, but must be done each time you composite. With pre-multiplied alpha, you store the R,G, and B components with the alpha already multiplied in, so compositing is more efficient.

However, this is not the only reason that 3ds max stores images in the pre-multiplied format. When you render an image, you typically want the edges of the objects to be anti-aliased. This effect is achieved by determining the fractional coverage of pixels on the edge of the object, and then adjusting the alpha of the pixel to reflect this. For example, a pixel which is 30% covered by the object will have an alpha of 0.30. To anti-alias the edges, the alpha **must** be pre-multiplied to darken these edge pixels. (This is basically equivalent to compositing the image over a black image). So it is natural, in a sense, for rendered images to have pre-multiplied alpha. If you didn't pre-multiply in the alpha of a rendered image, then just looking at the RGB you would see jaggies on the edges of objects: you'd have to composite it against black using the alpha channel whenever you wanted to display it.

Pre-multiplied alpha works as follows: If you have an image A which is partially transparent, and you want to composite it over an image B, the alpha channel of A tells you at each pixel how much of A and B to mix in. If A's alpha is pre-multiplied (as it always is in MAX) then the formula is:

$$\text{color} = A + (1-A.\text{alpha})*B$$

alpha can also be thought of as the "opacity" of A at a given pixel.

If image A was stored with Non-Pre-Multiplied Alpha (NPMA) then the formula for compositing would be:

$$\text{color} = A.\text{alpha}*A + (1-A.\text{alpha})*B$$

To understand why pre-multiplied alpha is used consider an anti-aliased edge of an object, rendered against a black background. The pixels along the edge will

have an alpha less than 1.0, and when the alpha is multiplied in, it will make the edge look smooth (i.e. anti-aliased). If you **don't** pre-multiply the alpha, the RGB image displayed as-is (with out taking into account alpha) looks jagged: multiplying by the alpha is what anti-aliases the image.

This begs the question "Why would you ever use non-pre-multiplied alpha?". Say you have an image without an alpha channel. You want to create an alpha channel to mask out all but a certain object, but want to leave the original image unchanged: you may be using the same image at other places in your rendering. In this case, you would want to paint an alpha channel using an image editor (such as Photoshop) which would mask out the image, and combine it using the non-pre-multiplied alpha formula.

MAX's Mask texture map allows you to do this, in fact. Basically Mask lets you combine a texture with an mask channel, where the mask channel acts as a NON pre-multiplied alpha channel. What the mask texture actually does is multiply all four channels of the map with the mask channel, so what is passed up the pipeline is RGBA with pre-multiplied alpha.

## Dithering and Filtering

When converting images with a palette of a greater number of colors to an image with a palette of fewer colors, dithering is a means of simulating colors not in the more limited palette by mixing different colored pixels together.

Dithering is also a method of smoothing the edges between two color regions by mixing their pixels so the edges appear to blend together.

In MAX, you have the option of setting dithering if you are rendering for the limited colors of an 8-bit display (256 colors). It can help prevent a banding effect in color gradients. Dithering does increase the size of 8-bit files and slows down the playback speed of animations.

3ds max is designed to render 64-bit color output. Consequently, you also have the option of setting dithering for true color (24 or 32-bit color). The Dither True Color option ensures that you get the best quality on true-color displays.

Users turn dithering on and off in the Rendering page of the Preferences dialog. Users can also set dithering for scene motion blur in Video Post. Here, dithering provides a smoothing effect between the separate images making up the "blur". Video Post dither is set as a percentage of total dither.

A developer can determine if a bitmap is dithered by calling **Bitmap::IsDithered()**. A developer can ask the system to dither an image at render time using the method **Bitmap::SetDither()**.

Normally, a developer is not directly concerned with the dithering or filtering of bitmaps. These two operations are performed by the renderer, and the 3ds max user sets these characteristics of the bitmaps using the 3ds max user interface. A developer can however determine if a bitmap will be filtered by calling **Bitmap::HasFilter()**. A developer can also ask the system to filter an image when it's rendered. This is done using the method **Bitmap::SetFilter()**. See [List of Bitmap Filter Types](#) for more details on the types of filtering available. There is also a method of the **Bitmap** class called **GetFiltered()** to compute averaged colors over a specified area of the bitmap using the bitmap's current filtering algorithm.

## Gamma Correction

This section contains an overview of the concept of gamma followed by a discussion of the methods available to developers in dealing with bitmaps and gamma correction.

Gamma correction compensates for the differences in color display on different output devices so that images look the same when viewed on different monitors. A gamma value of 1.0 corresponds to an "ideal" monitor; that is, one that has a perfectly linear progression from white through gray to black. However, the ideal display device doesn't exist. Computer monitors are "nonlinear" devices. The higher the gamma value, the greater the degree of nonlinearity. The standard gamma value for NTSC video is 2.2. For computer monitors, gamma values in the range of 1.5 to 2.0 are common.

When you create an image on your computer, you base your color values and intensities on what you see on your monitor. Thus, when you save an image that looks perfect on your own monitor, you're compensating for the variance caused by the monitor gamma. The same image displayed on another monitor (or recorded to another media affected by gamma) will look different, depending on that media's gamma values.

Two basic procedures are required to compensate for changes in gamma:

1. Calibrate your output display devices so that the midtones generated by 3ds max are accurately duplicated on your display device. You do this in the Gamma panel of the Preferences dialog (Display Gamma).
2. Determine the gamma value to be applied to files output by the 3ds max renderer and files input into 3ds max, such as texture maps. This control is also in the Gamma panel of the Preferences dialog (Files Gamma).

The most important rule about gamma correction is to do it only once. If you do it twice, the image quality is over bright and loses color resolution.

With regard to output file gamma, video devices such as video tape recorders have their own hardware gamma-correction circuitry. Therefore, you need to decide whether to let 3ds max do the output gamma correction or to let the output device handle it.

Gamma correction is not required for hardcopy print media.

Files coming into 3ds max from programs such as Adobe Photoshop will have been gamma-corrected already. If you've been viewing the files on the same monitor and they look good, you won't need to set input file gamma in 3ds max.

A developer can indicate that a bitmap should have a custom gamma setting using the method **BitmapInfo::SetGamma()**. To retrieve the gamma setting stored with the bitmap use **Bitmap::Gamma()**.

There are several ways to access pixel values in a bitmap. Some of these methods return gamma corrected pixels while other do not. Normally, a plug-in that access the pixels directly (for example an image filter plug-in that modifies the pixels) should use the method **GetLinearPixels()**. This method returns pixels that are NOT gamma corrected. The method **GetPixels()** is employed to access pixels that are gamma corrected.

Developers may also use the methods of [Class GammaMgr](#) to gamma correct colors.

## Aspect Ratio

Aspect ratio is usually expressed either as a ratio of bitmap width over bitmap height (for example, 4:3) or as a real value relative to 1 (for example, 1.333).

Methods are available in the **BitmapInfo** class to get and set the aspect ratio property of the **BitmapInfo** (**BitmapInfo::Aspect()** and **SetAspect()**).

Methods are available from the **Bitmap** and **BitmapStorage** classes to return the value of the **BitmapInfo** instance associated with the **Bitmap** or **BitmapStorage**.

3ds max also allows users to set the pixel aspect ratio, so that there is a different value for the distance covered by one pixel measured horizontally and one pixel measured vertically. A developer can check the pixel aspect ratio setting that is being used by the renderer using the method **Interface::GetRendAspect()**.

## **Hot Check Utilities**

There are functions that may be used to correct a pixel with RGB values that will give "unsafe" values of chrominance signal or composite signal amplitude when encoded into an NTSC or PAL color signal. This happens for certain high-intensity, high-saturation colors that are rare in real scenes, but can easily be present in computer generated images. See: [List of Video Color Check Utilities](#).

## G-Buffer Image Channels

Image Filter and Image Layer events in Video Post can use masks that are based on geometry/graphics buffer (G-Buffer) channels instead of the more widely used RGB and alpha channels. Also, some kinds of Filter and Layer events can post-process objects or materials designated by these channels. The 3ds max G-Buffer system allows developers to access additional data about rendered objects.

A G-buffer is used to store, at every pixel, information about the geometry at that pixel. All plug-ins in video post can request various components of the G-buffer. When video post calls the renderer it takes the sum of all the requests and asks the renderer to produce the G-buffer.

This allows a developer writing an image processing plug-in to locate parts of the image using a specific material, locate a specific node in the scene, and access UV coordinates, surface normals, and unclamped color values. This also allows the developer to access the Z (depth) buffer. In release 3.0 and later developers can access, color, transparency and weight for sub-pixel fragments as well as velocity (for motion blur).

A filter plug-in (derived from class **ImageFilter**) implements a method **ChannelsRequired()** to indicate what channels it needs. Then, at the time the filter's **Render()** method is called it will have access to these channels. See [List of Image Channels](#) for details on the available channels.

## Error Reporting

When an image loader/saver plug-in (derived from class **BitmapIO**) encounters an error during a bitmap operation (for example "No Disk Space" when attempting to write a bitmap) there is a system flag that controls how the error should be reported to the user.

The image loader/saver base class **BitmapIO** provides a method for reporting errors named **ProcessImageIOError()**. This presents a standard dialog that provides the user with options to cancel or retry the operation. So, when a developer runs into an error condition while processing a bitmap, they would use the **BitmapIO::ProcessImageIOError()** method to report it. Some "common" error messages are already defined, and for those you would simply use one of the error codes defined in **BITMAP.H**. These are:

**BMMRES\_MEMORYERROR** Generic memory error.

**BMMRES\_CANTSTORAGE** Generic can't create storage error.

**BMMRES\_BADFRAME** Generic Invalid Frame Number Requested.

More of these exist but are for internal use only. To send your own message, simply pass a **TCHAR** string instead. The file name and/or device name are taken from the given **BitmapInfo** object.

The **ProcessImageIOError()** returns either **BMMRES\_ERRORRETRY** or **BMMRES\_ERRORTAKENCARE** depending on the users selection from the dialog box. Normally a developer doesn't care about the return value and simply returns it and exits.

The idea of a standard error processing dialog is to control the display of dialogs. For example, when 3ds max is running in network rendering mode, no dialogs should be displayed. That would cause the machine to just sit there since there would be no user to respond to the dialog.

If you must handle the error yourself (that is, if you want to display your own error dialog), you should first check to see if dialogs are allowed by checking the **BitmapManager::SilentMode()** method. This method returns a value indicating if dialogs should indeed be displayed or not.

## Utility Functions for Use with Bitmaps.

The following functions are general utility routines for dealing with bitmap files (but are not methods of a specific class):

### Prototype:

```
BOOL BMMCreateNumberedFilename( const TCHAR  
*namein,DWORD frame, TCHAR *nameout );
```

### Remarks:

Implemented by the System.

This appends a 4 digit frame number string to the end of the name passed. For example, this will convert **BIGFILE.TGA** to **BIGFILE0000.TGA** (or **BIGF0000.TGA**). This function checks the file system to see if it supports long file names and manages the length appropriately.

### Parameters:

**const TCHAR \*namein**

The input name to append the numbers to.

**DWORD frame**

The frame number to append.

**TCHAR \*nameout**

The output string.

### Return Value:

TRUE if the function succeeded; otherwise FALSE.

### Prototype:

```
BOOL BMMGetFullFilename(BitmapInfo *bi);
```

### Remarks:

This function will search the system for a bitmap. The BitmapInfo pointer contains the name of the bitmap that is searched for (**bi->Name()**). If the filename found in the BitmapInfo is incorrect, and the bitmap is found somewhere else, this function will replace **bi->Name()** with the correct path.

The order of the search is as follows:

- The full UNC path/filename saved in the BitmapInfo object.
- The path where the current 3ds max file was loaded from.

- The directory tree under the directory where the current Max files was loaded.
- The Map path.

**Parameters:**

**BitmapInfo \*bi**

Describes the bitmap to find (using **bi->Name()**). This name is updated if the bitmap is found in a different location.

**Return Value:**

TRUE if the file was found; otherwise FALSE.

**Prototype:**

**BOOL BMMIsFile(const TCHAR \*filename);**

**Remarks:**

Returns TRUE if the specified filename is indeed an existing file; otherwise FALSE.

**Parameters:**

**const TCHAR \*filename**

The filename to check.

**Prototype:**

**void BMMSplitFilename(const TCHAR \*name, TCHAR \*p,  
TCHAR \*f, TCHAR \*e);**

**Remarks:**

This function will break the specified filename into path, file and extension components. **\*p**, **\*f**, and/or **\*e** can be NULL. It is possible, for example, to call with just **\*e** to collect just the file extension.

**Parameters:**

**const TCHAR \*name**

The filename to split apart.

**TCHAR \*p**

The path name is stored here.

**TCHAR \*f**

The file name is stored here.

**TCHAR \*e**

The file name extension is stored here. The name includes the period character (.).

**Prototype:**

**void BMMAppendSlash(TCHAR \*path);**

**Remarks:**

This function appends a slash character to the end of the path passed unless one already exists.

**Parameters:**

**TCHAR \*path**

The path name to append.

**Prototype:**

**BOOL BMMGetUniversalName(TCHAR \*out\_uncname, const TCHAR\* in\_path, BOOL nolocal = FALSE);**

**Remarks:**

This function is available in release 4.0 and later only.

Given a path (**E:\path\filename.ext**), the function will check and see if this drive is mapped to a network share. If successful, the full UNC version will be returned in **out\_uncname** ("**\\computer\share\path\file.ext**"). If the function returns **FALSE**, **out\_uncname** will be undefined.

This function has been enhanced to also return an UNC for a local drive that happens to be shared. For instance, if you pass in something like **d:\data\images\maps\background\rottenredmond.tga** and it happens that d:\data is shared as "Image Data", the function will return:

**\\computername\Image Data\images\rottenredmond.tga.**

**Parameters:**

**TCHAR \*out\_uncname**

This is a buffer you pass to it to receive the UNC path (if any). It must be at least **MAX\_PATH** long.

**const TCHAR \*in\_path**

The path for which to obtain the UNC name.

**BOOL nlocal = FALSE**

Pass this as TRUE if you just want to see if this is a network share (don't check if this local drive is shared).

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**BOOL BMMFindNetworkShare(const TCHAR\* in\_localpath,  
TCHAR\* out\_sharename, TCHAR\* out\_sharepath);**

**Remarks:**

This method is available in release 4.0 and later only.

Given a path (**E:\path\filename.ext**) this function will check and see if this [local] path is shared. If successful, it will return both the share name and the path of the share.

**Parameters:**

**TCHAR \*in\_localpath**

The local path provided.

**TCHAR \*out\_sharename**

The share name which is returned if the provided path is shared.

**TCHAR \*out\_sharepath**

The path of the share which is returned if the provided path is shared..

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**BOOL BMMGetLocalShare(const TCHAR \*local\_path, TCHAR  
\*share);**

**Remarks:**

This method is available in release 4.0 and later only.

This method represents the "second half" of **BMMGetUniversalName()**

above. This method will check local paths only and return a UNC version if a share exists somewhere up in the path hierarchy.

**Parameters:**

**const TCHAR \*local\_path**

The local path provided.

**TCHAR \*share**

The share name which is returned.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**LPTSTR BMMGetLastErrorText(LPTSTR lpszBuf, DWORD dwSize);**

**Remarks:**

This function is available in release 2.0 and later only.

Whenever you call a Win32 function and there is an error, this method may be used to return the descriptive string associated with the error.

**Parameters:**

**LPTSTR lpszBuf**

This is the string that is updated.

**DWORD dwSize**

The maximum length of the string that may be returned in **lpszBuf**.

## Working with Controllers

See Also: [Class Control](#), [Class StdControl](#), [Class SetXFormPacket](#), [Class IKeyControl](#), [Class IKDeriv](#), [Class IKEnumCallback](#), [Class JointParams](#), [Class AngAxis](#), [Class Quat](#), [Class Matrix3](#), [Class Point3](#), [Class ScaleValue](#), [Class Interval](#).

## Overview

Controllers are the plug-in type responsible for directing all animation in MAX. This topic provides information on working with controllers. There is an overview of the principal classes involved when creating and working with controllers, a list of the various controller types in MAX, information on accessing the keyframe for the systems controllers, and a note about the importance of undo / redo as related to controllers.

## Principal Classes

The following classes are the main ones used when dealing with controllers in the SDK.

### [Class Control](#)

This is the base class for the creation of controller plug-ins.

### [Class StdControl](#)

This class provides a simplified way to create controller plug-ins. Developers who sub-class their plug-in from this class will have fewer methods to implement.

### [Class IKeyControl](#)

This class is used to access the properties of controllers. The standard 3ds max plug-ins support this interface. Plug-In developers are encouraged to do so as well. This allows other third-party developers to access your controller's data.

### [Class SetXFormPacket](#)

This class is used to allow a transform controller to know that it is being specifically moved, rotated, or scaled.

### [Class EaseCurveList](#)

Ease curves vary the timing of a superior function curve. A normal function curve charts an animated parameter value over time. An Ease curve charts changes to the time of a function curve over time. A 3ds max user may apply multiple ease curves to a parameter. This class holds a list of ease curves and allows developers to access their data.

### [Class MultCurveList](#)

The value of a multiplier curve is a scale factor applied to the value of its superior function curve. This class holds a list of multiplier curves and allows developers to access their data.

There are several interface classes that provide access to the parameters for the standard 3ds max controllers. These are listed below:

Path Controller: [Class IPathPosition](#).

Noise Controller: [Class INoiseControl](#).

Surface Position: [Class ISurfPosition](#).

Link Inheritance Controller: [Class ILinkCtrl](#).

Look At Controller: [Class ILookatControl](#).

## Controller Types in MAX

The basic data types that can be animated by controllers are:

Integer values (**int**).

Floating point values (**float**).

Three floats (**Point3**). Note: The **Point3** class is also used for Color Controllers.

Position (**Matrix3**).

Scale (**ScaleValue**).

Transform (**Matrix3**).

## Accessing Keyframe Data of Controllers

Developers have access to the data of MAX's keyframe and procedural controllers. There is a separate Advanced Topics section on this topic. See [Keyframe and Procedural Controller Data Access](#).

## The Undo Mechanism and Transform Controllers

The following is a discussion of how a transform controller responds to movement of the mouse when the user is dragging and how the undo mechanism is involved.

When the user first clicks the mouse button down to begin a drag operation, the initial position of a node is stored. As each move takes place a new vector is calculated from the initial position to the new position of the mouse. This new calculation is used to update the position of the node. This is different than accumulating the new position from the previous position. The accumulation approach can introduce error. It may also be a problem for the user to get back to the initial position of the node if they move the mouse back to the initial position.

3ds max uses the undo mechanism to avoid the problems associated with the accumulation approach. Every time the user moves the mouse 3ds max re-computes the entire modification. So 3ds max uses the original point and the current location of the mouse to compute the position. What this means internally is that every time the user moves the mouse the state at initial mouse down must be restored. To do this, when the mouse is initially pressed, **theHold.Begin()** is called to create a **RestoreObj** that stores the initial state. When the mouse is moved, **theHold.Restore()** is called. This restores the state to when **theHold.Begin()** was called.

As noted above, when the user initially mouses down, **theHold.Begin()** is called. Then at each mouse move **theHold.Restore()** is called. In iterative operations such as this it is often useful to set one of the flags of **Animatable** to indicate that a restore object is being held. In the example above, when the user first clicks down on the mouse the developer checks if theHold is holding and if it is calls **theHold.Put()** to register a restore object. Then the developer calls a method of **Animatable SetAFlag(A\_HELD)**. This sets the **A\_HELD** bit of the **Animatable aflag** data member to indicate the restore object is held. Then on each iteration the bit is tested to see if it is set and if so another restore object is not registered. A single restore object can be restored over and over again. When **theHold.Accept()** or **theHold.Cancel()** is called, the system calls a method of the restore object called **EndHold()**. The developer may then clear the **A\_HELD** bit to indicate the restore object is no longer being held.

For sample code that does this see any of the transform controllers, for

example,

**\MAXSDK\SAMPLES\CONTROLLERS\BOOLCTRL.CPP**. Also, see the Advanced Topics section on [Undo / Redo](#).

## Global Functions

There are some global functions developer may use that relate to controllers. These provide things such as setting the default controllers used for various controller types, creating new instances of different controller types, turning animation mode on and off, temporarily suspending animate mode, and getting/setting the animation start and end times. There are also some global functions which provide access to the default tangent types for both the Bezier and TCB controllers. See [List of Additional Controller Related Functions](#) to review them.

## Working with Lights

See Also: [Class LightObject](#), [Class GenLight](#), [Class Light](#), [Class DefaultLight](#), [Structure LightState](#), [Class LightDesc](#), [Class LightRayTraversal](#), [Class ObjLightDesc](#), [Class ShadeContext](#).

## Overview

This topic provides information on working with lights. Covered are the main classes used when dealing with lights and a description of how these relate to one another. Also discussed are the types of lights, and how the parameters that define their characteristics may be retrieve and assigned.

There are several tasks lights are responsible for accomplishing. One is to provide an interface so the user can adjust their properties. Another is to provide access to these same parameters so other plug-ins (utilities, exporters, renderers, etc.) can get and set them. Another one is the actual illumination objects in the scene. Also, certain lights need to work with the 3ds max atmosphere system (such as fog and volume lighting).

## Overview of the Principal Classes

The following classes are the main ones used when dealing with lights in the SDK.

### [Class LightObject](#)

This is a base class for plug-in lights. It has a method used to retrieve the light's properties at a certain time, a method to create an **ObjLightDesc** object (described below). It also has methods that allow the many properties of the light to be read and altered.

### [Class GenLight](#)

This is also a possible candidate for plug-in lights to sub-class from. This class, sub-classed from **LightObject**, provides a set of additional methods (for example access to the controllers handling the animation of the light's properties).

### [Class Light](#)

This class describes the properties of the lights used in the interactive renderer. See [Light methods in Class GraphicsWindow](#) for methods which use this class.

### [Class DefaultLight](#)

An array of these objects is passed in the **Open()** method of a renderer plug-in. This class contains the transformation matrix of the light and a **LightState** object (described below) that describes the characteristics of the light.

### [Structure LightState](#)

This structure describes the properties of a light. When the **LightObject** method **EvalLightState()** is called, **LightState** is the structure that is updated. This is used, for example, by many of the file format export plug-ins to grab data about the lights they are exporting.

### [Class LightDesc](#)

This class has a single method **Illuminate()** used to determine the color and direction of the light striking a point. It has two public data members that indicate if the diffuse and/or specular color of objects should be affected by the light.

### [Class ObjLightDesc](#)

This class is sub-classed from **LightDesc** (described above). There is an

instance of this class for every instance of a light in the scene. The renderer asks each light in the scene to create one of these objects by calling **LightObject::CreateLightDesc()** and passing it the light node. This class has data members that provide information such as a pointer to the node for the light, a **LightState** object, various transformation matrices that provide conversions between the light's space and world and camera space. It has a method to retrieve the lights exclusion list, and a method to update the light's data members in this class once per render. It also has a method used in computing volume lighting effects, **TraverseVolume()**.

## Accessing Light Parameters

There are several approaches developers may use to access the parameters that define the properties of a light. One way is to call the **EvalLightState()** method of the light. This method updates the **LightState** structure passed to it. This structure contains basic information about the light such as its type, color, attenuation ranges, etc.). The structure is valid for a single point in time for animated lights.

Another approach is to call the individual Get/Set methods of the light directly. Since lights are sub-classed from **LightObject** or **GenLight**, the methods of these classes may be called to get and set additional properties. This provides not only read access but also the ability to change the lights properties. If animate mode is set to on, a developer may animate the lights in this way.

Finally, developers may access the controllers that handle the animation of lights. This can be used to determine exactly what type of interpolation is being done by the lights when they are animated (by looking at the Controller's Class\_ID). Also, the **GetValue()** and **SetValue()** method of the controller may be used to access the lights properties.

## **Global Light Scaling and Tinting**

In 3ds max 2.0 a new area, called 'Global Lighting' has been added to the Environment dialog. It contains two new controls that globally affect all lights in the scene, except the Ambient light. The two new controls are as follows:

**A Level spinner control:** The value in this spinner acts as a multiplier to all lights in the scene, except ambient light. Thus, a level of 1 (default) preserves the normal light settings, while higher numbers raise the lighting, and lower numbers reduce the lighting.

**A Tint color swatch:** The user may use this for to tint all lights in the scene by that color.

Both of these new controls can be animated. Developers that retrieve light color values from the Get/Set methods of the lights themselves do not have this global lighting factor applied. There are several methods of the Interface class that provide access to these properties. See [Interface -- Environment Access](#).

## Working with Materials and Textures

See Also: [Class MtlBase](#), [Class Mtl](#), [Class Texmap](#), [Class ShadeContext](#), [Class StdMat](#), [Class Mesh](#), [Class INode](#).

## Overview

This section provides information on working with materials and textures. This includes an overview of the principal classes, how plug-ins can access components of materials (such as their maps, colors, etc.), and how materials may be created and applied to nodes in the scene. Information is also provided about how the user interface is managed for plug-in materials and textures.

A texture and a material are similar in many ways. In MAX, every texture or material may have sub-texture or sub-materials. Materials that have sub-materials are referred to as meta-materials.

## Overview of the Principal Classes

The following are the main classes associated with **creating** plug-in materials and textures.

### [MtlBase](#)

This is the base class that the materials and texture classes are derived from.

### [Mtl](#)

This is the class plug-in materials are sub-classed from.

### [Texmap](#)

This is the class plug-in procedural textures are sub-classed from.

### [ShadeContext](#)

An instance of this class is passed into various methods associated with materials and texture maps. It provides data members and methods used in communication between the renderer and the plug-in material or texture.

### [RenderGlobalContext](#)

A pointer to an instance of this class is a data member of the ShadeContext (**RenderGlobalContext \*globContext**). This can be used by materials, texmaps, etc. to retrieve information about the global rendering environment. This is information such as the renderer in use, the project type for rendering, the output device width and height, several matrices for transforming between camera and world coordinates, the environment map, the atmospheric effects, the current time, field rendering information, and motion blur information.

### [IMtlParams](#)

This is an interface class passed to a material or texture map when it is being edited in the materials editor. The class has methods to do things such as add rollup pages to the dialog, remove rollup pages, and retrieve the current time (frame slider position).

### [ParamDlg](#)

Every **MtlBase** sub-class (i.e. texture map and material) defines a **ParamDlg** to manage its part of the material editor user interface. See the section below on Editing Material or Texture Parameters for details on how this class is used.

### [UVGen](#)

Most texture maps that use UVs will use a **UVGen**. This class encapsulates a

lot of the user interface for setting mirroring, tiling and so on. The **UVGen** is given a **MapSampler** callback, and the plug-in lets the **UVGen** call it. This lets the **UVGen** figure out the components for anti-aliasing, and it includes any transform including scaling, rotation, moving, and noise on the UVs.

### [MapSampler](#)

A texture map implements this class and passes it into the UVGen methods **EvalUVMaP()**, **EvalUVMaPMono()**, and **EvalDeriv()** to evaluate itself with tiling & mirroring. Each of the methods of this class are used to sample the map at a single UV coordinate.

### [XYZGen](#)

This class generates **Point3** coordinates based on the **ShadeContext**. A reference to one of these is referenced by all 3D texture maps. **XYZGen** does for 3D Texmaps what **UVGen** does for 2D Texmaps. It puts up the 3D "Coordinates" rollup, and supplies the 3D Texmap with transformed 3D coordinates.

### [TextureOutput](#)

This class may be used by textures to provide control over their output. The standard rollup page 'Output' is managed by this class. The Output Amount, RGB Level, and RGB Offset are settable. In the future this may be enhanced to include other things that are often desirable on the output stage such as tinting, color shifting, etc.

The following are the main classes associated with **accessing** properties of materials and textures:

### [StdMat](#)

This class provides access to the parameters of the 3ds max Standard material.

### [MultiMtl](#)

This class provides access to the developer alterable properties of the 3ds max Multi/Sub-Object material.

### [BitmapTex](#)

This class provides access to the parameters of the 3ds max Bitmap texture.

### [StdUVGen](#)

This class provides access to the parameters of the 3ds max UVGen class. These are the settings in the 'Coordinates' and 'Noise' rollups such as UV offsets, angle, blur, noise level, etc.

### [StdXYZGen](#)

This class provides access to the parameter of the 3ds max XYZGen class.

### [MtlBaseLib](#)

This class provides access to the materials stored in a materials library. See the sample code below for an example of use.

## Accessing Material Properties

This section covers how the various properties of a material may be retrieved from a node in the scene. These are properties such as color, mapping parameters, bitmaps used, etc.

In 3ds max there is only one material per node. In the **INode** class there are methods **GetMtl()** and **SetMtl()** that provide access to the node's material. **GetMtl()** returns a pointer to an instance of class **Mtl**.

**virtual Mtl \*GetMtl()=0;**

Returns the renderer material for the node. If the value returned is NULL this means the user has not assigned a material to the node. In such a case the renderer simply uses the wireframe color of the node (as well as many defaults) for the rendering properties.

The material returned can be any material that may be assigned by the user. These include those from 3ds max itself as well as those created by third party developers. One may look at the **Class\_ID** of the material to determine its type. For example, the 3ds max Standard material has a **Class\_ID** of **DMTL\_CLASS\_ID** while the Multi/Sub-Object material uses **MULTI\_CLASS\_ID**. See [List of Class IDs](#) to review the complete list provided by MAX.

Let's consider the case of accessing properties of the 3ds max Standard material first. If the **Class\_ID** returned is **DMTL\_CLASS\_ID** then it's a Standard material. This material does not have any sub-materials, but has many parameters that a developer may need to access.

The primary class for accessing the Standard material is [Class StdMat](#). It provides methods to get and set the material's properties. These are things like the shading limit, the diffuse, ambient and specular colors, and the shininess and opacity of the material. Developers may also need to access the mapping parameters, for example, to find the texture used in the diffuse slot. A developer may access these using **MtlBase::GetSubTexmap(int i)** and **MtlBase::SetSubTexmap(int i, Texmap \*m)**. The index passed to these method are a series of #defines. See [List of Texture Map Indices](#).

Next, let's consider access to the properties a Multi/Sub-Object material. If the **Class\_ID** returned is **MULTI\_CLASS\_ID** then it's a Multi/Sub-Object material. The multi-material is a plug-in that uses the **MtlID** assigned to each face of a mesh as an index into the list of sub-materials. There are methods of

the [Face](#) class **getMatID()** and **setMatID()** to access the MtlID for each face. Additionally, the [Mesh](#) class has methods **getFaceMtlIndex(int i)** and **setFaceMtlIndex(int i, MtlID id)**.

Note that the meaning of this material index assigned to the face is specific to the type of material. A third party developer could write a material that used the ID in an entirely different manner than 3ds max does.

For a multi-material, a developer can use the methods of **MtlBase**: **NumSubMtls()**, **GetSubMtl(i)** and **SetSubMtl(i, Mtl\* m)** to access the sub materials. Note that some 3ds max primitive have pre-assigned material ids (such as the Box and the Hedra). Since the object is not really aware of the material, it can happen that the material ID on a face is higher than the total number of sub materials in a material. Developers should look at the number of sub materials on the material, and use a modulo function to bring the material ID of the face down to a legal value before trying to access it.

The following code demonstrates access to material and texture properties. It shows how the ClassIDs are checked for the material and the texture map. It also demonstrates how to access materials properties using the **StdMat** class and the bitmap properties using the **BitmapTex** class.

```
// Test of access to material and texture map properties of a selected node.  
// This code determines if the two sided flag is set for the material  
// and retrieves the U and V tiling options for the diffuse texture.  
// It checks if the material is a 3ds max Standard material, and if the diffuse  
// texture map is a 3ds max Bitmap texture.  
#include "stdmat.h"  
void Utility::Test (Interface *ip) {  
    BOOL two;  
    float utile, vtile;  
    TSTR buf;  
  
// Get the material and texture properties from the node.  
if (ip->GetSelNodeCount() < 1) return;  
INode *node = ip->GetSelNode(0);
```

```

// Get the material from the node
Mtl *m = node->GetMtl();
if (!m) return; // No material assigned
// See if it's a Standard material
if (m->ClassID() == Class_ID(DMTL_CLASS_ID, 0)) {
// It is -- Access the two sided property of the material
StdMat* std = (StdMat *)m;
two = std->GetTwoSided();
// Access the Diffuse map and see if it's a Bitmap texture
Texmap *tmap = m->GetSubTexmap(ID_DI);
if (tmap->ClassID() == Class_ID(BMTEX_CLASS_ID, 0)) {
// It is -- Access the UV tiling settings at time 0.
BitmapTex *bmt = (BitmapTex*) tmap;
StdUVGen *uv = bmt->GetUVGen();
utile = uv->GetUScl(0);
vtile = uv->GetVScl(0);
buf.printf(_T("Two sided=%d, U Tile = %.1f, V Tile = %.1f"),
two, utile, vtile);
MessageBox(ip->GetMAXHwnd(), buf, _T("Info..."),
MB_ICONINFORMATION);
}
}
}

```

## Accessing Material Libraries

3ds max stores groups of materials in a library. The API provides methods to access the materials stored in these libraries. You can use the methods of class **Interface** to load and save material libraries. These methods are **LoadMaterialLib()** and **SaveMaterialLib()**. This class also has a method **GetMaterialLibrary()** which returns a reference to the currently loaded library. You may then use class **MtlBaseLib** to access the materials themselves. The sample code below shows how this is done. It loads a materials library, removes a single material, and saves it again.

```
void Utility::TestMaterialLib() {
    TCHAR *n1 = _T("D:\\3DSMAX\\MAPS\\3DSMAX.MAT");

    int okay = ip->LoadMaterialLib(n1);
    if (! okay) return;

    // Declare this as references since we want to manipulate the
    // original material library and not a copy.
    MtlBaseLib &mlib1 = ip->GetMaterialLibrary();
    int index = mlib1.FindMtlByName(TSTR(_T("Aqua Glaze")));
    if (index != -1)
        mlib1.Remove(mlib1[index]);
    ip->SaveMaterialLib(n1);
}
```

## Texture Coordinates

This section discusses how UVW texture coordinates are interpreted by the renderer for texture maps in 3ds max. The **Mesh** class provides access to the texture coordinates of a Mesh. The **PatchMesh** class provides access to the texture coordinates of a Patch. Both use a very similar approach. In this section we'll look at how this is done for the Mesh class.

The texture coordinates are available in the Mesh class public data member **UVVert \*tVerts;**

Note that a **UVVert** is simply a **Point3**, i.e.: **typedef Point3 UVVert;** **tVerts** is a pointer to the list of texture vertices. Each **UVVert** stores a single UVW coordinate (as the X, Y and Z public data members of the **Point3**). The UVW coordinates parallel the relative direction of the XYZ coordinates. If you look at a 2D map image, U is equivalent to X, and represents the horizontal direction of the map; V is the equivalent of Y, and represents the vertical direction of the map; W is the equivalent of Z and represents a direction perpendicular to the UV plane of the map. For a 2D mapping, only two of the coordinates are used, i.e. UV, VW, or WU. This provides greater flexibility for the user as it allows them to flip the orientation of the map relative to the geometry. Radio buttons in the Materials Editor user interface allow the user to choose which two are used. The texture coordinates don't provide exclusive control over the mapping -- the material containing the map may apply other transformations that affect how the texture coordinates are interpreted by the renderer. First we'll look at the default case where the material imparts no additional transformations. Later we'll see what happens when the material applies a transformation, and how we can access and interpret additional material transformations.

The texture coordinates stored in the **tVerts** array can be any floating point values. If we assume that the settings in the material do not apply any additional transformations (the default settings), then an image is fit in its entirety between texture coordinates 0.0 to 1.0. For example, if the two triangles pictured below in Figure 0 had the mapping coordinates shown at each vertex, the bitmap image in figure 1 would be mapped across the entire two triangles. This mapping is shown in Rendering 1. As you can see, the mapping coordinates from 0.0 to 1.0 provide an exact match, fitting the bitmap across the entire quad.

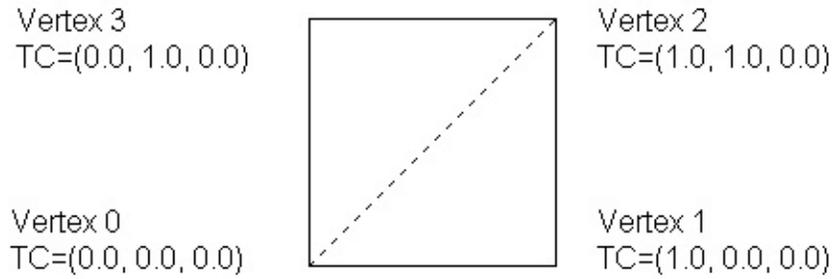
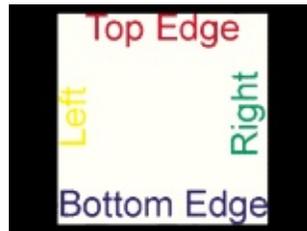


Figure 0:

Note: In figure 0 and those below 'TC' stands for Texture Coordinate (i.e. the value in the **tVerts** array)



Figure 1: The original bitmap (on a black background).



Rendering 1:

The next example shows how an image can be tiled across the geometry by using mapping coordinates greater than 1.0. The X mapping coordinate for vertex 1 and vertex 2 are 2.0 (and the material tiling check box are enabled in the Materials Editor). This results in the image being repeated in the U direction twice as shown below. Note that it is only tiled once in V since the Y coordinates are still 1.0.

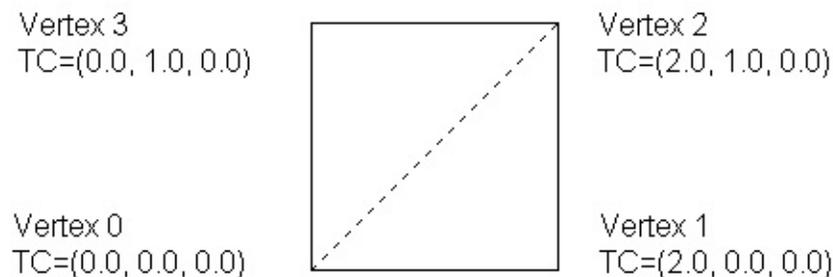


Figure 2:



Rendering 2:

The next example shows how negative texture coordinates are used to reverse the way the image is mapped to the geometry. In the figure below, the X texture coordinates are -2.0 for vertex 1 and vertex 2. This results in the image being mirrored about the vertical axis of the map. It is also repeated twice since the coordinates are -2.0 and not -1.0.

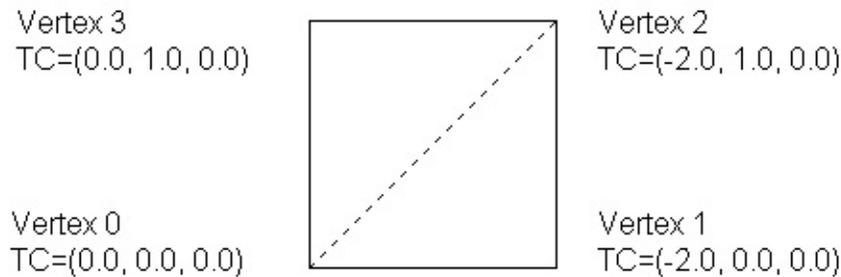


Figure 3:



Rendering 3:

The next diagram shows how mapping coordinate greater than 0.0 start the image off at an offset within the image. Here the coordinates are the same as in example 1, except all the values have been shifted by 0.5 in X and 0.5 in Y. This results in the image being shifted on the geometry. Here again, tiling is on, so the pattern begins in the center of geometry, but is tiled and thus reappears from the left and bottom edges.

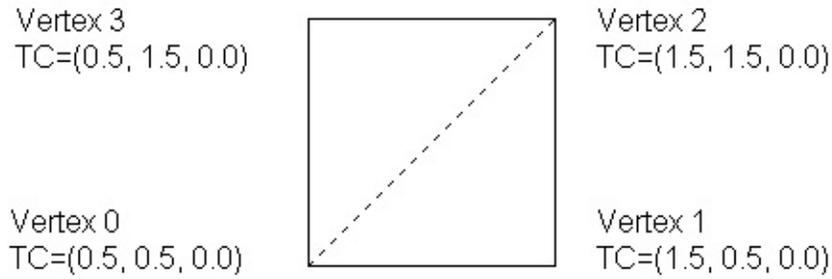


Figure 4:



Rendering 4:

There is something special about UVW coordinates in the range of 0.0 to 1.0. If image tiling is turned off in the Materials Editor (the 'Tile' check box is unchecked for a certain direction), then the image will only appear where the texture coordinates are in this range. UV values outside the range 0.0 to 1.0 will not be mapped when tiling is off in that direction. This is shown in the diagram below. The material editor settings have tiling off in both the U and V directions, and the mapping coordinates are from [0.0 to 2.0]. Note that the image only appears where the mapping coordinates are in the range [0.0 to 1.0], i.e. the lower left corner.

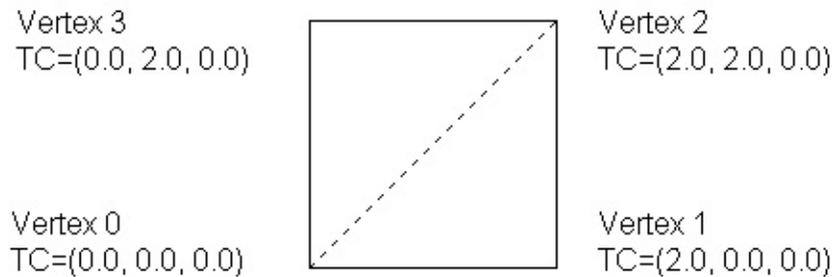
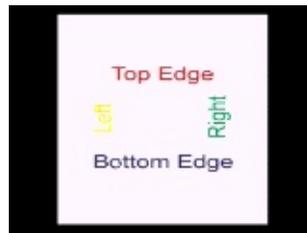


Figure 5:



Rendering 5:

As mentioned above, users may alter the tiling, mirroring, angle, and offset settings from within the material. These controls adjust the position of the map relative to the mapping coordinates. In the figure shown below, the material controls have been used to add an additional transformation to the mapping shown above in figure 5. The texture coordinates stored with the geometry are just the same, but a U Offset of 0.5 and a V Offset of 0.5 were used to shift the map onto the center of the geometry. Additionally, a U tiling setting of 0.75 was used to 'stretch' the mapping in the U direction. Thus the image is no longer square, but is rather elongated in U. Rendering 6 below shows the resulting image.



Rendering 6:

Developers may access all the users material transformation settings using the methods of class **StdUVGen**. The code shown above in the section **Accessing Materials Properties** sections shows how this is done.

## Texture Mapping

This section provides a quick look at how the texture coordinates described above are applied to objects in 3ds max using the standard mapping methods (planar, cylindrical, spherical, etc.).

In general this is quite simple. Basically, the geometric vertices of the object being mapped are transformed into some space (another coordinate system) and then these transformed geometric coordinates become the texture coordinates. For example in a simple planar mapping, the geometric vertices are transformed into the coordinate system of the mapping icon. Once transformed, the X, Y geometric coordinates become the UV mapping coordinates. For a cylindrical mapping the same thing is done except instead of taking just the X, Y coordinates, they are further transformed into a cylindrical coordinate system. Then these values become the UV mapping coordinates.

In camera mapping, where the texture coordinates of an object are used to match a background image, the geometric coordinates are transformed into camera screen space, and then screen X, Y becomes the UV coordinates.

## Face-Map Materials

There is another method of applying mapped materials that requires no application of mapping coordinates by the user whatsoever. In the Materials Editor, you can create a face-map material. When a face-map material is applied to an object, the map is automatically applied to each facet of the object. Below is the code used internally by 3ds max that computes the texture coordinates for face mapping:

```
static Point3 basic_tva[3] = {
    Point3(0.0,0.0,0.0),Point3(1.0,0.0,0.0),Point3(1.0,1.0,0.0)};
static Point3 basic_tvb[3] = {
    Point3(1.0,1.0,0.0),Point3(0.0,1.0,0.0),Point3(0.0,0.0,0.0)};
static int nextpt[3] = {1,2,0};
static int prevpt[3] = {2,0,1};

static void make_face_uv(Face *f, Point3 *tv) {
    int na,nhid,i;
    Point3 *basetv;
    /* make the invisible edge be 2->0 */
    nhid = 2;
    if (!(f->flags&EDGE_A)) nhid=0;
    else if (!(f->flags&EDGE_B)) nhid = 1;
    else if (!(f->flags&EDGE_C)) nhid = 2;
    na = 2-nhid;
    basetv = (f->v[prevpt[nhid]]<f->v[nhid]) ? basic_tva : basic_tvb;
    for (i=0; i<3; i++) {
        tv[i] = basetv[na];
        na = nextpt[na];
    }
}
```

## Bump Mapping in Procedural Textures

Bump mapping is a technique that enables a surface to appear wrinkled or dimpled without the need to model these depressions geometrically. Rather, the surface normal is perturbed according to information given in the 'bump map'. This results in variations to the smooth surface.

A 2D bump map is applied to the surface of an object. The space along the surface is called the UV space. The 2D bump map can be thought of as its own surface, where bright areas are hills and dark areas are valleys. On the surface then you can define a gradient, which can be thought of as the 'downhill' or 'uphill' directions. This gradient of values gives you a direction in UV space where you are going to perturb the surface normal. If you have a bump in a texture (a steep hill up for example) then you want to perturb the normal to make it look like there is a steep hill up there. Given the gradient in UV space, you need to figure out what direction in 3D (XYZ) space to perturb the normal that corresponds to that gradient in UV space. The **bump basis vectors** are used for this purpose. These are the vectors that represent the UVW axes of that texture in 3D. These are unit vectors that can be used to perturb the normal.

To understand how this is done, let's look at some sample code. The method responsible for returning the perturbed normal is

**Texmap::EvalNormalPerturb()**. The code below happens to be from `\MAXSDK\SAMPLES\MATERIALS\CHECKER.CPP` but all the other 2D textures use a similar approach.

```
Point3 Checker::EvalNormalPerturb(ShadeContext& sc) {  
    Point3 dPdu, dPdv;  
    if (!sc.doMaps) return Point3(0,0,0);  
    if (gbufID) sc.SetGBufferID(gbufID);  
    uvGen->GetBumpDP(sc,dPdu,dPdv);  
    Point2 dM = uvGen->EvalDeriv(sc,&mysamp);  
    return dM.x*dPdu+dM.y*dPdv;  
}
```

The first significant line related to bump mapping is:

```
uvGen->GetBumpDP(sc,dPdu,dPdv);
```

This method of **UVGen** gets the bump vectors. Developers can also get the bump vectors directly from the method **ShadeContext::DPdUVW()** although

these would not be affected by the **UVGen** transformations. In our case, since all the coordinates are coming through **UVGen**, we must use **GetBumpDP()** since the **UVGen** has rotated things around -- it has transformed the bump vectors to a new position. Basically, the UVGen has rotated the UV space into another position locally. So again, this method gets the bump basis vectors, which are really the U and V axes in 3D space (unit vectors in the U direction and the V direction, but in 3D space).

The next line computes **dM**. This is the derivative of the function across the pixel. This is the rate of change of the function in the U direction (**dM.x**) and the V direction (**dM.y**). So for example, if this is a flat function, these will both be zero. If the function is increasing in U but is constant in V then the value in the U direction will correspond to how fast it is changing while V will still be zero. Thus, **dM** can be thought of as the gradient -- how fast things are changing up and down.

**Point2 dM = uvGen->EvalDeriv(sc,&mysamp);**

Next, we need to compute the perturbation to the normal. This can be thought of as a small vector that will be added to the end of the existing normal that will move it over a little bit. There are several ways to do this. The common textbook algorithm (Blinn's algorithm for bump mapping) is not used by the 3ds max textures. Rather, the calculation shown below was found to be simpler and faster with no visual difference. To compute the perturbation to apply to the normal the following code is used:

**return dM.x\*dPdu+dM.y\*dPdv;**

This takes the sum of the U component (**dM.x**) multiplied by the U basis vector (**dPdu**) and the V component (**dM.y**) multiplied by the basis vector in the V direction. This gives the change (perturbation) to the normal as a unit vector.

The result of **EvalNormalPerturb()**, the perturbation to apply to the surface normal, is used by 3ds max as follows: Outside the procedural texture, for example in the Standard material, the value returned is added on to the surface normal. Then the normal is re-normalized (made a unit vector again). This altered normal results in the surface appearing 'bumped' when rendered.

## Assigning Materials to Nodes in the Scene

A developer may also wish to create materials and assign them to nodes in the scene. There are functions available for creating the standard 3ds max materials and textures such as Standard, Multi/Sub-Object, Bitmap texture, Composite texture, Mix, etc. These functions are not part of a class but are globally accessible (they are defined in `\MAXSDK\INCLUDE\STDMAT.H`).

### **StdMat \*NewDefaultStdMat();**

This function creates a new 3ds max Standard material. See [Class StdMat](#).

### **MultiMtl \*NewDefaultMultiMtl();**

This function creates a new 3ds max Multi/Sub-Object material. See [Class MultiMtl](#).

### **BitmapTex \*NewDefaultBitmapTex();**

This function creates a new 3ds max Bitmap texture. See [Class BitmapTex](#).

### **MultiTex \*NewDefaultCompositeTex();**

This function creates a new 3ds max Composite texture. See [Class MultiTex](#).

### **MultiTex \*NewDefaultMixTex();**

This function creates a new 3ds max Mix texture. See [Class MultiTex](#).

### **MultiTex \*NewDefaultTintTex();**

This function creates a new 3ds max Tint texture. See [Class MultiTex](#).

### **GradTex \*NewDefaultGradTex();**

This function creates a new 3ds max Gradient texture. See [Class GradTex](#).

### **StdCubic \*NewDefaultStdCubic();**

This function creates a new 3ds max Reflect/Refract texture. See [Class StdCubic](#).

### **StdMirror \*NewDefaultStdMirror();**

This function creates a new 3ds max Flat Mirror texture. See [Class StdMirror](#).

### **StdFog \*NewDefaultStdFog();**

This function creates a new 3ds max Fog atmospheric effect. See [Class StdFog](#).

Sample code using many of these functions is available in the 3D Studio DOS file import plug-in. This code may be found in `\MAXSDK\SAMPLES\IMPEXP\3DSIMP.CPP`.

The code below demonstrates how a 3ds max Standard material may be assigned to a mesh object. It assigns a material with Red Ambient and Diffuse settings to

the first node in the current selection set.

```
void Sample::AssignMtl(Interface *ip)  
{  
if (! ip->GetSelNodeCount()) return; // Nothing to assign.  
INode *node = ip->GetSelNode(0);  
// Create a new Standard material.  
StdMat *m = NewDefaultStdMat();  
// Set its properties...  
m->SetName(_T("Sample"));  
m->SetAmbient(Color(1.0f,0.0f,0.0f),0); // Pure Red  
m->SetDiffuse(Color(1.0f,0.0f,0.0f),0);  
m->SetSpecular(Color(1.0f,1.0f,1.0f),0);  
m->SetShininess(0.5f,0);  
m->SetShinStr(.7f,0);  
// Assign it to the node.  
node->SetMtl(m);  
ip->RedrawViews(ip->GetTime());  
}
```

## Editing Material or Texture Parameters in the Materials Editor

This section discusses the manner in which plug-in texture and materials manage their user interface in MAX. This is only a concern of developers creating plug-in materials or textures.

The way materials and texture maps handle their user interface is different than the way other plug-ins handle their UI in the command panel of MAX. For materials and textures, a developer derives a class from **ParamDlg** and implements its methods.

A method of **MtlBase** named **CreateParamDlg()** is called by the system when the material or texture is to be displayed in the material editor parameters area. This method is expected to create a new instance of a class derived from **ParamDlg**. The system then maintains the **ParamDlg** pointer. When the system needs to delete the memory associated with this instance it calls the method **ParamDlg::DeleteThis()**.

Within the instance of the class derived from **ParamDlg** a developer can store any data needed to handle the user interface (for example, spinner control handles, window handles, etc.). The class will also need to have a pointer to the 'thing' that is being edited. This 'thing' is either a texture or the material.

As the user works with different materials in the materials editor, these materials have to put up their user interfaces. For example, if the user is editing a Standard material in one sample window, then selects another Standard material in another sample window, the user interface changes to reflect the new material settings. The user interface does not 'flash' however -- in other words the entire rollup page is not deleted and replaced. Rather the fields are simply updated to reflect the new values. What the system is doing is effectively 'passing off' the user interface from one material to another. Two methods of **ParamDlg** allow this to happen. These are **SetThing()** and **GetThing()**.

When a system calls **SetThing()** it passes a pointer to a **ReferenceTarget**. This is the item that is being edited. So normally a developer would implement this method to store the item being edited (the 'thing') and update the user interface controls to reflect the state of the new 'thing'.

When the system calls **GetThing()**, the plug-in returns the 'thing' that is currently being edited.

As an example, consider the following code from the Checker texture map. Checker derives a class from **ParamDlg** and uses data members to store the

data needed for its operation. A portion of this code is shown below:

```
class CheckerDlg: public ParamDlg {
public:
    HWND hwmedit; // window handle of the materials editor dialog
    IMtlParams *ip;
    Checker *theTex; // current Checker being edited.
    HWND hPanel; // Rollup pane
    ISpinnerControl *blurSpin;
    IColorSwatch *cs[2];
    TimeValue curTime;
    ParamDlg *uvGenDlg;
    int isActive;
    ...
}
```

Note that **CheckerDlg** has a data member that is a pointer to an instance of **Checker** (**theTex**). This is where it stores the 'thing', i.e. the current **Checker** being edited. Shown below are **Checker**'s implementations of **GetThing()** and **SetThing()**.

```
ReferenceTarget* GetThing() { return (ReferenceTarget *)theTex;
}
```

When the system calls **GetThing()**, **Checker** just returns the pointer to the item that is currently being edited.

```
void CheckerDlg::SetThing(ReferenceTarget *m) {
    assert (m->ClassID()==checkerClassID);
    assert (m->SuperClassID()==TEXMAP_CLASS_ID);
    if (theTex) theTex->paramDlg = NULL;
    theTex = (Checker *)m;
    uvGenDlg->SetThing(theTex->uvGen);
    if (theTex)
        theTex->paramDlg = this;
    LoadDialog(TRUE);
}
```

When the system calls **SetThing()**, **Checker** store the pointer to the item that is currently being edited into **theTex**. The **Checker** class itself maintains a pointer to the instance of **ParamDlg** that is handling the user interface. Note that **Checker** stores the **this** pointer (**theTex->paramDlg = this;**).

Also note that **CheckerDlg** maintains a pointer to a **UVGen** (**UVGenDlg**). **UVGen** is a class developers use to encapsulate the user interface for UV coordinates. This pointer is initialized in the **CheckerDlg** constructor as follows:

```
uvGenDlg = theTex->uvGen->CreateParamDlg(hwmedit, imp);
```

In the **CheckerDlg** implementation of **SetThing()** it also calls the **SetThing()** method on the **uvGenDlg**. This allows the **UVGen** to update its user interface in the dialog. These are the controls in the 'Coordinate' and 'Noise' rollups.

## Pre-defined Categories of Texture Maps

Developers creating texture maps should use the text strings shown below to distinguish between the various types of maps so they can be separated in the Material/Map Browser.

**TCHAR TEXMAP\_CAT\_2D[];** - 2D maps.

**TCHAR TEXMAP\_CAT\_3D[];** - 3D maps.

**TCHAR TEXMAP\_CAT\_COMP[];** - Composite.

**TCHAR TEXMAP\_CAT\_COLMOD[];** - Color modifier.

**TCHAR TEXMAP\_CAT\_ENV[];** - Environment.

The appropriate string should be returned by the **ClassDesc::Category()** method of the Texmap. For example:

```
const TCHAR* Category() { return TEXMAP_CAT_3D; }
```

See [Class ClassDesc](#) for more details on this method.

## Miscellaneous Function and Macros for use with Materials

The following functions are not part of any class but are available for use by plug-ins.

Developers that have created a 3D Studio/DOS SXP and a corresponding 3ds max texture plug-in may want to have a look at [Class Tex3D](#). It provides a way to have an instance of your 3ds max texture plug-in created automatically when the corresponding SXP is found in a 3DS file being imported (using the standard 3ds max 3DS importer).

Another handy materials related function is **CombineMaterials()**. This function combines the two specified materials into a multi-material.

### Prototype:

```
Mtl *CombineMaterials(Mtl *mat1, Mtl *mat2, int &mat2Offset);
```

### Remarks:

Implemented by the System.

This function is available in release 3.0 and later only.

This function combines the two specified materials into a multi-material.

Either of the two input materials can themselves be multi materials.

### Parameters:

**Mtl \*mat1**

Points to the one of the source materials.

**Mtl \*mat2**

Points to the other source material.

**int &mat2Offset**

The index of the first **mat2** material in the combined material is returned here.

### Return Value:

A pointer to the new multi-material.

The following function may be used to determine if the MtlBase pointer passed is a material (Mtl):

```
inline int IsMtl(MtlBase *m)  
{ return m->SuperClassID()==MATERIAL_CLASS_ID; }
```

The following function may be used to determine if the MtlBase pointer passed is a texture (Texmap):

```
inline int IsTex(MtlBase *m)
{ return m->SuperClassID()==TEXMAP_CLASS_ID; }
```

The following functions return the intensity of the color passed:

```
static inline float Intens(const AColor& c)
{ return (c.r+c.g+c.b)/3.0f; }
static inline float Intens(const Color& c)
{ return (c.r+c.g+c.b)/3.0f; }
```

The following functions return default instances of several classes implemented by the system:

```
UVGen* GetNewDefaultUVGen();
XYZGen* GetNewDefaultXYZGen();
TextureOutput* GetNewDefaultTextureOutput();
```

Note the following typedef's used with materials and textures:

```
typedef MtlBase* MtlBaseHandle;
typedef Mtl* MtlHandle;
typedef Texmap* TexmapHandle;
```

## Working with Meshes

See Also: [Class Mesh](#), [Class TriObject](#), [Class Face](#), [Class TVFace](#), [Class MNMesh](#).

## **Overview**

This topic presents information on working with meshes. The main classes dealing with meshes are discussed as well as several support classes. The way texture mapping works with meshes is presented. How materials are assigned to meshes is also reviewed.

## Overview of the Principal Classes

This section presents an overview of the principal classes used when working with meshes.

### [Class Mesh](#)

This is the main class for working with mesh objects. It has data members that point to the vertices, faces, texture vertices and texture faces. Methods are provided to access all properties of the mesh and to render, snap to, and hit test the mesh. There are also methods to optimize, apply mapping coordinates, and perform boolean operations on the mesh.

### [Class TriObject](#)

All procedural objects must be able to convert themselves to TriObjects. This is the class that actually flows down the geometry pipeline. This class contains an instance of the Mesh class.

### [Class Face](#)

This is the class used to hold a single triangular face of the mesh object. It maintains three indices into the vertex array of the mesh. Methods are provided for setting materials, smoothing groups, edge visibility and hidden status.

### [Class TVFace](#)

This is the class used to hold a texture face. It contains an array of three indices into the texture vertex array of the mesh.

## Support Classes

This section lists several classes that are handy when dealing with mesh objects. There are a set of classes for working with parts of the mesh such as its face structure, element structure, and cluster structure. For details see: [Class AdjEdgeList](#), [Class AdjFaceList](#), [Class FaceElementList](#), [Class FaceClusterList](#). New for release 2.0 and later, the **MNMesh** class is provided for temporary use by plug-ins, to help with complex topology-based modifications to Meshes. It has capabilities, such as the ability to recognize faces with more than 3 sides, that are useful in certain applications. See [Class MNMesh](#) for details.

## Extracting the Mesh from a Node

Developers who want to get the **TriObject** representation of a geometric object from its node can use the source code shown along with the method **Inode::EvalWorldState()**. See [Inode::EvalWorldState\(\)](#).

## **Building Meshes Suitable for MAX Modifiers**

Developers should follow a few simple but important rules when building meshes to function ideally with modifiers in the Geometry Pipeline. The six basic rules are:

- 1) Referencing each edge at most once in each direction.
- 2) Avoiding self-intersection of faces.
- 3) Avoiding creating faces with vertices located at the same place.
- 4) Not 'bridging' separate mesh components with a single vertex.
- 5) Breaking the mesh into sensible elements.
- 6) Whenever convenient, closing the mesh.

See the Advanced Topics section [Style Guidelines for Creating Pipeline-Friendly Meshes](#) for details on these rules.

## **Material Assignment**

Materials are assigned in 3ds max at the node level. There is one material assigned per node. The material itself defines the meaning of how it acts upon a mesh. For example, the 3ds max Multi/Sub-Object material uses the material ID assigned to each mesh face as indices into its list of sub-materials. In this way, the user may assign several materials to a single mesh object. The Mesh class provides methods to get and set the material ID assigned to a face. For more details see [Working with Materials and Textures](#).

## **Texture Vertices and Texture Faces**

The mesh class keeps a pointer to a list of texture vertices. If mapping coordinates are assigned to the mesh these extra vertices are allocated. These vertices are completely independent of the regular vertices in the mesh. In addition to the texture vertices there are also texture faces. There needs to be one texture face for every regular face in the mesh. Each texture face has three indices into the texture vertex array. This allows every face of the mesh to have its own mapping.

## Mapping Channels in Release 3.0 and Later

In the Materials Editor a user can choose which of the channels a texture map is applied to. In the Coordinates rollout of the user interface, in the Texture Mapping dropdown the user may choose "Explicit Map Channel". In this case the Map Channel spinner is enabled and the user may choose a number between 1 and 99. This number is effectively an index into the Mesh class mapping methods. The number 1 corresponds to the pre-R3 Mesh class **TVerts** array. The numbers 2 through 99 correspond to the R3 and later channels. If the user chooses "Vertex Color Channel" from the dropdown then the Map Channel spinner is disabled and the vertex color channel is used. This corresponds to an index of 0 in the Mesh class mapping methods.

In release 2.0 there were just two possible mapping channels. The original release 1.x mapping channel and the color per vertex channel. These two still exist. However the map channel index numbers have changed. Henceforth at the object level, map channel 0 refers to what was map channel 1, the vertex color channel, while map channel 1 refers to the original map channel. The reason for this change is that vertex colors are treated differently than map channel vertices for some topological operations, and it's better to separate this map channel from all the other map channels.

Note the following details on adding map channels:

- 1) To support a map channel in a mesh, call **Mesh::setMapSupport(channel, TRUE)**. This works for the vertex colors (channel 0) or the original TVFaces (channel 1) as well as the new channels.
- 2) The method **Mesh::setMapSupport** allocates the map faces, since there are always **Mesh::numFaces** of these, but it does not allocate the map verts, since the number of these can change from map to map.
- 3) Use **Mesh::setNumMapVerts(int mp)** to set the number of map verts.
- 4) Calls to **Mesh::setNumFaces(int nf, BOOL keep)** also set the number of map faces in all supported maps.
- 5) Keep in mind that the **Mesh::mapVerts(int mp)** and **Mesh::mapFaces(int mp)** methods return pointers that may be invalid later if the number of verts or faces is reallocated. For example, the following **won't** work:

```
UVVert *mv = mesh.mapVerts(43); // map channel 43  
mesh.setNumMapVerts(43, 98);  
for (i=0; i<98; i++) mv[i] = UVVert(0,0,0);
```

For reference information on the new mapping related methods in 3.0 see [Mesh Mapping Related Methods](#).

## Color Per Vertex Information

In 3ds max 2.0 and later color per vertex information is stored with a Mesh. This allows the user to assign colors to vertices. This ability is primarily for game developers and developers of radiosity renderers. This information can be used in conjunction with the Vertex Color map

(**MAXSDK\SAMPLES\MATERIALS\VERTCOL.CPP**) or the Color Per Vertex utility and modifier

(**MAXSDK\SAMPLES\UTILITIES\APPLYVC\APPLYVC.CPP** and **AVCMOD.CPP**).

Three public data members and several methods in the Mesh class provide access to this data.

**int numCVerts;**

The number of color vertices.

**VertColor \*vertCol;**

Array of color vertices. Note: **typedef Point3 VertColor;**

**TVFace \*vcFace;**

Array of vertex color faces.

## Color Per Vertex Enhancements in Release 4.0 and Later

In 3ds max 4.0 and later it is possible to have the source data for vertex colors come from other than the internal vertex color array (**vertCol**). The data can come from an external array or one of the map channels. When 3ds max is rendering the color values come from the **vertColArray** variable.

**VertColor \*vertColArray;**

This array defaults to the internal array (**vertCol**) but can be set to an external array or a mapping channel.

If an external array is used the following data member is a pointer to it (this defaults to NULL):

**VertColor \*curVCArray;**

If a mapping channel is used the following data member indicates which one (this defaults to 0).

**int curVCChan;**

When 3ds max is rendering the vertex lookup comes from the **vcFaceData** variable. This defaults to the **vcFace** data but if a mapping channel is used for color lookup 3ds max uses its TVFace structure.

**TVFace \*vcFaceData;**

The methods associates with this are as follows:

To set the number of vertex colors:

**BOOL setNumVertCol(int ct, BOOL keep=FALSE);**

To retrieve the number of vertex colors:

**int getNumVertCol() const**

To set the number of vertex color faces:

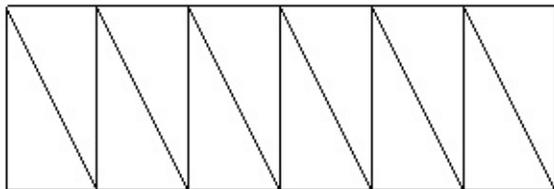
**BOOL setNumVCFaces(int ct, BOOL keep=FALSE, int oldCt=0);**

To use a different source array for vertex color data (this can be either an external array or one of the mapping channels):

**void setVCDisplayData(int mapChan = 0, VertColor \*VCArray=NULL, TVFace \*VCf=NULL);**

## Stripping

Stripping is the process of taking a mesh and turning it into all set of strips as shown below. Without stripping, when a triangle mesh is sent down the graphics display pipeline, three vertices plus three normals or colors must be sent for each triangle. However, if the triangles are turned into a 'strip', where one triangle points up, the next points down, then the next up, etc., forming parallel lines along the top and bottom (see diagram below), then for each new triangle in the strip, all that is sent down the graphics pipeline is the one new vertex. Since the communication between the CPU and the graphics card is one of the bottlenecks in the graphics pipeline this results in a significant speed increase.



This can only happen if 3ds max knows that each new triangle is adjoining the previous one. Stripping then, is the process of taking whatever configuration the mesh is in originally, and turning it into a sequence of sequence of vertices that all correspond to strips as shown above. This speeds up the display process dramatically.

Developers who create their own mesh objects for display need to be aware of two new methods for dealing with stripping if maximum speed is to be achieved. These method are **Mesh::BuildStrips()** and **Mesh::BuildStripsAndEdges()**. These builds the strip (and edge) databases inside the mesh. The standard 3ds max primitives call **BuildStripsAndEdges()** after creating their meshes for instance.

Changes have been made to the 3ds max geometry pipeline that make this important. In 3ds max 1.x, an algorithm was used to build the edge database when the mesh was about to be displayed. So for example, if a mesh was built, and some modifiers acted upon it, for instance a Bend, the geometry pipeline would be evaluated and then the when the object was about to be displayed its edge list would be built. If the modifiers were animated (say the Bend angle changed), the geometry pipeline would be evaluated and the edge list would be built again. This was very inefficient because the edge list would only get used for one display, and then would be thrown out, and then calculated again. Since a Bend modifier doesn't alter the mesh topology the edge list was actually still

valid.

In 3ds max 2.0 this changed. The edge and new strip topology flows down the pipeline. In this way, if a primitive builds its own edges and strips by calling **BuildStripsAndEdges()**, when animated modifiers are applied the display is dramatically faster. This is because the strip and edge database is never rebuilt unless the topology changes. Therefore when a developer creates a mesh to be displayed they should call **BuildStripsAndEdges()**. If this is not done, they won't be available to flow down the pipeline, and must be build automatically at the end of the pipeline.

If a developer knows that their strips are invalid, for example they've deleted a vertex or face, or otherwise changed the topology of the mesh, then the method **InvalidateStrips()** and **InvalidateEdgeList()** should be called. A developer could also call **InvalidateTopologyCache()** which simply calls both of the above.

However, if a modifier is written that changes topology (**ChannelsChanged()** includes **TOPO\_CHANNEL**), for example Bomb, then the invalidation happens automatically. Other objects or modifiers may have to make an explicit call to **InvalidateTopologyCache()**. For example, the Editable Mesh Object or the Edit Mesh Modifier can operate on a mesh at the push and pull vertex level rather than at the pipeline level. In these cases the mesh is being directly modified and thus the invalidate call is required.

## Working with NURBS

See Also: [Class NURBSObject](#), [Class NURBSSurface](#), [Class NURBSSet](#), [Class NURBSControlVertex](#), [Class NURBSPoint](#), [Class NURBSCurve](#), [Class NURBSFuseSurfaceCV](#), [Class NURBSFuseCurveCV](#), [Class TessApprox](#), [Class Object](#), [Class ShapeObject](#), [Class Matrix3](#), [Class Point3](#).

## Overview

This section presents an overview of working with NURBS using the SDK. It defines some terms and concepts related to NURBS, describes the main classes used, provides an overview of some key concepts when using NURBS, and discusses the sample code available in the SDK. There is also a section of reference information for some global functions available.

## The NURBS API

The NURBS API provides an interface into the NURBS objects used by MAX. Unlike much of the 3ds max API, the NURBS objects 3ds max provides don't use this API. Rather, the NURBS API has been added to allow developers to access the NURBS objects 3ds max uses internally. Using the API developer can create new NURBS objects or modify existing ones.

## ClassIDs of the NURBS Objects

Developers can check if an object is a NURBS object by checking it **ClassID()**. If it matches any of the following

```
#define EDITABLE_SURF_CLASS_ID Class_ID(0x76a11646,  
0x12a822fb)  
#define FITPOINT_PLANE_CLASS_ID Class_ID(0x76a11646,  
0xbadbeef)  
#define EDITABLE_CVCURVE_CLASS_ID  
Class_ID(0x76a11646, 0x12a82144)  
#define EDITABLE_FPCURVE_CLASS_ID  
Class_ID(0x76a11646, 0x12a82142)
```

then it's a NURBS object and the NURBS API may be used to manipulate it.

## Necessary Include File

The main NURBS include file is **\MAXSDK\INCLUDE\SURF\_API.H**. This is not included by default by **MAX.H** so you need to specifically include it via the following statement:

```
#include "surf_api.h"
```

## Reference Information on NURBS Mathematics

Developers who wish to understand the mathematics of NURBS should see: The NURBS Book: Les Piegl, Wayne Tiller, The NURBS Book (Berlin Heidelberg New York: Springer-Verlag, 1995). ISBN: 3-540-55069-0.

## **Providing Valid Data to the NURBS Methods**

The NURBS API is not intended to be bomb proof, i.e. it is possible to pass values to the methods than can cause a crash. It is up to the developer to insure that valid data is provided. As with the rest of the SDK it is easy to write a piece of code that will crash 3ds max if improper values are supplied.

## **Definition of Terms**

This section provides definitions of various terms used later in this topic and in the NURBS classes.

### **NURBS**

The term NURBS stands for Non-Uniform Rational B-Splines. Specifically: Non-Uniform means that the extent of a control vertex's influence can vary. This is useful when modeling irregular surfaces.

Rational means that the equation used to represent the curve or surface is expressed as a ratio of two polynomials, rather than a single summed polynomial. The rational equation provides a better model of some important curves and surfaces, especially conic sections, cones, spheres, and so on.

A B-spline (for basis spline) is a way to construct a curve that is interpolated between three or more points. Shape curves you create in 3ds max using the Line tool and other Shape tools are Bézier curves, which are a special case of B-splines.

### **Point**

A point in three-space. A Point Curve or Point Surface is constrained to pass through its points. Points behave somewhat like vertices for 3ds max spline objects, but their behavior is not identical and they are a distinct object type.

### **Curve**

This is a NURBS Curve. There are two kinds of NURBS curves in MAX. A Point Curve is controlled by points, which always lie on the curve. A CV Curve is controlled by control vertices (CVs), which don't necessarily lie on the curve.

### **CV**

This is a Control Vertex of a NURBS Curve or NURBS Surface. It's a vertex that controls a CV Curve or CV Surface. The 3D location of each CV affects the shape of the curve or surface. CVs aren't constrained to lie on the curve or surface. Each CV has a rational weight that can be used to adjust the influence of the CV on the curve's or surface's shape.

### **Point Curve**

A NURBS curve defined by points. The points are constrained to lie on the curve.

### **CV Curve**

A NURBS curve defined by CVs. The CVs don't necessarily lie on the curve. Instead, they form a control lattice that affects the curvature of the curve.

### **Surface**

This is an individual quadrilateral NURBS Surface. NURBS surfaces have essentially the same properties as NURBS curves, extended from a one-dimensional parameter space to two dimensions.

There are two kinds of NURBS surfaces: A Point Surface is controlled by points, which always lie on the surface. A CV Surface is controlled by control vertices (CVs). Instead of lying on the surface, CVs form a control lattice that surrounds the surface.

### **Point Surface**

A NURBS surface defined by points. The points are constrained to lie on the surface. More than one NURBS solution is possible for a Point Surface.

### **CV Surface**

A surface defined by CVs. Instead of lying on the surface, CVs form a control lattice that surrounds the surface.

### **Independent <object>**

This is an object (point, curve, surface) that is not dependent on any other object.

### **Dependent <object>**

This is an object (point, curve, surface) that depends on another object to define what it is. For example, a Blend Curve depends on the two curves that it blends between (as well as its own two tension parameters).

### **Constrained point**

Another term for a dependent point. A NURBS Point that is dependent on either another Point, Curve, or Surface, and that exists either on the object or relative to it. The relative cases are XYZ-relative, along a normal, or along a tangent (or set of tangents for a surface-dependent constrained point).

### **Continuity**

A curve is continuous if it is unbroken. There are different levels of continuity. A curve with an angle or cusp in it is C0 continuous<sup>3/4</sup>that is, the curve is continuous but has no derivative at the cusp. A curve with no such cusps but whose curvature changes is C1 continuous. Its derivative is also continuous, but its second derivative is not. A curve with uninterrupted, unchanging curvature is C2 continuous. Both its first and second derivatives

are also continuous.

A curve can have still higher levels of continuity, but for computer modeling these three are adequate. Usually the eye can't distinguish between a C2 continuous curve and one with higher continuity.

Continuity and degree are related. A degree 3 equation can generate a C2 continuous curve. This is why higher-degree curves aren't generally needed in NURBS modeling. Higher-degree curves are also less stable numerically, so using them isn't recommended.

Different segments of a NURBS curve can have different levels of continuity. In particular, by placing CVs at the same location or very close together, you reduce the continuity level. Two coincident CVs sharpen the curvature. Three coincident CVs create an angular cusp in the curve. This property of NURBS curves is known as multiplicity. In effect, the additional one or two CVs combine their influence in that vicinity of the curve.

By moving one of the CVs away from the other, you increase the curve's continuity level again. In MAX, multiplicity also applies when you fuse CVs. Fused CVs create a sharper curvature or a cusp in the curve. Again, the effect goes away if you unfuse the CVs and move one away from the other.

### **Multiplicity**

The property that coincident or nearly coincident CVs reduce the continuity level of the curve or surface. Two coincident CVs locally increase curvature. Three coincident CVs (or more) create an angular cusp. Fusing CVs shows the effect of multiplicity.

### **Degree**

The degree of a curve is highest exponent in the equation used to represent it. A linear equation is degree 1, a quadratic equation degree 2. NURBS curves typically are represented by cubic equations and have a degree of 3.

### **Order**

The order of a curve refers to its mathematical order. For instance a cubic curve is order 4, a quadratic curve is order 3, a linear curve is order 2. This is one more than the degree of polynomial of any segment of the curve.

### **Iso Line**

This is short for isoparametric line. It's a line of constant parameter value, similar to a contour line. Iso lines can be used to display a NURBS surface.

### **Segment**

The portion of a curve between two of its controlling points or CVs.

### **Knot**

This is a mathematical construct that helps define the span of control of CVs and blending functions that define NURBS Curves and Surfaces. The knots are an array of **double** precision values that determines the parameterization of a curve. . Values in the knot vector are nondecreasing. The knots specify the region of influence of the CVs on the curve. It is a way of partitioning the parameter space up into different segments. A B spline curve or a NURBS curve is a curve that is defined by a series of segments. On each one of the segments the curve is like a polynomial, or in the case of a rational one, it's like the ratio of polynomials. The knot vector describes how to partition the parameter space of the curve up for each of the different pieces of the polynomial.

### **Parameter space**

In addition to their existence in 3D space, NURBS objects have a parameter space that includes the array of knot values. NURBS curves have a single U dimension in parameter space. NURBS surfaces have two dimensions, UV, in parameter space.

### **Refine**

To increase the number of CVs on a curve or surface.

### **B-spline**

Short for basis spline. A kind of spline generated by so-called basis functions. The advantage of B-splines over Bézier curves (which are a special case of B-splines) is that the control vertices (CVs) of a B-spline affect only their local region of the curve or surface.

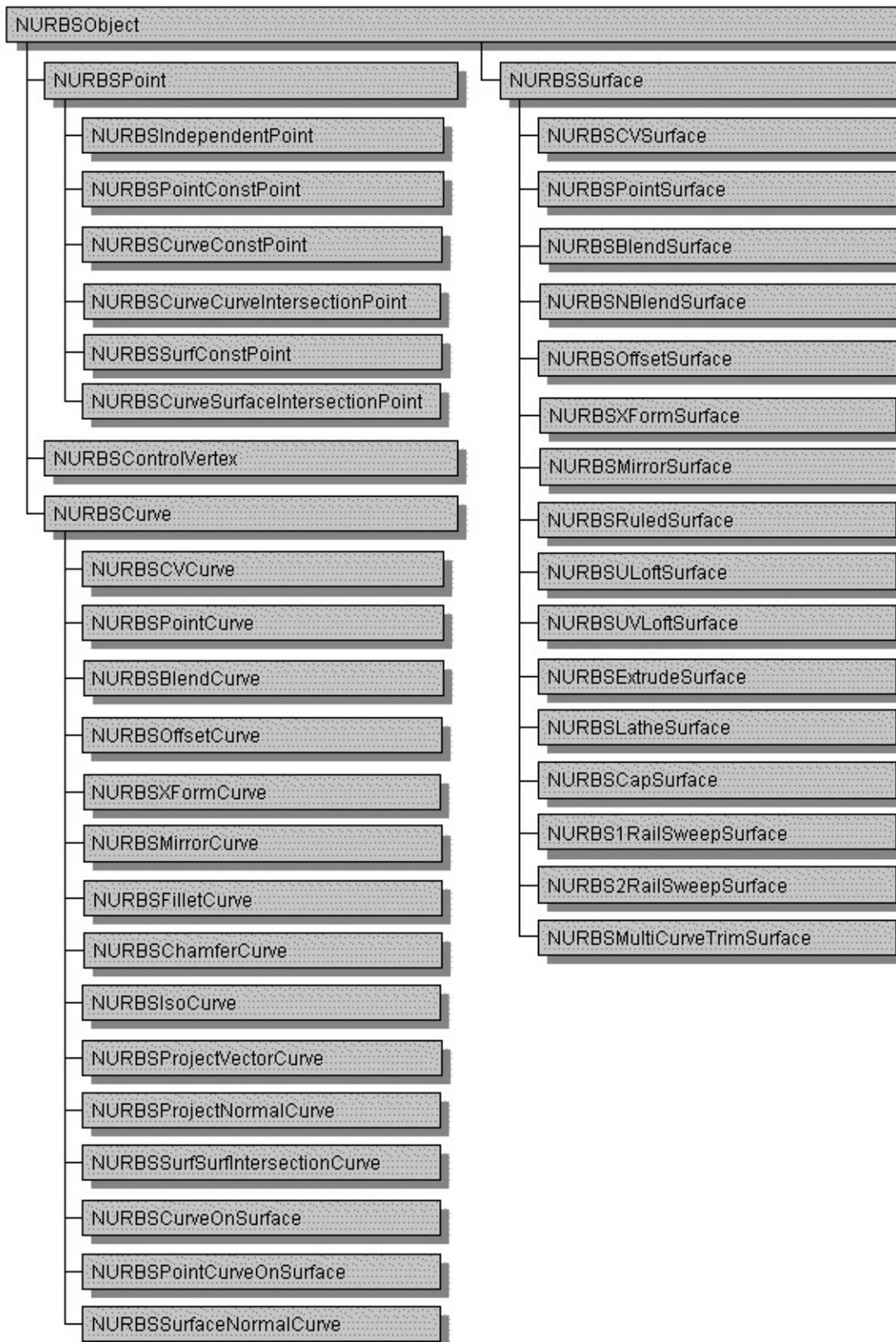
### **Bézier curve**

A curve modeled using a parametric polynomial technique. Bézier curves were developed by P. Bézier for computer modeling in automobile design. They are a special case of B-splines.

## Overview of the Principal Classes

The following diagram shows the inheritance tree of the principal classes in the NURBS API. The base classes are shown at the top, and the inheritance hierarchy proceeds toward the bottom and to the right. **NURBSObject** is the main base class, with **NURBSPoint**, **NURBSCurve** and **NURBSSurface** also functioning as base classes.

### Class Hierarchy



**The Base Class for NURBS Points, Curves and Surfaces:**

### [Class NURBSObject](#)

This is the base class for many of the other NURBS classes. It provides a set of methods that are common to the other classes such as getting and setting the name of the item, returning error messages, and determining the specific type of NURBS object. It also has a method to return a **NURBSId** which is an ID used to specify a particular object in communication between the NURBS classes.

### **Sets of NURBS Objects:**

#### [Class NURBSSet](#)

This is the class used when developers want to create 3ds max NURBS objects, or retrieve data from existing 3ds max NURBS objects. The **NURBSSet** acts as a container for the other objects.

This class contains a table of the various **NURBSObject** derived entities (points, curves, surfaces) used to make up the set. Additionally it has two 'fuse' tables: one for fuse curves and one for fuse surfaces. These are used to allow the CVs in the curves or surfaces to be 'stitched' or 'fused' together so that if one curve or surface moves the other moves with it. This class also has information required to tessellate the objects to triangle meshes for use in the viewports and the production renderer.

### **Independent Points:**

Independent Points are those that don't rely on other objects to define their location. This is different than the dependent (or constrained) points described later.

#### [Class NURBSPoint](#)

This class describes a point in 3 space. It is used as a base class for many of the dependent points in the API. It has methods to set and get the point position in various formats and floating point precisions. This is an abstract base class so only objects that are subclasses of this class can be created.

#### [Class NURBSIndependentPoint](#)

This class is derived from **NURBSPoint**. It is used to create an independent, free-standing point. There are methods to set the position of the point in various floating point formats and operators to compare points.

#### [Class NURBSControlVertex](#)

This is a Control Vertex of a NURBS Curve or NURBS Surface. This class

shares many of the same properties as a **NURBSPoint** and has the added property of a rational weight. The weight value of a CV is rational. That is, it is relative to other CVs in the curve or surface. Changing the weight of all CVs at once has no effect, because this doesn't change the ratio between weights.

#### [Class NURBSTrimPoint](#)

This class defines a point on a curve used to trim a portion of the curve from the point towards one of the ends of the curve.

### **Dependent (Constrained) Points:**

The following classes provide the ability to create dependent points. Dependent points are related to the objects they depend on such that the relationship established initially remains if the point, curve or surface moves. For example, a point can be created such that it always remains a certain distance from another point. The options are XYZ-relative, along a normal, or along a tangent (or set of tangents for a surface-dependent constrained point).

All the following classes are derived from **NURBSPoint**.

#### [Class NURBSPointConstPoint](#)

This class is used to create a dependent point that lies at a point or relative to it. It is a point constrained relative to another point. This can be a point used to define a point surface or point curve, it can be a trim point, or just a point in space.

#### [Class NURBSCurveConstPoint](#)

This class is used to create a dependent point that lies on a curve or relative to it.

#### [Class NURBSCurveCurveIntersectionPoint](#)

This class is used to create a dependent point at the intersection of two curves.

#### [Class NURBSSurfConstPoint](#)

This class is used to create a dependent point on a surface or related to it.

### **Independent Curves:**

The following classes provide the ability to create independent curves.

#### [Class NURBSCurve](#)

This base class describes the properties of a NURBS curve. This includes its open/closed state, and number of trim points. The **Evaluate()** method is

used to compute points and tangents on the curve.

#### [Class NURBSPointCurve](#)

This class is derived from **NURBSCurve**. It defines a curve that uses points to describe its shape. All the points lie on the curve itself. There are methods to get/set the number of points in the curve, get/set the points themselves, refine the curve (add points without changing its shape), and to get/set the transformation matrix used by the curve to position it within a NURBSSet.

#### [Class NURBSCVCurve](#)

This class is derived from **NURBSCurve**. It defines a curve that uses control vertices (CVs) to describe its shape. The CVs define a control lattice which surrounds the curve. This class has methods to close the curve, set its order, number of knots and number of CVs, and get/set the knots and CVs. There is also a method to add additional CVs to the curve.

### **Dependent Curves:**

The following classes are used to create dependent curves. A dependent curve is an object that depends on one or more other objects to define what it is. For example, a Blend Curve depends on the two parent curves that it blends between when it was created (as well as on its two tension parameters).

All the following classes are derived from **NURBSCurve**.

#### [Class NURBSBlendCurve](#)

This class defines a dependent blend curve. A blend curve connects the specified end of one curve to the specified end of another, blending the curvature of the parents to create a smooth curve between them.

#### [Class NURBSOffsetCurve](#)

This class defines a dependent offset curve. An offset curve is offset from the original, parent curve. It lies in the same plane as its parent, and is normal to the original.

#### [Class NURBSXFormCurve](#)

This class defines a dependent transform (xform) curve. A transform curve is a copy of the original curve with a different position, rotation, or scale

#### [Class NURBSMirrorCurve](#)

This class defines a dependent mirror curve. A mirror curve is similar to a mirror object that you create using the Mirror tool (on the 3ds max toolbar) or the Mirror modifier. It is the original curve reflected about one or two

axes.

#### [Class NURBSFilletCurve](#)

This class defines a dependent fillet curve. A fillet is a curve that creates a circular arc corner between two parent curves.

#### [Class NURBSChamferCurve](#)

This class defines a dependent chamfer curve. A chamfer is a curve that creates a straight line corner between two parent curves.

#### [Class NURBSIsoCurve](#)

This class defines a dependent iso curve. U and V iso curves are dependent curves created along lines of constant parameter value of a NURBS surface. Note the difference between "Iso Lines", which are a display artifact, and "Iso Curves" which are the dependent objects.

### **Independent Surface:**

#### [Class NURBSSurface](#)

This base class describes the properties of a NURBS surface. This includes its material ID, texture/tiling options, renderable state, and open/closed state, and normal inverted state. The **Evaluate()** method is used to compute points and tangents on the surface.

#### [Class NURBSPointSurface](#)

This class is derived from **NURBSSurface**. It defines a surface that uses points to describe its shape. This class has methods to close the surface in U and V, set the number of points in U and V, and get/set the points in U and V. There is also a method to add additional points to the surface. The point surface has a transformation matrix used to set the position of the surface within a **NURBSSet**.

#### [Class NURBSCVSurface](#)

This class is derived from **NURBSSurface**. It defines a surface that uses control vertices (CVs) to describe its shape. The CVs define a control lattice which surrounds the surface. This class has methods to close the surface in U and V, set its order in U and V, set the number of knots and CVs in U and V, and get/set the knots and CVs in U and V. There is also a method to add additional CVs to the surface. The CV surface has a transformation matrix used to position the surface within a **NURBSSet**.

### **Dependent Surfaces:**

Dependent surfaces are surface sub-objects whose geometry depends on other surfaces or curves in the NURBS model. When you change the geometry of the original parent surface or curve, the dependent surface changes as well.

#### [Class NURBSBlendSurface](#)

This class defines a dependent blend surface. A blend surface connects the edge of one surface to the edge of another, blending the curvature of the parents to create a smooth surface between them.

#### [Class NURBSOffsetSurface](#)

This class defines a dependent offset surface. An Offset surface is offset a specified distance from the original along the parent surface's normals.

#### [Class NURBSXFormSurface](#)

This class defines a dependent transform (xform) surface. A transform surface is a copy of the original surface with a different position, rotation, or scale

#### [Class NURBSMirrorSurface](#)

This class defines a dependent mirror surface. A mirror surface is similar to a mirror object that you create using the Mirror tool (on the 3ds max toolbar) or the Mirror modifier. It is the original surface reflected about one or two axes.

#### [Class NURBSRuledSurface](#)

This class defines a dependent ruled surface. A ruled surface is generated from two curve sub-objects. It lets you use curves to design the two opposite borders of a surface.

#### [Class NURBSULoftSurface](#)

This class defines a dependent U Loft surface. A U Loft surface interpolates a surface across multiple curve sub-objects. The curves become U-axis contours of the surface.

#### [Class NURBSExtrudeSurface](#)

This class defines a dependent extrude surface. An extrude surface is extruded from a curve sub-object. It is similar to a surface created with the Extrude modifier. The advantage is that an extrude sub-object is part of the NURBS model, so you can use it to construct other curve and surface sub-objects

#### [Class NURBSLatheSurface](#)

This class defines a dependent lathe surface. A lathe surface is generated from a curve sub-object. It is similar to a surface created with the Lathe modifier. The advantage is that a lathe sub-object is part of the NURBS model, so you can use it to construct other curve and surface sub-objects.

#### [Class NURBSCurveSurfaceIntersectionPoint](#)

This class is used to create a dependent point at the intersection of a curve and a surface.

#### [Class NURBSProjectNormalCurve](#)

This class provides access to the Normal Projected Curve. A Normal Projected curve lies on a surface. It is based on an existing curve, which is projected onto the surface in the direction of the surface's normals

#### [Class NURBSProjectVectorCurve](#)

This class provides access to the Vector Projected Curve. A Vector Projected curve lies on a surface. This is almost the same as a Normal Projected curve, except that the projection from the existing curve to the surface is in the direction of a vector that you can control. Vector projected curves may be used for trimming.

#### [Class NURBSSurfaceNormalCurve](#)

This provides access to the Surface Normal Curve. This is a curve created at a specified distance from a surface and normal to it.

#### [Class NURBSSurfSurfIntersectionCurve](#)

This class provides access to the Surface-Surface Intersection Curve. This is a curve that is defined by the intersection of two surfaces. You can use surface-surface intersection curves for trimming.

#### [Class NURBSCurveOnSurface](#)

This class provides access to the CV curve on surface parameters. These curves can be used for trimming the surface they lie on.

#### [Class NURBSPointCurveOnSurface](#)

This class provides access to the point curve on surface parameters. These curves can be used for trimming the surface they lie on.

#### [Class NURBSUVLoftSurface](#)

This class provides access to the UV Loft Surface. This surface is similar to the U Loft surface, but has a set of curves in the V dimension as well as the U dimension.

### [Class NURBSNBlendSurface](#)

This class provides access to the Multisided Blend surface. A Multisided Blend surface is a surface that "fills in" the edges defined by three or four other curve or surfaces. Unlike a regular, two-sided Blend surface, the curves or surfaces edges must form a closed loop<sup>¾</sup>that is, they must completely surround the opening the Multisided Blend will cover.

### [Class NURBS1RailSweepSurface](#)

This class provides access to the 1-Rail Sweep Surface. A 1-Rail Sweep Surface uses at least two curves. One curve, the "rail," defines one edge of the surface. The other curves define the surface's cross sections. The cross-section curves should intersect the rail curve. If the cross sections don't intersect the rail, the resulting surface is unpredictable.

### [Class NURBS2RailSweepSurface](#)

This class provides access to the 2-Rail Sweep Surface. A 2-Rail Sweep surface uses at least three curves. Two curves, the "rails," define the two edges of the surface. The other curves define the surface's cross sections. A 2-Rail Sweep surface is similar to a 1-Rail sweep. The additional rail gives you more control over the shape of the surface.

### [Class NURBSCapSurface](#)

This class provides access to the Cap Surface. A Cap Surface is a surface that caps a closed curve or the edge of a closed surface. Caps are especially useful with extruded surfaces.

### [Class NURBSMultiCurveTrimSurface](#)

This class provides access to the Multicurve Trim Surface which is a surface that is trimmed by multiple curves forming a loop.

## **Texture Mapping:**

The following classes are those involved in texture mapping NURBS surfaces. In 3ds max 3.0 and later there may be 99 texture channels assigned to each surface.

### [Class NURBSSurface](#)

This class (also described above) has a **NURBSTextureChannelSet** which is a table of pointers to all the mapping channels used by this surface. This class also has methods to return a reference to the **NURBSTextureSurface** associated with a specific channel.

### [Class NURBSTextureChannelSet](#)

This class holds a table of pointers to all the **NURBSTextureChannel** data for a surface. There are methods to return the table data by channel or by index and a method to add a new texture channel

#### [Class NURBSTextureChannel](#)

This class holds and provides access to a single texture channel. It holds the **NURBSTextureSurface** instance associated with the channel. It also holds the channel number (a number in the range 0 to 98), the tiling, offset, and angle parameters as well as the Generate UV flag.

#### [Class NURBSTextureSurface](#)

A texture surface is a surface associated with the surface sub-object. 3ds max uses the texture surface to control how materials are mapped. In effect, changing the texture surface stretches or otherwise changes the UV coordinates for the surface, altering the mapping. This class maintains a mapper type which controls how the texture surface generates the mapping. This class also provides access to the individual UV mapping coordinates. These coordinates are **NURBSTexturePoint** objects.

#### [Class NURBSTexturePoint](#)

This class holds and provides access to a single UV mapping coordinate.

## **NURBSIds and Indexes**

An important concept to understand when working with **NURBS**Sets and the other **NURBS** classes is the way you identify component items that are in the set. Before a **NURBS**Set is instantiated in the scene as a **NURBS** object, the individual points, curves, and surfaces are identified by an index number. As you append objects to the set they are assigned successive indexes, starting at 0. When you want to refer to a particular item, say to specify the parent surfaces in a blend surface, you use this index. Once an object has been added to a **NURBS**Set, its index can be retrieved using.

However, once a **NURBS**Set is associated with a scene object, either by being instantiated or because it was directly created from an existing scene object, this index is no longer valid. Instead, a different number, called a **NURBS**Id, is assigned to each item and you must use that Id when referring to items from then on. The **NURBS**Id of any item in an instantiated **NURBS**Set can be obtained through the **NURBS**Object::GetId() method of that object.

Seed Value Parameters

## **Seed Value Parameters**

Some kinds of dependent objects depend on geometry that might have more than one solution. For example, if you want to create a surface-curve intersection point but the curve intersects the surface more than once, 3ds max must decide which intersection is to be the location of the point.

For these kinds of objects, seed value parameters control the decision. The seed location is a location on a parent object, and the location nearest to the seed value that satisfies the creation condition is the one that 3ds max chooses.

For example, the seed location for a surface-curve intersection point is a U position along the length of the parent curve. The surface-curve intersection closest to the seed is chosen as the location of the dependent point.

For a surface, the seed location is a pair of UV coordinates in the surface's parameter space.

## Using the API to Create and Modify MAX NURBS Models

As mentioned above, 3ds max itself uses a different internal representation of NURBS objects than the API does. The API is simply a means to access these internal 3ds max NURBS objects and allow them to be manipulated.

The object used in communication between the internal 3ds max NURBS objects and the classes in the SDK available to work with these objects is the **NURBSSet**. There are also two global functions used in this communication. These are **CreateNURBSObject()** and **GetNURBSSet()**. How these are used is described below:

### Creating New NURBS Objects

The API can be used to create new NURBS objects. This is done by adding the objects (points, curves and surfaces) to an item called a NURBS 'set' and calling the global function **CreateNURBSObject()**. The **NURBSSet** is a container for the objects. Then the global function **CreateNURBSObject()** makes an internal 3ds max NURBS object out of the set.

As an example, consider the following code fragment that creates a **NURBSCVCurve** object (with **NURBSControlVertex** sub-objects) and puts it in the scene:

```
// Create an empty NURBSSet object  
NURBSSet nset;  
  
// Allocate a new NURBSCVCurve and set the knots and CVs  
NURBSCVCurve *c = new NURBSCVCurve();  
c->SetName(_T("CV Curve"));  
c->SetNumCVs(4);  
c->SetOrder(4);  
c->SetNumKnots(8);  
for (int k = 0; k < 4; k++) {  
  c->SetKnot(k, 0.0);  
  c->SetKnot(k+4, 1.0);  
}  
NURBSControlVertex cv;  
cv.SetPosition(0, Point3(0, 0, 50));  
c->SetCV(0, cv);
```

```

cv.SetPosition(0, Point3(-100, 0, 50));
c->SetCV(1, cv);
cv.SetPosition(0, Point3(-100, 100, 50));
c->SetCV(2, cv);
cv.SetPosition(0, Point3(0, 100, 50));
c->SetCV(3, cv);

// Add the NURBSCVCurve object to the set
nset.AppendObject(c);

// Create the NURBS object from the NURBSSet
Matrix3 mat(1);
Object *obj = CreateNURBSObject(ip, &nset, mat);

// Create the node in the scene and point it at the NURBS object
INode *node = ip->CreateObjectNode(obj);
node->SetName(_T("NURBS Set 1"));

```

The NURBS object created (**obj**) will have the Class\_ID:  
**EDITABLE\_SURF\_CLASS\_ID.**

### **Modifying Existing NURBS Objects**

The API can also be used to modify the properties of existing NURBS objects in the scene. To gain access to an internal 3ds max NURBS object you must have the data copied into a **NURBSSet**. This is done using the global function **GetNURBSSet()**. You can then use the methods of the **NURBSSet** to access the objects. After checking the type of the object you can cast it to the appropriate class and use that class's methods to alter the item.

For example, the following sample code attempts to grab the first NURBS sub-object from the first node in the current selection set. It then checks if it's a blend surface and if so adjust one of the tension parameters.

```

// Get the first selected node
INode* node = ip->GetSelNode(0);
if (!node) return;
// Get the object reference of the node
Object* obj = node->GetObjectRef();

```

```

// Make sure it's a NURBS object
if (obj->ClassID() == EDITABLE_SURF_CLASS_ID) {
    NURBSSet getSet;
    BOOL okay = GetNURBSSet(obj, ip->GetTime(), getSet,
    TRUE);
    if (okay) {
        NURBSObject *nObj;
        nObj = getSet.GetNURBSObject(0), ip->GetTime());
        if (nObj->GetType() == kNBlendSurface) {
            // It's a blend, adjust the tension
            NURBSBlendSurface *bSurf = (NURBSBlendSurface
            *)nObj;
            bSurf->SetTension(ip->GetTime(), 0, 2.0);
        }
    }
}

```

There are other global functions that may be used to modify NURBS objects. These functions are documented at the end of this topic. For instance, developers can use the function **BreakSurface()** to take an existing NURBS surface and break it into two separate surfaces.

## Accessing Details of NURBS Objects

Developers that need to access that internal of 3ds max NURBS objects can see the following methods to get at details such as knot counts, CV counts, order, weight and position data. The main call is:

```
BOOL GetNURBSSet(Object *object, TimeValue t, NURBSSet  
&nset, BOOL Relational);
```

with **Relational** set to **FALSE**. This will fill in the **NURBSSet** with the surfaces, curves and independent points of the object passed. The surfaces will all be **NURBSCVSurface**'s and the curves will all be **NURBSCVCurve**'s. These can be gotten from the **NURBSSet** with:

```
int GetNumObjects();  
NURBSObject* GetNURBSObject(int index);
```

The following can be called to get the knots (for surfaces, there is an equivalent for curves, of course):

```
int GetNumUKnots(void);  
int GetNumVKnots(void);  
double GetUKnot(int index);  
double GetVKnot(int index);
```

and the CVs can be retrieved with:

```
int GetNumUCVs(void);  
int GetNumVCVs(void);  
NURBSControlVertex* GetCV(int u, int v);
```

The order of the surface can be determined with:

```
int GetUOrder(void);  
int GetVOrder(void);
```

Finally, given a **NURBSControlVertex** the position and weights can be retrieved via member functions:

```
Point3 GetPosition(TimeValue t);  
void GetWeight(TimeValue t, double& wt);
```

## Parameter Ranges for Curves and Surfaces

Methods that deal with points in the parameter space of a curve work with arguments in U. Methods that deal with points in the parameter space of surfaces deal with arguments in U and V. For example, the function

**NURBSSurfConstPoint::SetUpParam()** sets the position of a dependent point in the parent surface's U parameter space. The valid U and V values that may be passed to this method must be obtained by calling

**NURBSSurface::GetParameterRange()** on the parent surface. This method retrieves the minimum and maximum values for U and V that may be used. For curves there is a similar function for getting the valid range for U. This is **NURBSCurve::GetParameterRange()**.

Developers should be aware that a curve or surface needs to be instantiated in the 3ds max data base before it is okay to call these methods (for example by calling **CreateNURBSObject()**). Prior to the object existing in the data base these calls will fail.

Generally, when CV curves and surfaces are created, the valid parameter range is known because they were specified in the beginning and ending knot values. In cases where they are not known, one can instantiate a **NURBSSet** that has just the base curves or surfaces. Once instantiated **GetParameterRange()** can be used. Then one can build the rest of the parametric model by making additions and/or modifications to the already instantiated object. The global function **AddNURBSObjects()** makes this easy to do.

## Materials Assignment and Texture Coordinates

### Materials and NURBS

Each NURBS surface has its own material ID. If you assign a Multi-SubObject material you can apply a different material to each surface in a NURBS object. The method to do this is **NURBSSurface::MatID()**. In release 3.0 and later each **NURBSCurve** can have an associated material. This is used by the Lathe and Extrude modifiers for example.

### Texture Coordinates and NURBS

Texture coordinates are assigned to each of the four corners of a NURBS surface. On the standard 3ds max primitives for example most are assigned to use (0,0) to (1,1) across the surface. Some primitive, such as the Prism, don't use (0,0) to (1,1) because of the way they wrap. Developers may of course assign any values using the API. The method to do this is **NURBSSurface::SetTextureUVs()**.

## Sample Code

There is a demonstration utility plug-in that exercises the NURBS API. This is available in

**\MAXSDK\SAMPLES\UTILITIES\NURBS\SDK\_TEST.CPP**. This is an excellent place to look for simple, clear sample code.

The standard 3ds max primitives also use the NURBS API. See this code in the various primitive objects in directory **\MAXSDK\SAMPLES\OBJECTS**.

Additionally, the Lathe and Extrude use the API. Developers can review this code in **\MAXSDK\SAMPLES\MODIFIERS\EXTRUDE.CPP** and **SURFREVS.CPP**.

## Additional Functions for Working with NURBS

The following global functions are available for working with NURBS. The hyperlinks below take you to the start of groups of related functions:

[Creating and Retrieving MAX NURBS Objects](#)

[Creating Primitive NURBS Objects \(Spheres, Cylinders, etc.\)](#)

[Modifying Existing NURBS Objects](#)

### Creating and Retrieving MAX NURBS Objects

#### Prototype:

```
Object *CreateNURBSObject(IObjParam* ip, NURBSSet *nset,  
Matrix3& mat);
```

#### Remarks:

This function is available in release 2.0 and later only.

This function takes a **NURBSSet** as input and outputs a pointer to an editable NURBS object. For example, this is what all the standard 3ds max primitives do when they implement **Object::ConvertToType()**. If you want to make a node in the scene reference a NURBS object, put the objects into a **NURBSSet** and use this function to create the NURBS object. Then pass this object to **Interface::CreateObjectNode()**.

#### Parameters:

##### **IObjParam\* ip**

The 3ds max interface pointer. If non-NULL, this is used to get at the 3ds max function to do unique naming. If a NULL is specified the names are not made unique.

##### **NURBSSet \*nset**

Points to the set of input objects to create. These are defined in object space. The matrix below is used to transform these. For instance, the **NURBSSet** could be defined in a unit cube and the **mat** parameter could be used to scale it up.

Note: The NURBSIds and parent NURBSIds in this NURBS set are filled in the process of doing the creation. Thus, you pass this set in, and the function modifies it so you can work with the set immediately (you don't have to later call **GetNURBSSet()** to get an appropriate one to work with). For example,

say you create a **NURBSCVSurface** using this API and then want to animate some of the CVs on the surface. You'd pass in this set, then when the object is created the CVs on the surface will have the parent ids filled in so you can actually manipulate the surface in the scene.

### **Matrix3& mat**

This is a transformation applied to all of the objects.

### **Return Value:**

A pointer to an editable NURBS object whose Class\_ID is **EDITABLE\_SURF\_CLASS\_ID**.

### **Prototype:**

**BOOL GetNURBSSet(Object \*object, TimeValue t, NURBSSet &nset, BOOL Relational);**

### **Remarks:**

This function is available in release 2.0 and later only.

This function is used to retrieve a **NURBSSet** that corresponds to the specified NURBS object. This allows a developer to access the internal objects inside a 3ds max editable NURBS object.

If the **Relational** parameter is FALSE the **NURBSSet** will contain CV curves and CV surfaces. So for example, if you pass an object that has a relational model (perhaps two CV surfaces and a dependent blend surface) it will decompose them into three CV surfaces. This will be the CV surfaces that represent the two surfaces and the blend, but you won't have the blend relational data. If **Relational** is TRUE, you'd get back two CV surfaces and a NURBS blend surface. If you get back a relational model, check the type of object using the base class method **NURBSObject::GetType()** and then cast the object to the appropriate type. Then you may call that classes methods on the object to work with it.

If you were using this API as part of an export plug-in, you'd probably want to set **Relational** to FALSE because getting back a relational blend surface would not be useful, but having CV surface data you could export would. On the other hand, if you simply wanted to animate the tension parameters on the blend surface you'd set **Relational** to TRUE and then call the methods of the blend surface object to change the tension parameters

**(NURBSBlendSurface::SetTension()).**

Important Note: Developers must use the method **NURBSSet::DeleteObjects()** when done with the **NURBSSet** to free the memory used.

**Parameters:**

**Object \*object**

The input object. This need to be a NURBS object (Class\_ID **EDITABLE\_SURF\_CLASS\_ID**).

**TimeValue t**

The time at which the object is evaluated and converted.

**NURBSSet &nset**

A NURBSSet that corresponds to the specified object is returned here.

**BOOL Relational**

If TRUE the **NURBSSet** will contain relational data; if FALSE the **NURBSSet** will contain a CV curve or CV surface equivalent of the object.

## **Creating primitive NURBS Objects (spheres, cylinders, etc.)**

**Prototype:**

```
NURBSResult GenNURBSLatheSurface(NURBSCurve& curve,  
Point3& origin, Point3& north, float start, float end,  
NURBSSurface& surf);
```

**Remarks:**

This function is available in release 2.0 and later only.

This function generates a NURBS surface of revolution given an input curve, origin, up axis, and start and end angles. The output of this function is a NURBS Surface. Note that this is a CV surface that matches the definition, not a relational surface.

**Parameters:**

**NURBSCurve& curve**

This is the NURBS curve that is revolved.

**Point3& origin**

Specifies the origin of the revolution.

**Point3& north**

This is the axis that specified the up direction.

**float start**

This is the start angle of the revolution in radians.

**float end**

This is the end angle of the revolution in radians.

**NURBSSurface& surf**

The surface definition is returned here.

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

```
NURBSResult GenNURBSSphereSurface(float radius, Point3&  
center, Point3& northAxis, Point3& refAxis, float startAngleU,  
float endAngleU, float startAngleV, float endAngleV,  
NURBSSurface& surf);
```

**Remarks:**

This function is available in release 2.0 and later only.

This function generates a NURBS sphere. The output of this function is a NURBS Surface. Note that this is a CV surface that matches the definition, not a relational surface.

Note: This surface is not closed surface.

**Parameters:****float radius**

The radius of the sphere.

**Point3& center**

The center point of origin of the sphere.

**Point3& northAxis**

This specifies the up axis. Use **Point3(0,0,1)** for the Z axis for example.

**Point3& refAxis**

This is the direction of the seam. The sphere primitive uses **Point3(0,-1,0)** as this value.

**float startAngleU**

The start angle for the sphere in the U direction, specified in radians.

**float endAngleU**

The end angle for the sphere in the U direction, specified in radians.

**float startAngleV**

The start angle for the sphere in the V direction, specified in radians.

**float endAngleV**

The end angle for the sphere in the V direction, specified in radians.

**NURBSSurface& surf**

The surface definition is returned here. This is a CV surface that matches the definition, not a relational surface.

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

**NURBSResult GenNURBSCylinderSurface(float radius, float height, Point3& origin, Point3& symAxis, Point3& refAxis, float startAngle, float endAngle, NURBSSurface& surf);**

**Remarks:**

This function is available in release 2.0 and later only.

This function generates a NURS cylinder. The output of this function is a NURBS Surface. Note that this is a CV surface that matches the definition, not a relational surface.

Note: This surface is not closed surface.

**Parameters:**

**float radius**

The radius of the cylinder.

**float height**

The height of the cylinder.

**Point3& origin**

The origin of the cylinder.

**Point3& symAxis**

The axis of symmetry. The standard cylinder primitive specifies **Point3(0,0,1)**.

**Point3& refAxis**

This is the direction of the seam. The standard cylinder primitive specifies **Point3(0,1,0)**.

**float startAngle**

The start angle in radians.

**float endAngle**

The end angle in radians.

**NURBSSurface& surf**

The surface definition is returned here.

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

**NURBSResult GenNURBSConeSurface(float radius1, float radius2, float height, Point3& origin, Point3& symAxis, Point3& refAxis, float startAngle, float endAngle, NURBSSurface& surf);**

**Remarks:**

This function is available in release 2.0 and later only.

This function generates a NURBS cone surface. The output of this function is a NURBS Surface. Note that this is a CV surface that matches the definition, not a relational surface.

Note: This surface is not closed surface.

**Parameters:**

**float radius1**

One of the radii of the cone.

**float radius2**

The other radius of the cone.

**float height**

The height of the cone.

**Point3& origin**

The origin of the cone.

**Point3& symAxis**

The axis of symmetry.

**Point3& refAxis**

This is the direction of the seam.

**float startAngle**

The start angle in radians.

**float endAngle**

The end angle in radians.

**NURBSSurface& surf**

The surface definition is returned here.

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

**NURBSResult GenNURBSTorusSurface(float majorRadius, float minorRadius, Point3& origin, Point3& symAxis, Point3& refAxis, float startAngleU, float endAngleU, float startAngleV, float endAngleV, NURBSSurface& surf);**

**Remarks:**

This function is available in release 2.0 and later only.

This function generates a NURBS torus surface. The output of this function is a NURBS Surface. Note that this is a CV surface that matches the definition, not a relational surface.

Note: This surface is not closed surface.

**Parameters:**

**float majorRadius**

This is the radius of the entire 'donut' shape.

**float minorRadius**

The is the radius of the 'tube'.

**Point3& origin**

The origin of the cone.

**Point3& symAxis**

The axis of symmetry.

**Point3& refAxis**

This is the direction of the seam.

**float startAngleU**

The start angle of the torus in the U direction.

**float endAngleU**

The end angle of the torus in the U direction.

**float startAngleV**

The start angle of the torus in the V direction.

**float endAngleV**

The end angle of the torus in the V direction.

**NURBSurface& surf**

The surface definition is returned here.

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

**Object \*CreateNURBSLatheShape(IObjParam\* ip, TSTR name, TimeValue t, ShapeObject \*shape, Matrix3& axis, float degrees, int capStart, int capEnd, int capType, BOOL weldCore, BOOL flipNormals, BOOL texturing, int segs, BOOL matIds, BOOL shapeIDs);**

**Remarks:**

This function is available in release 2.0 and later only.

This function generates a NURBS object based on the specified lathe (surface of revolution) definition and returns a pointer to it. This is used by the lathe modifier.

**Parameters:**

**IObjParam\* ip**

The 3ds max interface pointer. If non-NULL, this is used to get at the 3ds max function to do unique naming. If a NULL is specified the names are not made unique.

**TSTR name**

The name for the object. If the pointer above is non-NULL this name is made unique.

**TimeValue t**

The time at which to revolve the shape.

**ShapeObject \*shape**

Points to the shape object to revolve. Note that if the **ShapeObject** pointed to is a bezier spline then capping won't work properly.

**Matrix3& axis**

This specifies the axis of revolution.

**float degrees**

The angle for the surface of revolution in degrees.

**int capStart**

Specifies if the surface should be capped at the beginning: TRUE to cap; FALSE to leave open. Note that this is only used if the ShapeObject is a NURBS curve.

**int capEnd**

Specifies if the surface should be capped at the ending: TRUE to cap; FALSE to leave open. Note that this is only used if the ShapeObject is a NURBS curve.

**int capType**

This parameter is not currently used and the value passed is ignored.

**BOOL weldCore**

TRUE to collapse any coincident vertices at the center of the surface; otherwise FALSE.

**BOOL flipNormals**

TRUE to invert the orientation of surface normals; otherwise FALSE.

**BOOL texturing**

TRUE to generate mapping coordinates; otherwise FALSE.

**int segs**

This parameter is not currently used and the value passed is ignored.

**BOOL matIds**

If TRUE special material Ids are assigned to the surfaces and caps.

**BOOL shapeIDs**

This parameter is available in release 3.0 and later only.

If TRUE shape IDs are used. When on, this function uses the material ID values assigned to segments in the spline to be lathed, or curve sub-objects in the NURBS curve to be lathed. This is available only when **matIds** is TRUE.

**Prototype:**

**Object \*CreateNURBSExtrudeShape(IObjParam\* ip, TSTR name, TimeValue t, ShapeObject \*shape, float amount, int capStart, int capEnd, int capType, BOOL texturing, BOOL matIDs, BOOL shapeIDs);**

**Remarks:**

This function is available in release 2.0 and later only.

This function generates a NURBS object based on the specified extrusion definition and returns a pointer to it. This is used by the extrude modifier.

**Parameters:****IObjParam\* ip**

The 3ds max interface pointer. If non-NULL, this is used to get at the 3ds max function to do unique naming. If a NULL is specified the names are not made unique.

**TSTR name**

The name for the surface. If the pointer above is non-NULL this name is made unique.

**TimeValue t**

The time at which to extrude the shape.

**ShapeObject \*shape**

Points to the shape object to extrude. Note that if the **ShapeObject** pointed to is a bezier spline then capping won't work properly.

**float amount**

Specifies the height of the extrusion.

**int capStart**

Specifies if the surface should be capped at the bottom: TRUE to cap; FALSE to leave open. Note that this is only used if the ShapeObject is a NURBS curve.

**int capEnd**

Specifies if the surface should be capped at the top: TRUE to cap; FALSE to leave open. Note that this is only used if the ShapeObject is a NURBS curve.

**int capType**

This parameter is not currently used and the value passed is ignored.

### **BOOL texturing**

If TRUE texture coordinates are assigned; otherwise they are not.

### **BOOL matIds**

If TRUE special material Ids are assigned to the surfaces and caps.

### **BOOL shapeIDs**

This parameter is available in release 3.0 and later only.

If TRUE shape IDs are used. When on, this function uses the material ID values assigned to segments in the spline to be extruded, or curve sub-objects in the NURBS curve to be extruded. This is available only when **matIds** is TRUE.

## **Modifying Existing NURBS Objects**

### **Prototype:**

```
int AddNURBSObjects(Object* obj, IObjParam* ip, NURBSSet *nset);
```

### **Remarks:**

This function is available in release 2.0 and later only.

This function adds the specified **NURBSSet** to the specified object.

### **Parameters:**

#### **Object\* obj**

Points to the object to have the specified **NURBSSet** added to. This object must be a NURBS object (otherwise it will simply return 0).

#### **IObjParam\* ip**

The 3ds max interface pointer. If non-NULL, this is used to get at the 3ds max function to do unique naming. If a NULL is specified the names are not made unique.

#### **NURBSSet \*nset**

Points to the NURBS set that is added to the object.

### **Return Value:**

Nonzero if the object was added; otherwise zero.

### **Prototype:**

**NURBSResult SetSurfaceApprox(Object\* obj, BOOL viewport, TessApprox \*tess, BOOL clearSurfs=FALSE);**

**Remarks:**

This function is available in release 3.0 and later only.

This function sets the surface approximation characteristics of an existing NURBS object.

**Parameters:**

**Object\* obj**

The NURBS object to modify.

**BOOL viewport**

TRUE to set the viewport approximation settings; FALSE to set the rendering settings.

**TessApprox \*tess**

Specifies the properties of the tessellation approximation to the mathematical surface.

**BOOL clearSurfs=FALSE**

This tells the NURBS object to clear any over-ridden values on individual surfaces. The NURBS UI lets users set surface approximation value on individual surfaces that over-ride the values for the whole object. When this is TRUE, then these are cleared out.

**Return Value:**

See [List of NURBS Results](#).

**Function:**

**NURBSResult SetCurveApprox(Object\* obj, BOOL viewport, TessApprox \*tess, BOOL clearSurfs);**

**Remarks:**

This function is available in release 3.0 and later only.

This function sets the curve approximation characteristics of an existing NURBS object.

**Parameters:**

**Object\* obj**

The NURBS object to modify.

**BOOL viewport**

TRUE to set the viewport approximation settings; FALSE to set the rendering settings.

**TessApprox \*tess**

Specifies the properties of the tessellation approximation to the mathematical curve.

**BOOL clearSurfs**

This tells the NURBS object to clear any over-ridden values on individual curves. The NURBS UI lets users set surface approximation value on individual surfaces that over-ride the values for the whole object. When this is TRUE, then these are cleared out.

**Return Value:**

See [List of NURBS Results](#).

**Function:**

**NURBSResult SetDispApprox(Object\* obj, TessApprox \*tess, BOOL clearSurfs);**

**Remarks:**

This function is available in release 3.0 and later only.

This function sets the displacement mapping approximation characteristics of an existing NURBS object.

**Parameters:****Object\* obj**

The NURBS object to modify.

**TessApprox \*tess**

Specifies the properties of the tessellation approximation to the mathematical curve.

**BOOL clearSurfs**

This tells the NURBS object to clear any over-ridden values on individual surfaces. The NURBS UI lets users set surface approximation value on individual surfaces that over-ride the values for the whole object. When this is TRUE, then these are cleared out.

**Return Value:**

See [List of NURBS Results](#).

**Function:**

**NURBSResult SetSurfaceDisplaySettings(Object\* obj,  
NURBSDisplay& disp);**

**Remarks:**

This function is available in release 3.0 and later only.

This function sets the display properties of an existing NURBS object.

**Parameters:**

**Object\* obj**

The NURBS object to modify.

**NURBSDisplay& disp**

The display properties to use. See [Class NURBSDisplay](#).

**Return Value:**

See [List of NURBS Results](#).

**Function:**

**NURBSResult GetSurfaceDisplaySettings(Object\* obj,  
NURBSDisplay& disp);**

**Remarks:**

This function is available in release 3.0 and later only.

This function retrieves the display properties of an existing NURBS object.

**Parameters:**

**Object\* obj**

The NURBS object to check.

**NURBSDisplay& disp**

The display properties are returned here. See [Class NURBSDisplay](#).

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

**NURBSResult Transform(Object\* obj, NURBSIdTab& ids,  
SetXFormPacket& xPack, Matrix3& mat, TimeValue t);**

**Remarks:**

This function is available in release 2.0 and later only.

This function transforms the specified sub-objects within the NURBS object passed. Note that the parameter **xPack** is used for the transformation when Animate mode is on, and **mat** is used when Animate is off. This stems from the fact that 3ds max delays creation of controllers on NURBS sub-objects until they are animated.

**Parameters:****Object\* obj**

The NURBS object whose sub-objects are modified.

**NURBSIdTab& ids**

The table of NURBS Ids of the sub-objects to transform. These are the sub-objects within the 3ds max NURBS object **obj** that are transformed.

**SetXFormPacket& xPack**

When Animate mode is on, this matrix is used for the transformation. See [Class SetXFormPacket](#).

**Matrix3 mat**

When not in Animate mode, this matrix is used for the transformation.

**TimeValue t**

The time at which to do the transformation.

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

```
NURBSResult BreakCurve(Object* obj, NURBSId id, double u,  
TimeValue t);
```

**Remarks:**

This function is available in release 2.0 and later only.

This function takes the specified NURBS curve sub-object and generates two curves from it.

**Parameters:****Object\* obj**

The NURBS object whose curve sub-object is broken.

**NURBSId id**

The id of the curve sub-object to break.

**double u**

The point along the curve at which the break is made.

**TimeValue t**

The time at which the curve is broken. The **u** parameter is relative to the state of the curve at this time (for instance if it's animated).

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

```
NURBSResult BreakSurface(Object* obj, NURBSId id, BOOL  
breakU, double param, TimeValue t);
```

**Remarks:**

This function is available in release 2.0 and later only.

This function takes the specified NURBS surface sub-object and generates two surfaces from it.

**Parameters:****Object\* obj**

The NURBS object whose surface sub-object is broken.

**NURBSId id**

The id of the surface sub-object to break.

**BOOL breakU**

If TRUE the break occurs along a constant U parameter; otherwise along a constant V parameter.

**double param**

The distance in U or V space that the break should happen in.

**TimeValue t**

The time at which the surface is broken. The **u** parameter is relative to the state of the surface at this time (for instance if it's animated).

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

**NURBSId JoinCurves(Object\* obj, NURBSId id1, NURBSId id2, BOOL begin1, BOOL begin2, double tolerance, double ten1, double ten2, TimeValue t);**

**Remarks:**

This function is available in release 3.0 and later only.

This function takes two curves and joins them together and makes a single curve out of them. That is, the endpoints of the two curve objects are connected by new segments, and the two original curves are replaced by a single curve.

**Parameters:****Object\* obj**

The NURBS object whose curve sub-objects are joined.

**NURBSId id1**

The id of the first curve sub-object.

**NURBSId id2**

The id of the second curve sub-object.

**BOOL begin1**

Specifies if the beginning or the end of curve 1 is joined. If TRUE the beginning is used; otherwise the end is used. The beginning of the curve is the point of minimum parameter value.

**BOOL begin2**

Specifies if the beginning or the end of curve 2 is joined. If TRUE the beginning is used; otherwise the end is used. The beginning of the curve is the point of minimum parameter value.

**double tolerance**

A distance in 3ds max units. If the gap between the curves you are joining is greater than this value, this function creates the join by first creating a blend curve and then joining the three parts. If the gap is less than this value, or if the curves are overlapping or coincident, no blend is created.

Creating a blend and then joining the three curves into a single curve is the better technique. The result matches the parent curves well. Without the blend step, the resulting curve can deviate from the parent curves, in order to maintain smoothness. (The amount of deviation depends on how far from

tangent the two input curves were at the join.)

A problem arises when there is a gap but it is too small. In this case, this function generates the blend but because there isn't enough room for it, the resulting curve has a loop in it. To avoid this loop, set this parameter higher than the gap distance.

If you use a value of 0.0, the function chooses a value to use for the tolerance.

**double ten1**

Specifies the tension of the new curve at the end of the first curve sub-object.

**double ten2**

Specifies the tension of the new curve at the end of the second curve sub-object.

**TimeValue t**

The time at which the curves are joined.

**Return Value:**

The NURBSId of the newly joined curve.

**Prototype:**

**NURBSId JoinSurfaces(Object\* obj, NURBSId id1, NURBSId id2, int edge1, int edge2, double tolerance, double ten1, double ten2, TimeValue t);**

**Remarks:**

This function is available in release 2.0 and later only.

This function takes two surfaces and joins them together and makes a single surface out of them. That is, the specified edges of the two surface objects are connected by a new surface, and the two original surfaces are replaced by a single surface.

**Parameters:**

**Object\* obj**

The NURBS object whose surface sub-objects are joined.

**NURBSId id1**

The id of the first surface sub-object.

**NURBSId id2**

The id of the second surface sub-object.

**int edge1**

The index of the edge of surface 1 to join.

**int edge2**

The index of the edge of surface 2 to join.

**double tolerance**

A distance in 3ds max units. If the gap between the surfaces you are joining is greater than this value, this function creates the join by first creating a blend surface. If the gap is less than this value, or if the surfaces are overlapping or coincident, 3ds max doesn't create the blend.

Creating a blend and then joining the three surfaces into a single surface is the better technique. The result matches the parent surfaces well. Without the blend step, the resulting surface can deviate from the parent surfaces, in order to maintain smoothness. (The amount of deviation depends on how far from tangent the two input surfaces were at the join.)

A problem arises when there is a gap but it is too small. In this case, this function generates the blend but because there isn't enough room for it, the resulting surface has a loop in it. To avoid this loop, set the tolerance parameter higher than the gap distance.

If you set the tolerance to 0.0, the function chooses a value to use for the tolerance.

**double ten1**

Specifies the tension of the new surface at the edge of the first surface sub-object.

**double ten2**

Specifies the tension of the new surface at the edge of the second surface sub-object.

**TimeValue t**

The time at which the surfaces are joined.

**Return Value:**

The NURBSId of the newly joined surface.

**Function:**

```
NURBSResult ZipCurves(Object* obj, NURBSId id1, NURBSId id2, BOOL begin1, BOOL begin2, double tolerance, TimeValue t);
```

**Remarks:**

This function is available in release 3.0 and later only.

This method 'zips' two curves. Zipping concatenates the CV lattices of the two original curves. Zipping can change the shape of the original curves, but usually it produces a better result than joining. If both curves are untrimmed point curves, the result is a point curve. In all other cases, the result of zipping is a CV curve.

**Parameters:**

The NURBS object whose curve sub-objects are zipped.

**NURBSId id1**

The id of the first curve sub-object.

**NURBSId id2**

The id of the second curve sub-object.

**BOOL begin1**

Specifies if the beginning or the end of curve 1 is joined. If TRUE the beginning is used; otherwise the end is used. The beginning of the curve is the point of minimum parameter value.

**BOOL begin2**

Specifies if the beginning or the end of curve 2 is joined. If TRUE the beginning is used; otherwise the end is used. The beginning of the curve is the point of minimum parameter value.

**double tolerance**

A distance in 3ds max units. If the gap between the curves you are zipping is greater than this value, this function creates the join by first creating a blend curve and then joining the three parts. If the gap is less than this value, or if the curves are overlapping or coincident, no blend is created.

If you use a value of 0.0, the function chooses a value to use for the tolerance.

**TimeValue t**

The time at which the surfaces are joined.

**Return Value:**

See [List of NURBS Results](#).

**Function:**

**NURBSResult ZipSurfaces(Object\* obj, NURBSId id1, NURBSId**

**id2, int edge1, int edge2, double tolerance, TimeValue t);**

**Remarks:**

This function is available in release 3.0 and later only.

This method 'zips' two surface. Zipping concatenates the CV lattices of the two original surfaces. Zipping can change the shape of the original surfaces, but compared to joining it usually produces a simpler surface that is easier to edit.

**Parameters:**

**Object\* obj**

The NURBS object whose surface sub-objects are zipped.

**NURBSId id1**

The id of the first surface sub-object.

**NURBSId id2**

The id of the second surface sub-object.

**int edge1**

The index of the edge of surface 1 to join.

**int edge2**

The index of the edge of surface 2 to join.

**double tolerance**

A distance in 3ds max units. If the gap between the surfaces you are zipping is greater than this value, this function creates the join by first creating a blend surface. If the gap is less than this value, or if the surfaces are overlapping or coincident, 3ds max doesn't create the blend.

If you set the tolerance to 0.0, the function chooses a value to use for the tolerance.

**double ten1**

Specifies the tension of the new surface at the edge of the first surface sub-object.

**double ten2**

Specifies the tension of the new surface at the edge of the second surface sub-object.

**TimeValue t**

The time at which the surfaces are joined.

**Return Value:**

See [List of NURBS Results](#).

**Prototype:**

**NURBSId MakeIndependent(Object\* obj, NURBSId id, TimeValue t);**

**Remarks:**

This function is available in release 2.0 and later only.

This function takes a dependent sub-object (fillet, offset, blend, etc.) and turns it into a CV variant of itself such that it is independent (no longer dependent on a parent).

**Parameters:****Object\* obj**

The NURBS object whose sub-object is made independent.

**NURBSId id**

The id of the dependent sub-object.

**TimeValue t**

The time at which the sub-object is made independent.

**Return Value:**

The NURBSId of the resulting object or zero if it could not be done.

**Function:**

**NURBSId MakeRigid(Object\* obj, NURBSId id, TimeValue t);**

**Remarks:**

This function is available in release 3.0 and later only.

To improve performance, you can make any kind of surface sub-object into a rigid surface. Once you have made a surface rigid, the only editing allowed is to transform it at the Surface sub-object level. You can't move a rigid surface's points or CVs, or change the number of points or CVs.

Rigid surfaces reduce the amount of memory used by the NURBS model.

Making surfaces rigid improves performance, especially for large and complex models.

**Parameters:**

**Object\* obj**

The NURBS object whose surface sub-objects is made rigid.

**NURBSId id**

The id of the sub-object who is made rigid.

**TimeValue t**

The time at which to make it rigid.

**Return Value:**

The NURBSId of the rigid surface.

**Function:**

```
void SetApproxPreset(Object* pObj, int i);
```

**Remarks:**

This function is available in release 3.0 and later only.

This allows choosing one of the Tessellation Presets which appear in the 3ds max UI in the Surface Approximation rollup. These are the Low, Medium and High buttons. See **GetTessPreset()** and **SetTessPreset()** below.

**Parameters:****Object\* pObj**

The objects whose specified Tessellation Preset is set.

**int i**

Specifies which preset to make active. One of the following values:

**0**: Low

**1**: Medium

**2**: High

**Function:**

```
void ToggleShadedLattice(Object* pObj);
```

**Remarks:**

This function is available in release 3.0 and later only.

Inverts the state of the Shaded Lattice setting for the object. This goes between 'Tesselated Mesh' and 'Shaded Lattice' surface display mode.

**Parameters:**

## **Object\* pObj**

The object whose Shaded Lattice setting is toggled.

### **Function:**

**TessApprox\* GetTessPreset(int which, int preset);**

### **Remarks:**

This function is available in release 3.0 and later only.

Returns a pointer to the tessellation object corresponding to the specified type of tessellation preset.

### **Parameters:**

#### **int which**

Determines which type of tessellation is set. One of the following values:

- 0:** Curve tessellation in the viewports.
- 1:** Curve tessellation by the production renderer.
- 2:** Surface tessellation in the viewports.
- 3:** Surface tessellation by the production renderer.
- 4:** Displacement mapping (by the production renderer).

#### **int preset**

Specifies which preset to get. One of the following values:

- 0:** Low
- 1:** Medium
- 2:** High

### **Return Value:**

See [Class TessApprox](#).

### **Function:**

**void SetTessPreset(int which, int preset, TessApprox& tess);**

### **Remarks:**

This function is available in release 3.0 and later only.

Sets the tessellation object corresponding to the specified type of tessellation preset.

**Parameters:****int which**

Determines which type of tessellation is set. One of the following values:

- 0:** Curve tessellation in the viewports.
- 1:** Curve tessellation by the production renderer.
- 2:** Surface tessellation in the viewports.
- 3:** Surface tessellation by the production renderer.
- 4:** Displacement mapping (by the production renderer).

**int preset**

Specifies which preset to get. One of the following values:

- 0:** Low
- 1:** Medium
- 2:** High

**TessApprox & tess**

The tessellation object to set. See [Class TessApprox](#).

**Function:**

**Object \*BuildEMObjectFromLofterObject(Object \*loftObject, double tolerance);**

**Remarks:**

This function is available in release 3.0 and later only.

Generates a new NURBS object from the specified loft object at the current time. This is a special call used internally in the lofter to convert a loft object to a NURBS surface.

**Parameters:****Object \*loftObject**

Points to the source loft object.

**double tolerance**

This parameter is the maximum deviation of the NURBS approximation to the true loft surface.

**Return Value:**

A pointer to the new NURBS object.

**Function:**

**Object \*BuildEMObjectFromPatchObject(Object \*patchObject);**

**Remarks:**

This function is available in release 3.0 and later only.

Generates a new NURBS object from the specified patch object at the current time.

**Parameters:**

**Object \*patchObject**

Points to the source patch object.

**Return Value:**

A pointer to the new NURBS object.

**Function:**

**Object \*DetachObjects(TimeValue t, INode \*pNode, Object\* pObj, NURBSIdList list, char \*newObjName, BOOL copy, BOOL relational);**

**Remarks:**

This global function is available in release 3.0 and later only.

Creates a NURBS object in the scene named **newObjName** respecting the copy and relational flags like the 3ds max Detach operation.

**Parameters:**

**TimeValue t**

The time at which to detach the object.

**INode \*pNode**

The node of the existing object.

**Object\* pObj**

Points to the NURBS object whose sub-objects are detached.

**NURBSIdList list**

The list of objects to detach. Note: **typedef Tab<NURBSId> NURBSIdList**. See [Template Class Tab](#).

**char \*newObjName**

The name for the new node created.

**BOOL copy**

TRUE to create a copy for the detached objects leaving the original; FALSE to remove the originals.

**BOOL relational**

If TRUE the object will contain relational data; if FALSE the object will contain a CV curve or CV surface equivalent of the object.

**Return Value:**

A pointer to the new object created by the detach. If an error occurs, NULL is returned.

**Function:**

**NURBSSubObjectLevel GetSelectionLevel(Object\* pObj);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns the current sub-object selection level for the specified NURBS object.

**Parameters:**

**Object\* pObj**

The object to check.

**Return Value:**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**Function:**

**NURBSResult SetSelectionLevel(Object\* pObj,  
NURBSSubObjectLevel level);**

**Remarks:**

This global function is available in release 3.0 and later only.

Sets the current sub-object selection level for the specified NURBS object.

**Parameters:**

**Object\* pObj**

Points to the object whose sub-object selection state is set.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

**NURBSResult GetSelection(Object\* pObj,  
NURBSSubObjectLevel level, BitArray& selset);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns a BitArray containing the sub-object selection state of the NURBS object passed.

**Parameters:**

**Object\* pObj**

Points to the object whose sub-object selection data is retrieved.

**NURBSSubObjectLevel level**

The sub-object level data to get. See [List of NURBSSubObjectLevel Options](#).

**BitArray& selset**

The BitArray for the result. Bits set are selected components. See [Class BitArray](#).

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

**NURBSResult SetSelection(Object\* pObj, NURBSSubObjectLevel  
level, BitArray& selset);**

**Remarks:**

This global function is available in release 3.0 and later only.

Sets the specified sub-object selection state of the NURBS object passed.

**Parameters:**

**Object\* pObj**

Points to the object whose sub-object selection state is set.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

## **BitArray& serset**

The BitArray with the selection data. Bits which are set indicate items to select.

### **Return Value:**

See [List of NURBS Results](#) .

### **Function:**

**NURBSResult MoveCurrentSelection(Object\* pObj,  
NURBSSubObjectLevel level, TimeValue t, Matrix3& partm,  
Matrix3& tmAxis, Point3& val, BOOL localOrigin);**

### **Remarks:**

This global function is available in release 3.0 and later only.  
Moves the current sub-object selection at the specified level.

### **Parameters:**

#### **Object\* pObj**

The object whose components are transformed.

#### **NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

#### **TimeValue t**

The time to transform the sub-objects.

#### **Matrix3& partm**

The 'parent' transformation matrix.

#### **Matrix3& tmAxis**

The axis system about which the selection is transformed.

#### **Point3& val**

The amount of the transformation relative to the axis system.

#### **BOOL localOrigin**

If TRUE the transformation takes place about the nodes local origin; otherwise about the world origin.

### **Return Value:**

See [List of NURBS Results](#) .

### **Function:**

**NURBSResult RotateCurrentSelection(Object\* pObj,  
NURBSSubObjectLevel level, TimeValue t, Matrix3& partm,  
Matrix3& tmAxis, Quat& val, BOOL localOrigin);**

**Remarks:**

This global function is available in release 3.0 and later only.  
Rotates the current sub-object selection at the specified level.

**Parameters:**

**Object\* pObj**

The object whose components are transformed.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**TimeValue t**

The time to transform the sub-objects.

**Matrix3& partm**

The 'parent' transformation matrix.

**Matrix3& tmAxis**

The axis system about which the selection is transformed.

**Quat& val**

The amount of the transformation relative to the axis system.

**BOOL localOrigin**

If TRUE the transformation takes place about the nodes local origin; otherwise about the world origin.

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

**NURBSResult ScaleCurrentSelection(Object\* pObj,  
NURBSSubObjectLevel level, TimeValue t, Matrix3& partm,  
Matrix3& tmAxis, Point3& val, BOOL localOrigin);**

**Remarks:**

This global function is available in release 3.0 and later only.  
Scales the current sub-object selection at the specified level.

**Parameters:****Object\* pObj**

The object whose components are transformed.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**TimeValue t**

The time to transform the sub-objects.

**Matrix3& partm**

The 'parent' transformation matrix.

**Matrix3& tmAxis**

The axis system about which the selection is transformed.

**Point3& val**

The amount of the transformation relative to the axis system.

**BOOL localOrigin**

If TRUE the transformation takes place about the nodes local origin; otherwise about the world origin.

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

```
int SubObjectCount(Object* pObj, NURBSSubObjectLevel level);
```

**Remarks:**

This global function is available in release 3.0 and later only.

Returns the number of sub-objects at the specified level.

**Parameters:****Object\* pObj**

Points to the object to check.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**Function:**

```
int NamedSelSetCount(Object* pObj, NURBSSubObjectLevel
```

**level);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns the number of named selection sets at the specified level

**Parameters:**

**Object\* pObj**

Points to the object to check.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**Function:**

**TCHAR\* GetNamedSelSetName(Object\* pObj,  
NURBSSubObjectLevel level, int i);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns the name of the 'i-th' named selection set of the specified sub-object level.

**Parameters:**

**Object\* pObj**

Points to the object to check.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**int i**

The zero based index of the selection set (between **0** and **NamedSelSetCount()-1**).

**Function:**

**NURBSResult GetNamedSelSet(Object\* pObj,  
NURBSSubObjectLevel level, TCHAR\* name, BitArray& selSet);**

**Remarks:**

This global function is available in release 3.0 and later only.

Retrieves the name of, and BitArray holding, the named selection set for the

specified sub-object level.

**Parameters:**

**Object\* pObj**

Points to the object whose named selection set is retrieved.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**TCHAR\* name**

The retrieved name of the set.

**BitArray& selSet**

The retrieved selection set data. See [Class BitArray](#).

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

**NURBSResult SetNamedSelSet(Object\* pObj,  
NURBSSubObjectLevel level, TCHAR\* name, BitArray& sel);**

**Remarks:**

This global function is available in release 3.0 and later only.

Set the specified named selection set. The set must exist. To add a new set see **AppendNamedSelSet()**.

**Parameters:**

**Object\* pObj**

Points to the object whose named selection set is stored.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**TCHAR\* name**

The name of the set.

**BitArray& sel**

The selection data. See [Class BitArray](#).

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

**NURBSResult AppendNamedSelSet(Object\* pObj,  
NURBSSubObjectLevel level, TCHAR\* name, BitArray& sel);**

**Remarks:**

This global function is available in release 3.0 and later only.  
Adds a new named selection set to the specified object.

**Parameters:****Object\* pObj**

Points to the object whose named selection set list is appended.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**TCHAR\* name**

The name for the set.

**BitArray& sel**

The selection data. See [Class BitArray](#).

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

**NURBSResult DeleteCurrentSelection(Object\* pObj,  
NURBSSubObjectLevel level);**

**Remarks:**

This global function is available in release 3.0 and later only.  
Deletes the current sub-object selection.

**Parameters:****Object\* pObj**

Points to the object whose current selection is deleted.

**NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

**NURBSResult MapNURBSIdToSelSetIndex(Object\* pObj,  
NURBSId id, int& index, NURBSSubObjectLevel& level);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns the index into a sub-object selection set of the sub-object component whose NURBSId is passed.

**Parameters:****Object\* pObj**

Points to the object whose sub-object component selection set index is returned.

**NURBSId id**

The ID of the sub-object to find.

**int& index**

The zero based selection set index is returned here.

**NURBSSubObjectLevel& level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

**Return Value:**

See [List of NURBS Results](#) .

**Function:**

**NURBSResult MapSelSetIndexToNURBSId(Object\* pObj, int  
index, NURBSSubObjectLevel level, NURBSId& id);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns the NURBSId of the sub-object component whose selection set index is passed.

**Parameters:****Object\* pObj**

Points to the object whose sub-object component NURBSId is returned.

**int index**

The zero based index into the selection set.

## **NURBSSubObjectLevel level**

The sub-object level. See [List of NURBSSubObjectLevel Options](#).

## **NURBSId& id**

The ID is returned here.

### **Return Value:**

See [List of NURBS Results](#) .

### **Function:**

```
void ApplyUVWMapAsTextureSurface(Object* pObj, int type,  
float utile, float vtile, float wtile, int uflip, int vflip, int wflip, int  
cap, const Matrix3 &tm,int channel);
```

### **Remarks:**

This global function is not currently supported as of 3ds max release 3.1.

### **Function:**

```
void UpdateSurfaceMapper(Modifier* pMod);
```

### **Remarks:**

This global function is available in release 3.0 and later only.

When called, passing in an instance of a NURBS Surface Mapper WSM, this has the effect of pressing the "Update" button. If any other modifier is passed in, the function does nothing.

### **Parameters:**

**Modifier\* pMod**

Points to the NURBS Surface Mapper WSM.

## Working with Patches

See Also: [Class Patch](#), [Class PatchMesh](#), [Class PatchObject](#), [Class PatchVec](#), [Class PatchVert](#), [Class PatchEdge](#), [Class TVPatch](#).

## Overview

This section provides an overview of how to work with Patches. It covers the main classes used when working with patches, and an example of creating a patch mesh.

The SDK provides a set of classes for creating and working with patches. A Patch is a three or four sided surface. Each edge of a patch is a bezier curve. Each edge has a vertex at the end and a vector attached to each of those ends. Inside a four sided patch are four interior vertices used to generate the surface inside. Inside a three sided patch are three interior vertices. Normally these vertices are set to be automatically calculated by the program to maintain continuity at the corners. They can however be set to be user-definable interior points.

## Overview of the Principal Classes

The SDK provides a set of classes for creating patch objects. These classes are listed below.

### [Class PatchObject](#)

This is the base class for the creation of Patch Objects in MAX. The **PatchObject** is the object that flows down the 3ds max geometry pipeline. This object stores a **PatchMesh** as well as a **Mesh** cache. This is similar to the way a **TriObject** has a **Mesh**. The **TriObject** provides methods to manage the operations associated with flowing down the pipeline, while the **Mesh** stores the vertex, edge and face information. In the **PatchObject** case, the **PatchObject** provides methods for working with the pipeline while the **Mesh** cache is used for display in the viewports.

### [Class PatchMesh](#)

A **PatchMesh** can be made up of any number of Patches. Each of these patches can be three or four sided. The topology of these patches are not quite as freeform as a regular **TriObject** mesh. For example, in a normal triangle mesh you can have an edge that is used by any number of faces. However with patches, in order for the system to figure out how the topology is connected together, each edge can only be used by either one patch (which makes it an open edge), or two patches (which makes it a transitional edge between the two).

### [Class Patch](#)

A **PatchMesh** is made up of a series of Patch objects derived from this class. This is similar to the way faces relate to a mesh, i.e. a **Patch** is like the face, and the **PatchMesh** is like the **Mesh**. Each Patch can be three or four sided. The **Patch** contains the vertices (three or four), and a set of vectors (six or eight) that go between the vertices (two per edge). There are also a set of vertices in the interior of the patch (three or four of these).

### [Class PatchVec](#)

This class represents a patch vector. This can be either an interior vector or an edge vector.

### [Class PatchVert](#)

This class stores the information associated with a patch vertex and provides methods to access this data. This information includes the vertex location, a table of vectors attached to the patch, and a table of patches using the vertex.

### [Class PatchEdge](#)

This class describes a patch edge using the vertices at the edge ends, and the indices of the patches sharing the edge.

### [Class TVPatch](#)

This class is a texture vertex patch structure. This is similar to the **TVFace** class used with a **Mesh**.

## Main Methods in Creating a Patch

A good sample to look at to see how patches are created is the patch grid plug-in. The source code is in

**\MAXSDK\SAMPLES\OBJECTS\PATCHGRD.CPP.**

The code below is a portion of the source of this plug-in. This section discusses some of the key methods used in the creation of the patch.

```
// Quad patch layout  
//  
// A---> ad ----- da <---D  
// | |  
// | |  
// v v  
// ab i1 i4 dc  
//  
// | |  
// | |  
//  
// ba i2 i3 cd  
// ^ ^  
// | |  
// | |  
// B---> bc ----- cb <---C  
//  
// Vertices ( a b c d ) are in counter clockwise order when viewed  
from  
// outside the surface  
void QuadPatchObject::BuildPatch(TimeValue t,PatchMesh&  
amesh)  
{  
int ix,iy,np,kv;  
int wsegs,lsegs,nv;  
Point3 v,p;  
float l, w;
```

**// Start the validity interval at forever and widdle it down.**

```
ivalid = FOREVER;  
pblock->GetValue( PB_LENGTH, t, l, ivalid );  
pblock->GetValue( PB_WIDTH, t, w, ivalid );  
pblock->GetValue( PB_LSEGS, t, lsegs, ivalid );  
pblock->GetValue( PB_WSEGS, t, wsegs, ivalid );  
int lv = lsegs + 1;  
int wv = wsegs + 1;  
int nverts = lv * wv;  
int npatches = lsegs * wsegs;  
int nexteriors = npatches * 4 + lsegs * 2 + wsegs * 2;  
int ninteriors = npatches * 4;  
int nvecs = ninteriors + nexteriors;
```

Here the number of vertices, texture vertices, vectors, patches and texture patches are established in the **PatchMesh**.

```
amesh.setNumVerts(nverts);  
amesh.setNumTVerts(textured ? nverts : 0);  
amesh.setNumVecs(nvecs);  
amesh.setNumPatches(npatches);  
amesh.setNumTVPatches(textured ? npatches : 0);  
v = Point3(-w, -l, 0.0f) / 2.0f;  
float dx = w/wsegs;  
float dy = l/lsegs;  
float fws = (float)wsegs;  
float fls = (float)lsegs;
```

Next the vertices are created. The **setVert()** method of the **PatchMesh** is used to do this.

```
// Create the vertices.  
nv = 0;  
p.z = v.z;  
p.y = v.y;  
for(iy=0; iy<=lsegs; iy++) {  
    p.x = v.x;  
    for (ix=0; ix<=wsegs; ix++) {
```

```

if(textured)
    amesh.tVerts[nv] = UVVert((float)ix /
        fws, (float)iy / fls, 0.0f);
amesh.verts[nv].flags = PVERT_COPLANAR;
amesh.setVert(nv++, p);
p.x += dx;
}
p.y += dy;
}

```

Next the patches are created. The **PatchMesh** method **MakeQuadPatch()** is used once the vertices, vectors, and interiors are set up.

```

// Create patches.
np = 0;
int interior = nexteriors;
int vecRowInc = lsegs * 2;
int vecColInc = wsegs * 2;
for(iy=0; iy<lsegs; iy++) {
    kv = iy*(wsegs+1);
    int rv = iy * vecColInc; // Row vector start
    int cv = vecColInc * lv + iy * 2; // column vector start
    for (ix=0; ix<wsegs; ix++,++np) {
        Patch &p = amesh.patches[np];
        int a = kv, b = kv+1, c = kv+wsegs+2, d = kv + wsegs + 1;
        int ab = rv, ba = rv+1;
        int bc = cv+vecRowInc, cb = cv + vecRowInc + 1;
        int cd = rv+vecColInc+1, dc = rv+vecColInc;
        int da = cv + 1, ad = cv;
        amesh.MakeQuadPatch(np, a, ab, ba, b, bc, cb, c, cd, dc,
            d, da, ad, interior+1, interior+2,
            interior+3, 1);
    }
}

```

If textures are being used, then **setTVerts()** is called.

```

if(textured) {
    TVPatch &tp = amesh.tvPatches[np];
    tp.setTVerts(a,b,c,d);
}

```

```
}
```

Next the vectors are set using **setVec()**. One third of the distance from vertex **a** to vertex **b** is vector **ab**, and one third of the distance from vertex **b** to vertex **a** is **ba**.

```
// Create the default vectors
Point3 pa = amesh.getVert(a).p;
Point3 pb = amesh.getVert(b).p;
Point3 pc = amesh.getVert(c).p;
Point3 pd = amesh.getVert(d).p;
amesh.setVec(ab, pa + (pb - pa) / 3.0f);
amesh.setVec(ba, pb - (pb - pa) / 3.0f);
amesh.setVec(bc, pb + (pc - pb) / 3.0f);
amesh.setVec(cb, pc - (pc - pb) / 3.0f);
amesh.setVec(cd, pc + (pd - pc) / 3.0f);
amesh.setVec(dc, pd - (pd - pc) / 3.0f);
amesh.setVec(da, pd + (pa - pd) / 3.0f);
amesh.setVec(ad, pa - (pa - pd) / 3.0f);
kv++;
cv += vecRowInc;
rv += 2;
interior += 4;
}
}
```

```
// Verify that we have the right number of parts!
```

```
assert(np==npatches);
```

```
assert(nv==nverts);
```

Next the **PatchMesh** method **buildLinkages()** is called. This makes sure that everything is connected correctly and there are not any edges that are used by more than two patches. It will return **FALSE** if these conditions are not met.

```
// Finish up patch internal linkages (and bail out if it fails!)
```

```
assert(amesh.buildLinkages());
```

Next the method **computeInteriors()** is called. For any automatic patches, the interior vertices will be computed.

```
// Calculate the interior bezier points on the PatchMesh's patches  
amesh.computeInteriors();
```

Then the geometry cache is cleared to make sure that any cache that might have been in the **PatchMesh** is emptied.

```
amesh.InvalidateGeomCache();
```

Finally the validity of the **Mesh** cache is set to FALSE to indicate the **PatchMesh** has been changed.

```
// Tell the PatchObject its mesh just got changed  
meshValid = FALSE;  
}
```

## Patch Interpolation

The patch system supports any combination of quadrilateral and triangular patches, as long as they are set up so that a single edge is only shared by at most two patches.

Adjacent patches share control points and vectors on their common edges. This minimizes the amount of data and makes book keeping easier. It also guarantees matching patch edges.

The only problem with this system is that the 3ds max quad patches are bicubic (degree 3), and if they are connected to degree 3 triangular patches, discontinuities will result at the boundaries. The solution used in 3ds max is to store the patch information and allow users to manipulate it in a degree-3 manner (4 points on the side of a patch). When we go to interpolate it, however, we convert all the triangular patches' control points to degree 4, where they can be completely compatible with their quad neighbors.

Below is the source code for the interpolators for both the quad and tri patches.

Note: Interested developers should see the "Patch Tessellation" upload in the private [Sparks Developer](#) section on the webboard. Here developers can get a working example on how to tessellate a PatchMesh for use in a game engine. This modifier sample is available in the [Sparks Developer](#) section of the webboard at <http://sparks.discreet.com>

Also Note: The UpdateHooks() method shown below is modified from the original version in PATCH.CPP.

**// Triangular patch interpolator**

**Point3 Patch::interp(PatchMesh \*pMesh,float pu, float pv, float pw) {**

**// It had better be a triangular patch!**

**assert(type == PATCH\_TRI);**

**Point3 p;**

**PatchVert \*vp = pMesh->verts;**

**PatchVec \*vecp = pMesh->vecs;**

**float pu2 = pu \* pu;**

**float pu3 = pu2 \* pu;**

**float pu4 = pu3 \* pu;**

**float pv2 = pv \* pv;**

**float pv3 = pv2 \* pv;**

```

float pv4 = pv3 * pv;
float pw2 = pw * pw;
float pw3 = pw2 * pw;
float pw4 = pw3 * pw;
// Hold on to your hats -- Here it comes!
p = vp[v[0]].p * pw4 +
aux[0] * 4.0f * pu * pw3 +
aux[1] * 6.0f * pu2 * pw2 +
aux[2] * 4.0f * pu3 * pw +
vp[v[1]].p * pu4 +
aux[3] * 4.0f * pv * pu3 +
aux[4] * 6.0f * pu2 * pv2 +
aux[5] * 4.0f * pv3 * pu +
vp[v[2]].p * pv4 +
aux[6] * 4.0f * pw * pv3 +
aux[7] * 6.0f * pv2 * pw2 +
aux[8] * 4.0f * pw3 * pv +
vecp[interior[0]].p * 12.0f * pu * pv * pw2 +
vecp[interior[1]].p * 12.0f * pu2 * pv * pw +
vecp[interior[2]].p * 12.0f * pu * pv2 * pw;
return p;
}
// Quadrilateral patch interpolator
Point3 Patch::interp(PatchMesh *pMesh,float pu, float pv) {
// It had better be a quad patch!
assert(type == PATCH_QUAD);
Point3 p;
PatchVert *vp = pMesh->verts;
PatchVec *vecp = pMesh->vecs;
float pu2 = pu * pu;
float pu1 = 1.0f - pu;
float pu12 = pu1 * pu1;

```

```

float u0 = pu12 * pu1;
float u1 = 3.0f * pu * pu12;
float u2 = 3.0f * pu2 * pu1;
float u3 = pu2 * pu;
float pv2 = pv * pv;
float pv1 = 1.0f - pv;
float pv12 = pv1 * pv1;
float v0 = pv12 * pv1;
float v1 = 3.0f * pv * pv12;
float v2 = 3.0f * pv2 * pv1;
float v3 = pv2 * pv;
// Hold on to your hats -- Here it comes!
p = vp[v[0]].p * u0 * v0 +
vecp[vec[7]].p * u1 * v0 +
vecp[vec[6]].p * u2 * v0 +
vp[v[3]].p * u3 * v0 +
  vecp[vec[0]].p * u0 * v1 +
vecp[interior[0]].p * u1 * v1 +
vecp[interior[3]].p * u2 * v1 +
vecp[vec[5]].p * u3 * v1 +
  vecp[vec[1]].p * u0 * v2 +
vecp[interior[1]].p * u1 * v2 +
vecp[interior[2]].p * u2 * v2 +
vecp[vec[4]].p * u3 * v2 +
  vp[v[1]].p * u0 * v3 +
vecp[vec[2]].p * u1 * v3 +
vecp[vec[3]].p * u2 * v3 +
vp[v[2]].p * u3 * v3;
return p;
}

```

Building the vertices are just a matter of looping through your patches and then running through each patches UV space to get the corresponding point on the

surface. Below is sample showing the basics.

```
float fpd = (float)patchDivs;
for(px = 0; px < numPatches; ++px) {
    Patch &p = patches[px];
    switch(p.type) {
        case PATCH_TRI: {
            for(int ax = patchDivs - 1; ax > 0; --ax) {
                for(int bx = 1; bx < ax; ++bx) {
                    float u = (float)bx / fpd;
                    float v = (float)(patchDivs - ax) / fpd;
                    float w = 1.0f - u - v; // Barycentric validity guaranteed!
                    mesh.setVert(vert++, p.interp(this, u, v, w));
                }
            }
        }
        break;
        case PATCH_QUAD: {
            for(int u = 1; u < patchDivs; ++u) {
                float fu = (float)u / fpd;
                for(int v = 1; v < patchDivs; ++v) {
                    float fv = (float)v / fpd;
                    mesh.setVert(vert++, p.interp(this, fu, fv));
                }
            }
        }
        break;
    }
}
```

Below is the sample code that puts it all together. It shows the interpolation of the surface points, the book keeping code for shared edges, the construction of the mesh faces, the handling of texture vertices and faces, and how bind patches are handled.

```

// vertices ( a b c d ) are in counter clockwise order when viewed from
// outside the surface
//watje 12-8-98
static void MakeQuad(int nVerts, Face *f, int a, int b , int c , int d, DWORD
sm, MtlID m,
    int e1a, int e1b, int e1c,
    int e2a, int e2b, int e2c,
    int hide
) {
//DebugPrint("Make quad %d %d %d %d \n",a,b,c,d);
assert(a<nVerts);
assert(b<nVerts);
assert(c<nVerts);
assert(d<nVerts);
f[0].setVerts( a, b, c);
f[0].setSmGroup(sm);
f[0].setEdgeVisFlags(e1a,e1b,e1c);
f[0].setMatID(m);
f[0].SetHide(hide);
f[1].setVerts( c, d, a);
f[1].setSmGroup(sm);
f[1].setEdgeVisFlags(e2a,e2b,e2c);
f[1].setMatID(m);
f[1].SetHide(hide);
}
//watje 12-8-98
#define MAKE_QUAD(na,nb,nc,nd,sm,m,e1a,e1b,e1c,e2a,e2b,e2c,hide) {
MakeQuad(nVerts,&(mesh.faces[face]),na, nb, nc, nd, sm,
m,e1a,e1b,e1c,e2a,e2b,e2c,hide); face+=2; }
// Make texture face quad
static void MakeTQuad(int chan, int nTVerts, TVFace *f, int a, int b , int c ,
int d) {

```

```

assert(a<nTVerts);
assert(b<nTVerts);
assert(c<nTVerts);
assert(d<nTVerts);
f[0].setTVerts( a, b, c);
f[1].setTVerts( c, d, a);
}
#define MAKE_TQUAD(ch,na,nb,nc,nd) { MakeTQuad(ch, nTVerts[ch], &
(mesh.mapFaces(ch)[tface]),na, nb, nc, nd); tface+=2; }
// vertices ( a b c d ) are in counter clockwise order when viewed from
// outside the surface
//watje 12-8-98
static void MakeTri(int nVerts, Face *f, int a, int b, int c, DWORD sm,
MtlID m,
    int ea, int eb, int ec, int hide
    ) {
assert(a<nVerts);
assert(b<nVerts);
assert(c<nVerts);
//DebugPrint("Making tri %d %d %d \n",a,b,c);
f->setVerts( a, b, c);
f->setSmGroup(sm);
f->setEdgeVisFlags(ea,eb,ec);
f->setMatID(m);
f->SetHide(hide);
}
//watje 12-8-98
#define MAKE_TRI(na,nb,nc,sm,mat,ea,eb,ec,hide) { MakeTri(nVerts,&
(mesh.faces[face]),na, nb, nc, sm, mat,ea,eb,ec,hide); face++; }
#define MAKE_QUAD_SPECIAL(na,nm,nb,nc,sm,mat,ea,eb,ec,ed,hide) \
{ MakeTri(nVerts,&(mesh.faces[face]),na, nm, nc, sm, mat,ea,0,ed,hide);
face++; \

```

```

    MakeTri(nVerts,&(mesh.faces[face]),nm, nb, nc, sm, mat,eb,ec,0,hide);
face++; }
#define MAKE_PENTA_EDGE1(na,nm,nb,nc,nd,sm,mat,ea,eb,ec,ed,hide)
{ \
MakeTri(nVerts,&(mesh.faces[face]),na, nm, nc, sm, mat,ea,0,0,hide);
face++; \
    MakeTri(nVerts,&(mesh.faces[face]),nm, nb, nc, sm, mat,ea,eb,0,hide);
face++; \
    MakeTri(nVerts,&(mesh.faces[face]),nc, nd, na, sm, mat,ec,ed,0,hide);
face++; \
}
#define MAKE_PENTA_EDGE2(na,nb,nm,nc,nd,sm,mat,ea,eb,ec,ed,hide)
{ \
    MakeTri(nVerts,&(mesh.faces[face]),na,nb,nm, sm, mat,ea,eb,0,hide);
face++; \
    MakeTri(nVerts,&(mesh.faces[face]),na, nm, nc, sm, mat,0,eb,0,hide);
face++; \
    MakeTri(nVerts,&(mesh.faces[face]),nc, nd, na, sm, mat,ec,ed,0,hide);
face++; \
}
#define MAKE_PENTA_EDGE3(na,nb,nc,nm,nd,sm,mat,ea,eb,ec,ed,hide)
{ \
    MakeTri(nVerts,&(mesh.faces[face]),na,nb,nc, sm, mat,ea,eb,0,hide);
face++; \
    MakeTri(nVerts,&(mesh.faces[face]),nc, nm, na, sm, mat,ec,0,0,hide);
face++; \
    MakeTri(nVerts,&(mesh.faces[face]),nm, nd, na, sm, mat,ec,ed,0,hide);
face++; \
}
#define MAKE_PENTA_EDGE4(na,nb,nc,nd,nm,sm,mat,ea,eb,ec,ed,hide)
{ \
    MakeTri(nVerts,&(mesh.faces[face]),na,nb,nc, sm, mat,ea,eb,0,hide);
face++; \
    MakeTri(nVerts,&(mesh.faces[face]),nc, nd, nm, sm, mat,ec,ed,0,hide);

```

```

face++; \
    MakeTri(nVerts,&(mesh.faces[face]),nc, nm, na, sm, mat,0,ed,0,hide);
face++; \
}
// Make texture face triangle
static void MakeTTri(int chan, int nTVerts, TVFace *f, int a, int b, int c) {
    assert(a<nTVerts);
    assert(b<nTVerts);
    assert(c<nTVerts);
    f->setTVerts( a, b, c);
}
#define MAKE_TTRI(ch,na,nb,nc) { MakeTTri(ch, nTVerts[ch], &
(mesh.mapFaces(ch)[tface]),na, nb, nc); tface++; }
#define MAKE_TQUAD_SPECIAL(ch,na,nm,nb,nc) \
{ MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),na, nm, nc);
tface++; \
    MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),nm, nb, nc);
tface++; \
    tvertHookStart++; }
#define MAKE_TPENTA_EDGE1(ch,na,nm,nb,nc,nd) \
{ \
    MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),na, nm, nc);
tface++; \
    MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),nm, nb, nc);
tface++; \
    MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),nc, nd, na);
tface++; \
    tvertHookStart++; }
#define MAKE_TPENTA_EDGE2(ch,na,nb,nm,nc,nd) \
{ \
    MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),na, nb, nm);
tface++; \
    MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),na, nm, nc);

```

```

tface++; \
MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),nc, nd, na);
tface++; \
tvertHookStart++; }
#define MAKE_TPENTA_EDGE3(ch,na,nb,nc,nm,nd) \
{ \
MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),na, nb, nc);
tface++; \
MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),nc, nm, na);
tface++; \
MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),nm, nd, na);
tface++; \
tvertHookStart++; }
#define MAKE_TPENTA_EDGE4(ch,na,nb,nc,nd,nm) \
{ \
MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),na, nb, nc);
tface++; \
MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),nc, nd, nm);
tface++; \
MakeTTri(ch, nTVerts[ch], &(mesh.mapFaces(ch)[tface]),nc, nm, na);
tface++; \
tvertHookStart++; }
void PatchMesh::PrepareMesh() {
// If mesh is already there, just return it!
if(meshValid && cacheSteps == GetMeshSteps() && cacheAdaptive ==
GetAdaptive())
return;
// TH 3/23/99 -- Added this because something in the pipeline is updating
the
// vert/vector coords without calling computeInteriors()!
computeAux();
// Keep a record of the settings used for the cache
cacheSteps = GetMeshSteps();

```

```

cacheAdaptive = GetAdaptive();
int patchDivs = cacheAdaptive ? 11 : cacheSteps + 1 ;
float fpd = (float)patchDivs;
int workSteps = patchDivs - 1;
int px;
int nFaces = 0;
int nVerts = 0;
// Calc the number of verts and faces in the mesh
nVerts += numVerts;    // Corner verts
nVerts += (getNumEdges() * (patchDivs - 1)); // Edge verts
// watje compute the hook verts additions
nVerts += hooks.Count() * (patchDivs) - hooks.Count() - (hooks.Count() *2
* (patchDivs - 1));
// Precalculate interior vertex counts
int quadint = (patchDivs - 1) * (patchDivs - 1);
int triint = (patchDivs + 1) * (patchDivs + 2) / 2 - patchDivs * 3;
// Precalculate number of texture verts
int quadTVerts = (patchDivs + 1) * (patchDivs + 1);
int triTVerts = (patchDivs + 1) * (patchDivs + 2) / 2;
// Now the interior verts
for(px = 0; px < numPatches; ++px) {
    switch(patches[px].type) {
        case PATCH_TRI:
            nFaces += (patchDivs * patchDivs);
            nVerts += triint;
            break;
        case PATCH_QUAD:
            nFaces += (patchDivs * patchDivs * 2);
            nVerts += quadint;
            break;
    }
}

```

```

// watje compute the hook face additions
nFaces += hooks.Count() * (patchDivs);
// watje compute the hook texture verts additions
// Build the vertices!
mesh.setNumVerts(nVerts);
// The first vertices in the Mesh match those in the PatchMesh
for(int i = 0; i < numVerts; ++i)
    mesh.setVert(i, verts[i].p);
int edgeBase = numVerts;
int vert = edgeBase;
// Now we put in the edge vertices
// Set up a spline object we'll use for ease of edge interpolation
Spline3D work;
work.AddKnot(SplineKnot(KTYPE_BEZIER, LTYPE_CURVE,
zeroPoint, zeroPoint, zeroPoint));
work.AddKnot(SplineKnot(KTYPE_BEZIER, LTYPE_CURVE,
zeroPoint, zeroPoint, zeroPoint));
Tab<int> hookEdges;
hookEdges.SetCount(getNumEdges());
Tab<int> hookEdgeList;
hookEdgeList.SetCount(hooks.Count() * (patchDivs*2-1));
BitArray eEdges;
eEdges.SetSize(getNumEdges());
eEdges.ClearAll();
for(i = 0; i < getNumEdges(); i++) {
    hookEdges[i] = -1;
}
for(i = 0; i < hooks.Count(); i++)
{
    hookEdges[hooks[i].hookEdge] = i;
    int a = hooks[i].upperEdge;
    int b = hooks[i].lowerEdge;

```

```

eEdges.Set(a);
eEdges.Set(b);
}
Tab<int> edgeStart;
Tab<int> edgeEnd;
Tab<int> hookEdgeStart;
Tab<int> hookEdgeEnd;
edgeStart.SetCount(getNumEdges());
edgeEnd.SetCount(getNumEdges());
hookEdgeStart.SetCount(hooks.Count());
hookEdgeEnd.SetCount(hooks.Count());
int ct = 0;
for(i = 0; i < getNumEdges(); ++i) {
    PatchEdge &edge = edges[i];
    work.SetKnotPoint(0, verts[edge.v1].p);
    work.SetOutVec(0, vecs[edge.vec12].p);
    work.SetInVec(1, vecs[edge.vec21].p);
    work.SetKnotPoint(1, verts[edge.v2].p);
//need to check if edge is hooked if so double the number of points - 1
    if (hookEdges[i]!=-1)
    {
//need to take out middle point since this the corner of another patch
        edgeStart[i] = vert;
        int index = hookEdges[i];
        hookEdgeStart[index]=ct;
        edgeStart[hooks[index].upperEdge] = vert;
        int mid = (patchDivs+patchDivs)/2;
        for(int terp = 1; terp < (patchDivs+patchDivs); ++terp)
        {
//add ehook edges upper start
            if (terp != mid)
            {

```

```

hookEdgeList[ct++] = vert;
mesh.setVert(vert++, work.InterpCurve3D((float)terp / (fpd+fpd)));
}
else
{
//add ehook edges upper end
hookEdgeList[ct++] = hooks[index].hookPoint;
edgeEnd[hooks[index].upperEdge] = vert-1;
//add ehook edges lower start
edgeStart[hooks[index].lowerEdge] = vert;
}
}
//add ehook edges lower end
hookEdgeEnd[index]=ct-1;
edgeEnd[hooks[index].lowerEdge] = vert-1;
//check if these need to be flipped
int u,l;//,h;
u = edge.v1;
l = edge.v2;
PatchEdge eu = edges[hooks[index].upperEdge];
PatchEdge el = edges[hooks[index].lowerEdge];
if (u != eu.v1)
{
int t = edgeEnd[hooks[index].upperEdge];
edgeEnd[hooks[index].upperEdge] = edgeStart[hooks[index].upperEdge];
edgeStart[hooks[index].upperEdge] = t;
}
if (l != el.v2)
{
int t = edgeEnd[hooks[index].lowerEdge];
edgeEnd[hooks[index].lowerEdge] = edgeStart[hooks[index].lowerEdge];
edgeStart[hooks[index].lowerEdge] = t;
}
}

```

```

    }
    edgeEnd[i] = vert-1;
    }
    else
    {
//look for edges that are share with a hook patch and don't use since we
already
    if ( !eEdges[i] )
    {
        edgeStart[i] = vert;
        for(int terp = 1; terp < patchDivs; ++terp)
        {
            mesh.setVert(vert++, work.InterpCurve3D((float)terp / fpd));
        }
        edgeEnd[i] = vert-1;
    }
    }
}
// Now generate the patch interior points
int interiorBase = vert;
IntTab patchVOff;
for(px = 0; px < numPatches; ++px) {
    patchVOff.Append(1, &vert); // Keep a record of this patch's starting
interior vertex
    Patch &p = patches[px];
    switch(p.type) {
    case PATCH_TRI: {
        for(int ax = patchDivs - 1; ax > 0; --ax) {
            for(int bx = 1; bx < ax; ++bx) {
                float u = (float)bx / fpd;
                float v = (float)(patchDivs - ax) / fpd;
                float w = 1.0f - u - v; // Barycentric validity guaranteed!
            }
        }
    }
    }
}

```

```

    mesh.setVert(vert++, p.interp(this, u, v, w));
    }
    }
}
break;
case PATCH_QUAD: {
    for(int u = 1; u < patchDivs; ++u) {
        float fu = (float)u / fpd;
        for(int v = 1; v < patchDivs; ++v) {
            float fv = (float)v / fpd;
            mesh.setVert(vert++, p.interp(this, fu, fv));
        }
    }
}
break;
}
}
assert(vert == nVerts);

// Build the faces!
// This is a bit tricky because we're sharing vertices at patch edges
mesh.setNumFaces(nFaces);
int face = 0;
//watje 12-10-98
BOOL hidden;
//build hook patch db so we can quickly determine which edge is hooked
Tab<int> hookPatches;
Tab<int> hookPoints;
hookPatches.SetCount(numPatches);
hookPoints.SetCount(numPatches);
for(px = 0; px < numPatches; px++)
{

```

```

hookPatches[px] = 0;
hookPoints[px] = -1;
}
for(px = 0; px < hooks.Count(); px++)
{
int a= hooks[px].hookPatch;
//find which edge is the hook
int ea,eb;
ea = hooks[px].upperPoint;
eb = hooks[px].lowerPoint;
Patch p = patches[a];
int pa,pb,pc,pd;
pa = p.v[0];
pb = p.v[1];
pc = p.v[2];
if (p.type == PATCH_TRI)
pd = -1;
else pd = p.v[3];
//check ab edge
if ( ((pa==ea) && (pb==eb)) ||
((pa==eb) && (pb==ea)) )
{
hookPatches[a] |= 1;
hookPoints[a] = px;
}
//check bc edge
else if ( ((pb==ea) && (pc==eb)) ||
((pb==eb) && (pc==ea)) )
{
hookPatches[a] |= 2;
hookPoints[a] = px;
}
}

```

```

//check if tri
else if (pd == -1)
{
//check ca edge
if ( ((pc==ea) && (pa==eb)) ||
((pc==eb) && (pa==ea)) )
{
hookPatches[a] |= 4;
hookPoints[a] = px;
}
}
else
{
//check cd edge
if ( ((pc==ea) && (pd==eb)) ||
((pc==eb) && (pd==ea)) )
{
hookPatches[a] |= 4;
hookPoints[a] = px;
}
//check da edge
else if ( ((pd==ea) && (pa==eb)) ||
((pd==eb) && (pa==ea)) )
{
hookPatches[a] |= 8;
hookPoints[a] = px;
}
}
}
// Table used to store texture vertex base for each patch
Tab<IntTab *> patchTVOff;
Tab<int> nTVerts;

```

```

patchTVOff.SetCount (getNumMaps());
for (int chan=0; chan<getNumMaps(); chan++) patchTVOff[chan] =
NULL;
nTVerts.SetCount (getNumMaps());
int tvertHookStart;
// Build texture verts if necessary
for(chan = 0; chan < getNumMaps(); ++chan) {
nTVerts[chan] = 0;
if(tvPatches[chan]) {
patchTVOff[chan] = new IntTab;
for(px = 0; px < numPatches; ++px) {
Patch &p = patches[px];
patchTVOff[chan]->Append(1, &nTVerts[chan]);
switch(p.type) {
case PATCH_TRI:
nTVerts[chan] += (patchDivs + 1) * (patchDivs + 2) / 2;
break;
case PATCH_QUAD:
nTVerts[chan] += (patchDivs + 1) * (patchDivs + 1);
break;
}
}
}
//add in additional vertices for hooks
int hv = hooks.Count() * (patchDivs);
tvertHookStart = nTVerts[chan];
nTVerts[chan] += hv;
mesh.setMapSupport (chan);
mesh.setNumMapVerts (chan, nTVerts[chan]);
// Generate the texture verts!
for(px = 0; px < numPatches; ++px) {
Patch &p = patches[px];
TVPatch &tp = getMapPatch (chan,px);

```

```

int tvert = (*patchTVOff[chan])[px];
switch(p.type) {
case PATCH_TRI: {
    UVVert &t0 = tVerts[chan][tp.tv[0]];
    UVVert &t1 = tVerts[chan][tp.tv[1]];
    UVVert &t2 = tVerts[chan][tp.tv[2]];
    for(int ax = patchDivs; ax >= 0; --ax) {
        for(int bx = 0; bx <= ax; ++bx) {
            float u = (float)bx / fpd;
            float v = (float)(patchDivs - ax) / fpd;
            float w = 1.0f - u - v; // Barycentric validity guaranteed!
            Point3 vertLoc = t0 * w + t1 * u + t2 * v;
            mesh.setMapVert (chan, tvert++, vertLoc);
        }
    }
    break;
case PATCH_QUAD: {
    UVVert &t0 = tVerts[chan][tp.tv[0]];
    UVVert &t1 = tVerts[chan][tp.tv[1]];
    UVVert &t2 = tVerts[chan][tp.tv[2]];
    UVVert &t3 = tVerts[chan][tp.tv[3]];
    for(int v = 0; v <= patchDivs; ++v) {
        float fv = (float)v / fpd;
        UVVert iv1 = t0 + (t3 - t0) * fv;
        UVVert iv2 = t1 + (t2 - t1) * fv;
        for(int u = 0; u <= patchDivs; ++u) {
            float fu = (float)u / fpd;
            UVVert iu = iv1 + (iv2 - iv1) * fu;
            mesh.setMapVert (chan, tvert++, iu);
        }
    }
}
}

```

```

    }
    break;
}
}
//now add additional hook points
int tvIndex = tvertHookStart;
for(px = 0; px < numPatches; ++px)
{
    Patch &p = patches[px];
    TVPatch &tp = getMapPatch (chan,px);
    UVVert t0;
    UVVert t1;
    if (hookPatches[px]&1)
    {
        t0 = tVerts[chan][tp.tv[0]];
        t1 = tVerts[chan][tp.tv[1]];
    }
    else if (hookPatches[px]&2)
    {
        t0 = tVerts[chan][tp.tv[1]];
        t1 = tVerts[chan][tp.tv[2]];
    }
    else if (hookPatches[px]&4)
    {
        if (p.type == PATCH_TRI)
        {
            t0 = tVerts[chan][tp.tv[2]];
            t1 = tVerts[chan][tp.tv[0]];
        }
        else
        {
            t0 = tVerts[chan][tp.tv[2]];

```

```

    t1 = tVerts[chan][tp.tv[3]];
    }
    }
else if (hookPatches[px]&8)
    {
    t0 = tVerts[chan][tp.tv[3]];
    t1 = tVerts[chan][tp.tv[0]];
    }
if (hookPatches[px]!=0)
    {
    Point3 vec = (t1-t0)/(fpd*2.0f);
//put first in TV for start corner
    t0 += vec;
    mesh.setMapVert (chan, tvIndex++, t0);
//put last in TV for end corner
    if (patchDivs > 1)
        {
        Point3 tEnd;
        tEnd = t0+ (vec * ((float)(patchDivs*2)-2.0f));
        mesh.setMapVert (chan, tvIndex++, tEnd);
//put middle in TV for edges
        t0 += vec;
        for(int v = 2; v <= ((patchDivs*2)-3); ++v) {
            if ((v%2) == 1) mesh.setMapVert (chan, tvIndex++, t0);
            t0 += vec;
        }
        }
    }
}
if (nTVerts[chan]!=tvIndex)
    {
DebugPrint("error\n");

```

```

    }
    }
else {
    mesh.setMapSupport (chan, 0);
}
}
for(px = 0; px < numPatches; ++px) {
    Patch &p = patches[px];
    int baseVert = patchVOff[px]; // Interior vertex base
    MtlID matid = p.getMatID();
    hidden = p.IsHidden();
    switch(p.type) {
    case PATCH_TRI: {
        int vbase = baseVert;
        // If only need to make 1 triangle, make it and blow!
        //no need to check if it is a hooked patch is so one edge or modre needs to
be split
//watje 12-10-98
        if(workSteps == 0) {
            if (hookPatches[px]==0)
            {
                MAKE_TRI(p.v[0],p.v[1],p.v[2], p.smGroup, matid,1,1,1,hidden);
            }
            else
            {
                if (hookPatches[px]&1)
                {
                    int ex = hooks[hookPoints[px]].hookPoint;
                    MAKE_QUAD_SPECIAL(p.v[0],ex,p.v[1],p.v[2], p.smGroup,
matid,1,1,1,1,hidden);
                }
                else if (hookPatches[px]&2)

```

```

    {
        int ex = hooks[hookPoints[px]].hookPoint;
        MAKE_QUAD_SPECIAL(p.v[1],ex,p.v[2],p.v[0], p.smGroup,
matid,1,1,1,1,hidden);
    }
    else if (hookPatches[px]&4)
    {
        int ex = hooks[hookPoints[px]].hookPoint;
        MAKE_QUAD_SPECIAL(p.v[2],ex,p.v[0],p.v[1], p.smGroup,
matid,1,1,1,1,hidden);
    }
    }
    break;
}
if(workSteps > 2) {
    // Make inside faces if necessary
    for(int ax = 0; ax < (workSteps - 2); ++ax) {
        int vix = vbase;
        for(int bx = 0; bx < (workSteps - 2 - ax); ++bx, ++vix)
//watje 12-10-98
        {
            if (showInterior)
            {
                MAKE_TRI(vix, vix + 1, vix + workSteps - 1 - ax, p.smGroup,
matid,1,1,1,hidden);
            }
            else{
                MAKE_TRI(vix, vix + 1, vix + workSteps - 1 - ax, p.smGroup,
matid,0,0,0,hidden);
            }
        }
    }
    vix = vbase;
}

```

```

//watje 12-10-98
    for(bx = 1; bx < (workSteps - 2 - ax); ++bx, ++vix)
    {
//watje
        if (showInterior)
        {
            MAKE_TRI(vix + 1, vix + workSteps - ax, vix + workSteps - 1 - ax,
p.smGroup, matid,1,1,1,hidden);
        }
        else{
            MAKE_TRI(vix + 1, vix + workSteps - ax, vix + workSteps - 1 - ax,
p.smGroup, matid,0,0,0,hidden);
        }
    }
    vbase += (workSteps - 1 - ax);
}
}
// Make corners
int edge0 = p.edge[0];
int edge1 = p.edge[1];
int edge2 = p.edge[2];
PatchEdge &e0 = edges[edge0];
PatchEdge &e1 = edges[edge1];
PatchEdge &e2 = edges[edge2];
int e0start,e1start,e2start,e0end,e1end,e2end,e0dir,e1dir,e2dir;
if(e0.v1 == p.v[0]) {
    e0start = edgeStart[edge0];
    e0end = edgeEnd[edge0];
    e0dir = 1;
}
else {
    e0start = edgeEnd[edge0];

```

```
e0end = edgeStart[edge0];
e0dir = -1;
}
if(e1.v1 == p.v[1]) {
    e1start = edgeStart[edge1];
    e1end = edgeEnd[edge1];
    e1dir = 1;
}
else {
    e1start = edgeEnd[edge1];
    e1end = edgeStart[edge1];
    e1dir = -1;
}
if(e2.v1 == p.v[2]) {
    e2start = edgeStart[edge2];
    e2end = edgeEnd[edge2];
    e2dir = 1;
}
else {
    e2start = edgeEnd[edge2];
    e2end = edgeStart[edge2];
    e2dir = -1;
}
if (e2start > e2end) e2dir = -1;
    else e2dir = 1;
if (e1start > e1end) e1dir = -1;
    else e1dir = 1;
if (e0start > e0end) e0dir = -1;
    else e0dir = 1;
// Create the corners
int i0 = baseVert;
int i1 = baseVert + (workSteps - 2);
```

```

int i2 = baseVert + triint - 1;
if (showInterior)
{
if (hookPatches[px]&1)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];
int e1 = hookEdgeList[end-1];
int e2 = hookEdgeList[end];
MAKE_QUAD_SPECIAL(p.v[0],s1,s2,e2end, p.smGroup,
matid,1,1,1,1,hidden);
MAKE_QUAD_SPECIAL(e1,e2,p.v[1],e1start, p.smGroup,
matid,1,1,1,1,hidden);
MAKE_TRI(p.v[2], e2start, e1end, p.smGroup, matid,1,1,1,hidden);
}
else if (hookPatches[px]&2)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];
int e1 = hookEdgeList[end-1];
int e2 = hookEdgeList[end];
MAKE_TRI(p.v[0], e0start, e2end, p.smGroup, matid,1,1,1,hidden);
MAKE_QUAD_SPECIAL(p.v[1],s1,s2,e0end, p.smGroup,
matid,1,1,1,1,hidden);
MAKE_QUAD_SPECIAL(e1,e2,p.v[2],e2start, p.smGroup,
matid,1,1,1,1,hidden);
}
}

```

```

}
else if (hookPatches[px]&4)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];
int e1 = hookEdgeList[end-1];
int e2 = hookEdgeList[end];
MAKE_TRI(p.v[1], e1start, e0end, p.smGroup, matid,1,1,1,hidden);
MAKE_QUAD_SPECIAL(p.v[2],s1,s2,e1end, p.smGroup,
matid,1,1,1,1,hidden);
MAKE_QUAD_SPECIAL(e1,e2,p.v[0],e0start, p.smGroup,
matid,1,1,1,1,hidden);
}
else
{
MAKE_TRI(p.v[0], e0start, e2end, p.smGroup, matid,1,1,1,hidden);
MAKE_TRI(p.v[1], e1start, e0end, p.smGroup, matid,1,1,1,hidden);
MAKE_TRI(p.v[2], e2start, e1end, p.smGroup, matid,1,1,1,hidden);
}
}
else
{
if (hookPatches[px]&1)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];

```

```

    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];
    MAKE_QUAD_SPECIAL(p.v[0],s1,s2,e2end, p.smGroup,
matid,1,1,0,1,hidden);
    MAKE_QUAD_SPECIAL(e1,e2,p.v[1],e1start, p.smGroup,
matid,1,1,1,0,hidden);
    MAKE_TRI(p.v[2], e2start, e1end, p.smGroup, matid,1,0,1,hidden);
}
else if (hookPatches[px]&2)
{
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int end = hookEdgeEnd[index];
    int s1 = hookEdgeList[st];
    int s2 = hookEdgeList[st+1];
    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];
    MAKE_TRI(p.v[0], e0start, e2end, p.smGroup, matid,1,0,1,hidden);
    MAKE_QUAD_SPECIAL(p.v[1],s1,s2,e0end, p.smGroup,
matid,1,1,0,1,hidden);
    MAKE_QUAD_SPECIAL(e1,e2,p.v[2],e2start, p.smGroup,
matid,1,1,1,0,hidden);
}
else if (hookPatches[px]&4)
{
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int end = hookEdgeEnd[index];
    int s1 = hookEdgeList[st];
    int s2 = hookEdgeList[st+1];
    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];

```

```

    MAKE_TRI(p.v[1], e1start,e0end, p.smGroup, matid,1,0,1,hidden);
    MAKE_QUAD_SPECIAL(p.v[2],s1,s2,e1end, p.smGroup,
matid,1,1,0,1,hidden);
    MAKE_QUAD_SPECIAL(e1,e2,p.v[0],e0start, p.smGroup,
matid,1,1,1,0,hidden);
    }
    else
    {
    MAKE_TRI(p.v[0], e0start, e2end, p.smGroup, matid,1,0,1,hidden);
    MAKE_TRI(p.v[1], e1start, e0end, p.smGroup, matid,1,0,1,hidden);
    MAKE_TRI(p.v[2], e2start, e1end, p.smGroup, matid,1,0,1,hidden);
    }
    }
    // Create the edges, if necessary
    if(workSteps > 1) {
    // Corner-to-interiors
    //watje 12=10=98
    if (showInterior)
    {
    if (hookPatches[px]&1)
    {
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int end = hookEdgeEnd[index];
    int s1 = hookEdgeList[st];
    int s2 = hookEdgeList[st+1];
    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];
    MAKE_TRI(s2, i0, e2end, p.smGroup, matid,1,1,1,hidden);
    MAKE_TRI(e1start, i1, e1, p.smGroup, matid,1,1,1,hidden);
    MAKE_TRI(e2start, i2, e1end, p.smGroup, matid,1,1,1,hidden);

```

```

}
else if (hookPatches[px]&2)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];
int e1 = hookEdgeList[end-1];
int e2 = hookEdgeList[end];
MAKE_TRI(e0start, i0, e2end, p.smGroup, matid,1,1,1,hidden);
MAKE_TRI(s2, i1, e0end, p.smGroup, matid,1,1,1,hidden);
MAKE_TRI(e2start, i2, e1, p.smGroup, matid,1,1,1,hidden);
}
else if (hookPatches[px]&4)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];
int e1 = hookEdgeList[end-1];
int e2 = hookEdgeList[end];
MAKE_TRI(e0start, i0, e1, p.smGroup, matid,1,1,1,hidden);
MAKE_TRI(e1start, i1, e0end, p.smGroup, matid,1,1,1,hidden);
MAKE_TRI(s2, i2, e1end, p.smGroup, matid,1,1,1,hidden);
}
else
{
MAKE_TRI(e0start, i0, e2end, p.smGroup, matid,1,1,1,hidden);
MAKE_TRI(e1start, i1, e0end, p.smGroup, matid,1,1,1,hidden);
MAKE_TRI(e2start, i2, e1end, p.smGroup, matid,1,1,1,hidden);
}

```

```

    }
    }
else
{
if (hookPatches[px]&1)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];
int e1 = hookEdgeList[end-1];
int e2 = hookEdgeList[end];
MAKE_TRI(s2, i0, e2end, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(e1start, i1, e1, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(e2start, i2, e1end, p.smGroup, matid,0,0,0,hidden);
}
else if (hookPatches[px]&2)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];
int e1 = hookEdgeList[end-1];
int e2 = hookEdgeList[end];
MAKE_TRI(e0start, i0, e2end, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(s2, i1, e0end, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(e2start, i2, e1, p.smGroup, matid,0,0,0,hidden);
}
else if (hookPatches[px]&4)
{

```

```

int index = hookPoints[px];
int st = hookEdgeStart[index];
int end = hookEdgeEnd[index];
int s1 = hookEdgeList[st];
int s2 = hookEdgeList[st+1];
int e1 = hookEdgeList[end-1];
int e2 = hookEdgeList[end];
MAKE_TRI(e0start, i0, e1, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(e1start, i1, e0end, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(s2, i2, e1end, p.smGroup, matid,0,0,0,hidden);
}
else
{
MAKE_TRI(e0start, i0, e2end, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(e1start, i1, e0end, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(e2start, i2, e1end, p.smGroup, matid,0,0,0,hidden);
}
}
// Now the edges
int e0ix = e0start, e1ix = e1start, e2ix = e2start;
int i0ix = i0, i1ix = i1, i2ix = i2;
for(int i = 0; i < (workSteps - 1);
++i,e0ix+=e0dir,e1ix+=e1dir,e2ix+=e2dir) {
if (showInterior)
{
if (hookPatches[px]&1)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s1 = hookEdgeList[st+i*2+1];
int s2 = hookEdgeList[st+i*2+2];
int s3 = hookEdgeList[st+i*2+3];

```

```

    MAKE_QUAD_SPECIAL(s1,s2,s3,i0ix, p.smGroup,
matid,1,1,1,1,hidden);
    MAKE_TRI(e1ix, e1ix+e1dir, i1ix, p.smGroup, matid,1,1,1,hidden);
    MAKE_TRI(e2ix, e2ix+e2dir, i2ix, p.smGroup, matid,1,1,1,hidden);
}
else if (hookPatches[px]&2)
{
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int s1 = hookEdgeList[st+i*2+1];
    int s2 = hookEdgeList[st+i*2+2];
    int s3 = hookEdgeList[st+i*2+3];
    MAKE_TRI(e0ix, e0ix+e0dir, i0ix, p.smGroup, matid,1,1,1,hidden);
    MAKE_QUAD_SPECIAL(s1,s2,s3,i1ix, p.smGroup,
matid,1,1,1,1,hidden);
    MAKE_TRI(e2ix, e2ix+e2dir, i2ix, p.smGroup, matid,1,1,1,hidden);
}
else if (hookPatches[px]&4)
{
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int s1 = hookEdgeList[st+i*2+1];
    int s2 = hookEdgeList[st+i*2+2];
    int s3 = hookEdgeList[st+i*2+3];
    MAKE_TRI(e0ix, e0ix+e0dir, i0ix, p.smGroup, matid,1,1,1,hidden);
    MAKE_TRI(e1ix, e1ix+e1dir, i1ix, p.smGroup, matid,1,1,1,hidden);
    MAKE_QUAD_SPECIAL(s1,s2,s3,i2ix, p.smGroup,
matid,1,1,1,1,hidden);
}
else
{
    MAKE_TRI(e0ix, e0ix+e0dir, i0ix, p.smGroup, matid,1,1,1,hidden);
}

```

```

    MAKE_TRI(e1ix, e1ix+e1dir, i1ix, p.smGroup, matid,1,1,1,hidden);
    MAKE_TRI(e2ix, e2ix+e2dir, i2ix, p.smGroup, matid,1,1,1,hidden);
}

}
else {
    if (hookPatches[px]&1)
    {
        int index = hookPoints[px];
        int st = hookEdgeStart[index];
        int s1 = hookEdgeList[st+i*2+1];
        int s2 = hookEdgeList[st+i*2+2];
        int s3 = hookEdgeList[st+i*2+3];
        MAKE_QUAD_SPECIAL(s1,s2,s3,i0ix, p.smGroup,
matid,1,1,0,0,hidden);
        MAKE_TRI(e1ix, e1ix+e1dir, i1ix, p.smGroup, matid,1,0,0,hidden);
        MAKE_TRI(e2ix, e2ix+e2dir, i2ix, p.smGroup, matid,1,0,0,hidden);
    }
    else if (hookPatches[px]&2)
    {
        int index = hookPoints[px];
        int st = hookEdgeStart[index];
        int s1 = hookEdgeList[st+i*2+1];
        int s2 = hookEdgeList[st+i*2+2];
        int s3 = hookEdgeList[st+i*2+3];
        MAKE_TRI(e0ix, e0ix+e0dir, i0ix, p.smGroup, matid,1,0,0,hidden);
        MAKE_QUAD_SPECIAL(s1,s2,s3,i1ix, p.smGroup,
matid,1,1,0,0,hidden);
        MAKE_TRI(e2ix, e2ix+e2dir, i2ix, p.smGroup, matid,1,0,0,hidden);
    }
    else if (hookPatches[px]&4)
    {

```

```

    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int s1 = hookEdgeList[st+i*2+1];
    int s2 = hookEdgeList[st+i*2+2];
    int s3 = hookEdgeList[st+i*2+3];
    MAKE_TRI(e0ix, e0ix+e0dir, i0ix, p.smGroup, matid,1,0,0,hidden);
    MAKE_TRI(e1ix, e1ix+e1dir, i1ix, p.smGroup, matid,1,0,0,hidden);
    MAKE_QUAD_SPECIAL(s1,s2,s3,i2ix, p.smGroup,
matid,1,1,0,0,hidden);
    }
    else
    {
        MAKE_TRI(e0ix, e0ix+e0dir, i0ix, p.smGroup, matid,1,0,0,hidden);
        MAKE_TRI(e1ix, e1ix+e1dir, i1ix, p.smGroup, matid,1,0,0,hidden);
        MAKE_TRI(e2ix, e2ix+e2dir, i2ix, p.smGroup, matid,1,0,0,hidden);
    }
}
i0ix++;
i1ix += (workSteps - i - 2);
i2ix -= (i + 2);
}
// Make interleaving triangles
e0ix = e0start + e0dir;
e1ix = e1start + e1dir;
e2ix = e2start + e2dir;
i0ix = i0;
i1ix = i1;
i2ix = i2;
for(i = 1; i < (workSteps - 1); ++i,e0ix+=e0dir,e1ix+=e1dir,e2ix+=e2dir) {
    if (showInterior)
    {
        if (hookPatches[px]&1)

```

```

{
  int index = hookPoints[px];
  int st = hookEdgeStart[index];
  int s1 = hookEdgeList[st+i*2+1];
  MAKE_TRI(s1, i0ix + 1, i0ix, p.smGroup, matid,1,1,1,hidden);
  MAKE_TRI(e1ix, i1ix + (workSteps - i - 1), i1ix, p.smGroup,
matid,1,1,1,hidden);
  MAKE_TRI(e2ix, i2ix - (i + 1), i2ix, p.smGroup, matid,1,1,1,hidden);
}
else if (hookPatches[px]&2)
{
  int index = hookPoints[px];
  int st = hookEdgeStart[index];
  int s1 = hookEdgeList[st+i*2+1];
  MAKE_TRI(e0ix, i0ix + 1, i0ix, p.smGroup, matid,1,1,1,hidden);
  MAKE_TRI(s1, i1ix + (workSteps - i - 1), i1ix, p.smGroup,
matid,1,1,1,hidden);
  MAKE_TRI(e2ix, i2ix - (i + 1), i2ix, p.smGroup, matid,1,1,1,hidden);
}
else if (hookPatches[px]&4)
{
  int index = hookPoints[px];
  int st = hookEdgeStart[index];
  int s1 = hookEdgeList[st+i*2+1];
  MAKE_TRI(e0ix, i0ix + 1, i0ix, p.smGroup, matid,1,1,1,hidden);
  MAKE_TRI(e1ix, i1ix + (workSteps - i - 1), i1ix, p.smGroup,
matid,1,1,1,hidden);
  MAKE_TRI(s1, i2ix - (i + 1), i2ix, p.smGroup, matid,1,1,1,hidden);
}
else
{
  MAKE_TRI(e0ix, i0ix + 1, i0ix, p.smGroup, matid,1,1,1,hidden);
}

```

```

    MAKE_TRI(e1ix, i1ix + (workSteps - i - 1), i1ix, p.smGroup,
matid,1,1,1,hidden);
    MAKE_TRI(e2ix, i2ix - (i + 1), i2ix, p.smGroup, matid,1,1,1,hidden);
}
}
else
{
if (hookPatches[px]&1)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s1 = hookEdgeList[st+i*2+1];
MAKE_TRI(s1, i0ix + 1, i0ix, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(e1ix, i1ix + (workSteps - i - 1), i1ix, p.smGroup,
matid,0,0,0,hidden);
MAKE_TRI(e2ix, i2ix - (i + 1), i2ix, p.smGroup, matid,0,0,0,hidden);
}
else if (hookPatches[px]&2)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s1 = hookEdgeList[st+i*2+1];
MAKE_TRI(e0ix, i0ix + 1, i0ix, p.smGroup, matid,0,0,0,hidden);
MAKE_TRI(s1, i1ix + (workSteps - i - 1), i1ix, p.smGroup,
matid,0,0,0,hidden);
MAKE_TRI(e2ix, i2ix - (i + 1), i2ix, p.smGroup, matid,0,0,0,hidden);
}
else if (hookPatches[px]&4)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s1 = hookEdgeList[st+i*2+1];

```

```

    MAKE_TRI(e0ix, i0ix + 1, i0ix, p.smGroup, matid,0,0,0,hidden);
    MAKE_TRI(e1ix, i1ix + (workSteps - i - 1), i1ix, p.smGroup,
matid,0,0,0,hidden);
    MAKE_TRI(s1, i2ix - (i + 1), i2ix, p.smGroup, matid,0,0,0,hidden);
}
else
{
    MAKE_TRI(e0ix, i0ix + 1, i0ix, p.smGroup, matid,0,0,0,hidden);
    MAKE_TRI(e1ix, i1ix + (workSteps - i - 1), i1ix, p.smGroup,
matid,0,0,0,hidden);
    MAKE_TRI(e2ix, i2ix - (i + 1), i2ix, p.smGroup, matid,0,0,0,hidden);
}
}
i0ix++;
i1ix += (workSteps - i - 1);
i2ix -= (i + 1);
}
}
else
{
    if (showInterior)
    {
        if (hookPatches[px]&1)
        {
            int index = hookPoints[px];
            int st = hookEdgeStart[index];
            int s2 = hookEdgeList[st+1];
            MAKE_TRI(s2, e1start, e2start, p.smGroup, matid,1,1,1,hidden);
        }
        else if (hookPatches[px]&2)
        {
            int index = hookPoints[px];

```

```

int st = hookEdgeStart[index];
int s2 = hookEdgeList[st+1];
MAKE_TRI(e0start, s2, e2start, p.smGroup, matid,1,1,1,hidden);
}
else if (hookPatches[px]&4)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s2 = hookEdgeList[st+1];
MAKE_TRI(e0start, e1start, s2, p.smGroup, matid,1,1,1,hidden);
}
else
{
MAKE_TRI(e0start, e1start, e2start, p.smGroup, matid,1,1,1,hidden);
}
}
else
{
if (hookPatches[px]&1)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s2 = hookEdgeList[st+1];
MAKE_TRI(s2, e1start, e2start, p.smGroup, matid,0,0,0,hidden);
}
else if (hookPatches[px]&2)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s2 = hookEdgeList[st+1];
MAKE_TRI(e0start, s2, e2start, p.smGroup, matid,0,0,0,hidden);
}
}
}

```



```

    }
//b edge
    if (hookPatches[px]&2)
    {
        int edge = p.edge[1];
        int ex = hooks[hookPoints[px]].hookPoint;
        MAKE_PENTA_EDGE2(p.v[0], p.v[1], ex, p.v[2], p.v[3], p.smGroup,
matid,1,1,1,1,hidden);
    }
//c edge
    if (hookPatches[px]&4)
    {
        int edge = p.edge[2];
        int ex = hooks[hookPoints[px]].hookPoint;
        MAKE_PENTA_EDGE3(p.v[0], p.v[1], p.v[2],ex, p.v[3], p.smGroup,
matid,1,1,1,1,hidden);
    }
//d edge
    if (hookPatches[px]&8)
    {
        int edge = p.edge[3];
        int ex = hooks[hookPoints[px]].hookPoint;
        MAKE_PENTA_EDGE4(p.v[0], p.v[1], p.v[2], p.v[3], ex, p.smGroup,
matid,1,1,1,1,hidden);
    }
    }
    break;
}
if(workSteps > 1) {
    // Make the inside faces if necessary
    for(int u = 0; u < (workSteps-1); ++u) {
        int uix = baseVert + u * workSteps;

```

```

    for(int v = 0; v < (workSteps-1); ++v, ++uix) {
//watje 12-8-98
    if (showInterior)
    {
        MAKE_QUAD(uix, uix + 1, uix + workSteps + 1, uix + workSteps,
p.smGroup, matid,1,1,0,1,1,0,hidden);
    }
    else
    {
        MAKE_QUAD(uix, uix + 1, uix + workSteps + 1, uix + workSteps,
p.smGroup, matid,0,0,0,0,0,0,hidden);
    }
    }
    }
// Make the corner faces
int edge0 = p.edge[0];
int edge1 = p.edge[1];
int edge2 = p.edge[2];
int edge3 = p.edge[3];
PatchEdge &e0 = edges[edge0];
PatchEdge &e1 = edges[edge1];
PatchEdge &e2 = edges[edge2];
PatchEdge &e3 = edges[edge3];
int
e0start,e1start,e2start,e3start,e0end,e1end,e2end,e3end,e0dir,e1dir,e2dir,e3di
if(e0.v1 == p.v[0]) {
    e0start = edgeStart[edge0];
    e0end = edgeEnd[edge0];
    e0dir = 1;
}
else {

```

```
e0start = edgeEnd[edge0];
e0end = edgeStart[edge0];
e0dir = -1;
}
if(e1.v1 == p.v[1]) {
e1start = edgeStart[edge1];
e1end = edgeEnd[edge1];
e1dir = 1;
}
else {
e1start = edgeEnd[edge1];
e1end = edgeStart[edge1];
e1dir = -1;
}
if(e2.v1 == p.v[2]) {
e2start = edgeStart[edge2];
e2end = edgeEnd[edge2];
e2dir = 1;
}
else {
e2start = edgeEnd[edge2];
e2end = edgeStart[edge2];
e2dir = -1;
}
if(e3.v1 == p.v[3]) {
e3start = edgeStart[edge3];
e3end = edgeEnd[edge3];
e3dir = 1;
}
else {
e3start = edgeEnd[edge3];
e3end = edgeStart[edge3];
```

```

    e3dir = -1;
}
if (e3start > e3end) e3dir = -1;
else e3dir = 1;
if (e2start > e2end) e2dir = -1;
else e2dir = 1;
if (e1start > e1end) e1dir = -1;
else e1dir = 1;
if (e0start > e0end) e0dir = -1;
else e0dir = 1;
// Create the corners
int i0 = baseVert;
int i1 = baseVert + (workSteps - 1);
int i2 = baseVert + quadint - 1;
int i3 = i2 - (workSteps - 1);
//watje 12-8-98
if (showInterior)
{
    if (hookPatches[px]&1)
    {
        int index = hookPoints[px];
        int st = hookEdgeStart[index];
        int end = hookEdgeEnd[index];
        int s1 = hookEdgeList[st];
        int s2 = hookEdgeList[st+1];
        int e1 = hookEdgeList[end-1];
        int e2 = hookEdgeList[end];
        MAKE_PENTA_EDGE1(p.v[0],s1,s2,i0,e3end, p.smGroup,
matid,1,1,1,1,hidden);
        MAKE_PENTA_EDGE1(e1,e2,p.v[1],e1start,i1, p.smGroup,
matid,1,1,1,1,hidden);
        MAKE_QUAD(p.v[2], e2start, i2, e1end, p.smGroup,

```

```

matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(e3start, i3, e2end, p.v[3], p.smGroup,
matid,1,1,0,1,1,0,hidden);
    }
    else if (hookPatches[px]&2)
    {
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int end = hookEdgeEnd[index];
    int s1 = hookEdgeList[st];
    int s2 = hookEdgeList[st+1];
    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];
    MAKE_QUAD(p.v[0], e0start, i0, e3end, p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_PENTA_EDGE2(e0end, p.v[1],s1,s2,i1, p.smGroup,
matid,1,1,1,1,hidden);
    MAKE_PENTA_EDGE2(i2,e1, e2, p.v[2],e2start, p.smGroup,
matid,1,1,1,1,hidden);
    MAKE_QUAD(e3start, i3, e2end, p.v[3], p.smGroup,
matid,1,1,0,1,1,0,hidden);
    }
    else if (hookPatches[px]&4)
    {
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int end = hookEdgeEnd[index];
    int s1 = hookEdgeList[st];
    int s2 = hookEdgeList[st+1];
    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];
    MAKE_QUAD(p.v[0], e0start, i0, e3end, p.smGroup,
matid,1,1,0,1,1,0,hidden);

```

```

    MAKE_QUAD(e1start, i1, e0end, p.v[1], p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_PENTA_EDGE3(i2,e1end,p.v[2],s1, s2, p.smGroup,
matid,1,1,1,1,hidden);
    MAKE_PENTA_EDGE3(e3start,i3,e1,e2,p.v[3], p.smGroup,
matid,1,1,1,1,hidden);
}
else if (hookPatches[px]&8)
{
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int end = hookEdgeEnd[index];
    int s1 = hookEdgeList[st];
    int s2 = hookEdgeList[st+1];
    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];
    MAKE_QUAD(e1start, i1, e0end, p.v[1], p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(p.v[2], e2start, i2, e1end, p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_PENTA_EDGE4(s2,i3,e2end,p.v[3],s1, p.smGroup,
matid,1,1,1,1,hidden);
    MAKE_PENTA_EDGE4(p.v[0],e0start,i0,e1,e2, p.smGroup,
matid,1,1,1,1,hidden);
}
else
{
    MAKE_QUAD(p.v[0], e0start, i0, e3end, p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(e1start, i1, e0end, p.v[1], p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(p.v[2], e2start, i2, e1end, p.smGroup,
matid,1,1,0,1,1,0,hidden);

```

```

    MAKE_QUAD(e3start, i3, e2end, p.v[3], p.smGroup,
matid,1,1,0,1,1,0,hidden);
    }
    }
else
{
    if (hookPatches[px]&1)
    {
        int index = hookPoints[px];
        int st = hookEdgeStart[index];
        int end = hookEdgeEnd[index];
        int s1 = hookEdgeList[st];
        int s2 = hookEdgeList[st+1];
        int e1 = hookEdgeList[end-1];
        int e2 = hookEdgeList[end];
        MAKE_PENTA_EDGE1(p.v[0],s1,s2,i0,e3end, p.smGroup,
matid,1,0,0,1,hidden);
        MAKE_PENTA_EDGE1(e1,e2,p.v[1],e1start,i1, p.smGroup,
matid,1,1,0,0,hidden);
        MAKE_QUAD(p.v[2], e2start, i2, e1end, p.smGroup,
matid,1,0,0,0,1,0,hidden);
        MAKE_QUAD(e3start, i3, e2end, p.v[3], p.smGroup,
matid,0,0,0,1,1,0,hidden);
    }
    else if (hookPatches[px]&2)
    {
        int index = hookPoints[px];
        int st = hookEdgeStart[index];
        int end = hookEdgeEnd[index];
        int s1 = hookEdgeList[st];
        int s2 = hookEdgeList[st+1];
        int e1 = hookEdgeList[end-1];
        int e2 = hookEdgeList[end];
    }
}

```

```

    MAKE_QUAD(p.v[0], e0start, i0, e3end, p.smGroup,
matid,1,0,0,0,1,0,hidden);
    MAKE_PENTA_EDGE2(e0end, p.v[1],s1,s2,i1, p.smGroup,
matid,1,1,0,0,hidden);
    MAKE_PENTA_EDGE2(i2,e1, e2, p.v[2],e2start, p.smGroup,
matid,0,1,1,0,hidden);
    MAKE_QUAD(e3start, i3, e2end, p.v[3], p.smGroup,
matid,0,0,0,1,1,0,hidden);
}
else if (hookPatches[px]&4)
{
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int end = hookEdgeEnd[index];
    int s1 = hookEdgeList[st];
    int s2 = hookEdgeList[st+1];
    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];
    MAKE_QUAD(p.v[0], e0start, i0, e3end, p.smGroup,
matid,1,0,0,0,1,0,hidden);
    MAKE_QUAD(e1start, i1, e0end, p.v[1], p.smGroup,
matid,0,0,0,1,1,0,hidden);
    MAKE_PENTA_EDGE3(i2,e1end,p.v[2],s1, s2, p.smGroup,
matid,0,1,1,0,hidden);
    MAKE_PENTA_EDGE3(e3start,i3,e1,e2,p.v[3], p.smGroup,
matid,0,0,1,1,hidden);
}
else if (hookPatches[px]&8)
{
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int end = hookEdgeEnd[index];
    int s1 = hookEdgeList[st];

```

```

    int s2 = hookEdgeList[st+1];
    int e1 = hookEdgeList[end-1];
    int e2 = hookEdgeList[end];
    MAKE_QUAD(e1start, i1, e0end, p.v[1], p.smGroup,
matid,0,0,0,1,1,0,hidden);
    MAKE_QUAD(p.v[2], e2start, i2, e1end, p.smGroup,
matid,1,0,0,0,1,0,hidden);
    MAKE_PENTA_EDGE4(s2,i3,e2end,p.v[3],s1, p.smGroup,
matid,0,0,1,1,hidden);
    MAKE_PENTA_EDGE4(p.v[0],e0start,i0,e1,e2, p.smGroup,
matid,1,0,0,1,hidden);
}
else
{
    MAKE_QUAD(p.v[0], e0start, i0, e3end, p.smGroup,
matid,1,0,0,0,1,0,hidden);
    MAKE_QUAD(e1start, i1, e0end, p.v[1], p.smGroup,
matid,0,0,0,1,1,0,hidden);
    MAKE_QUAD(p.v[2], e2start, i2, e1end, p.smGroup,
matid,1,0,0,0,1,0,hidden);
    MAKE_QUAD(e3start, i3, e2end, p.v[3], p.smGroup,
matid,0,0,0,1,1,0,hidden);
}
}
// Create the edges, if necessary
if(workSteps > 1) {
    int e0ix = e0start, e1ix = e1start, e2ix = e2start, e3ix = e3start;
    for(int i = 0; i < (workSteps - 1);
++i,e0ix+=e0dir,e1ix+=e1dir,e2ix+=e2dir,e3ix+=e3dir) {
//watje 12-8-98
    if (showInterior)
    {
        if (hookPatches[px]&1)

```

```

{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s1 = hookEdgeList[st+i*2+1];
int s2 = hookEdgeList[st+i*2+2];
int s3 = hookEdgeList[st+i*2+3];
MAKE_PENTA_EDGE1(s1,s2,s3, i0 + 1 + i, i0 + i, p.smGroup,
matid,1,1,1,1,hidden);
MAKE_QUAD(e1ix+e1dir, i1 + workSteps * (i + 1), i1 + workSteps * i,
e1ix, p.smGroup, matid,1,1,0,1,1,0,hidden);
MAKE_QUAD(e2ix, e2ix+e2dir, i2 - 1 - i, i2 - i, p.smGroup,
matid,1,1,0,1,1,0,hidden);
MAKE_QUAD(e3ix+e3dir, i3 - workSteps * (i + 1), i3 - workSteps * i,
e3ix, p.smGroup, matid,1,1,0,1,1,0,hidden);
}
else if (hookPatches[px]&2)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s1 = hookEdgeList[st+i*2+1];
int s2 = hookEdgeList[st+i*2+2];
int s3 = hookEdgeList[st+i*2+3];
MAKE_QUAD(e0ix, e0ix+e0dir, i0 + 1 + i, i0 + i, p.smGroup,
matid,1,1,0,1,1,0,hidden);
MAKE_PENTA_EDGE2(i1 + workSteps * i,s1,s2,s3, i1 + workSteps *
(i + 1), p.smGroup, matid,1,1,1,1,hidden);
MAKE_QUAD(e2ix, e2ix+e2dir, i2 - 1 - i, i2 - i, p.smGroup,
matid,1,1,0,1,1,0,hidden);
MAKE_QUAD(e3ix+e3dir, i3 - workSteps * (i + 1), i3 - workSteps * i,
e3ix, p.smGroup, matid,1,1,0,1,1,0,hidden);
}
else if (hookPatches[px]&4)
{

```

```

    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int s1 = hookEdgeList[st+i*2+1];
    int s2 = hookEdgeList[st+i*2+2];
    int s3 = hookEdgeList[st+i*2+3];
    MAKE_QUAD(e0ix, e0ix+e0dir, i0 + 1 + i, i0 + i, p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(e1ix+e1dir, i1 + workSteps * (i + 1), i1 + workSteps * i,
e1ix, p.smGroup, matid,1,1,0,1,1,0,hidden);
    MAKE_PENTA_EDGE3(i2 - 1 - i,i2 - i,s1,s2,s3, p.smGroup,
matid,1,1,1,1,hidden);
    MAKE_QUAD(e3ix+e3dir, i3 - workSteps * (i + 1), i3 - workSteps * i,
e3ix, p.smGroup, matid,1,1,0,1,1,0,hidden);
}
else if (hookPatches[px]&8)
{
    int index = hookPoints[px];
    int st = hookEdgeStart[index];
    int s1 = hookEdgeList[st+i*2+1];
    int s2 = hookEdgeList[st+i*2+2];
    int s3 = hookEdgeList[st+i*2+3];
    MAKE_QUAD(e0ix, e0ix+e0dir, i0 + 1 + i, i0 + i, p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(e1ix+e1dir, i1 + workSteps * (i + 1), i1 + workSteps * i,
e1ix, p.smGroup, matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(e2ix, e2ix+e2dir, i2 - 1 - i, i2 - i, p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_PENTA_EDGE4(s3,i3 - workSteps * (i + 1), i3 - workSteps *
i,s1,s2, p.smGroup, matid,1,1,1,1,hidden);
}
else
{
    MAKE_QUAD(e0ix, e0ix+e0dir, i0 + 1 + i, i0 + i, p.smGroup,

```

```

matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(e1ix+e1dir, i1 + workSteps * (i + 1), i1 + workSteps * i,
e1ix, p.smGroup, matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(e2ix, e2ix+e2dir, i2 - 1 - i, i2 - i, p.smGroup,
matid,1,1,0,1,1,0,hidden);
    MAKE_QUAD(e3ix+e3dir, i3 - workSteps * (i + 1), i3 - workSteps * i,
e3ix, p.smGroup, matid,1,1,0,1,1,0,hidden);
    }
    }
else
{
    if (hookPatches[px]&1)
    {
        int index = hookPoints[px];
        int st = hookEdgeStart[index];
        int s1 = hookEdgeList[st+i*2+1];
        int s2 = hookEdgeList[st+i*2+2];
        int s3 = hookEdgeList[st+i*2+3];

        MAKE_PENTA_EDGE1(s1,s2,s3, i0 + 1 + i, i0 + i, p.smGroup,
matid,1,0,0,0,hidden);
        MAKE_QUAD(e1ix+e1dir, i1 + workSteps * (i + 1), i1 + workSteps * i,
e1ix, p.smGroup, matid,0,0,0,0,1,0,hidden);
        MAKE_QUAD(e2ix, e2ix+e2dir, i2 - 1 - i, i2 - i, p.smGroup,
matid,1,0,0,0,0,0,hidden);
        MAKE_QUAD(e3ix+e3dir, i3 - workSteps * (i + 1), i3 - workSteps * i,
e3ix, p.smGroup, matid,0,0,0,0,1,0,hidden);
    }
    else if (hookPatches[px]&2)
    {
        int index = hookPoints[px];
        int st = hookEdgeStart[index];
        int s1 = hookEdgeList[st+i*2+1];

```

```

int s2 = hookEdgeList[st+i*2+2];
int s3 = hookEdgeList[st+i*2+3];

MAKE_QUAD(e0ix, e0ix+e0dir, i0 + 1 + i, i0 + i, p.smGroup,
matid,1,0,0,0,0,hidden);
MAKE_PENTA_EDGE2(i1 + workSteps * i,s1,s2,s3, i1 + workSteps *
(i + 1), p.smGroup, matid,0,1,0,0,hidden);
MAKE_QUAD(e2ix, e2ix+e2dir, i2 - 1 - i, i2 - i, p.smGroup,
matid,1,0,0,0,0,hidden);
MAKE_QUAD(e3ix+e3dir, i3 - workSteps * (i + 1), i3 - workSteps * i,
e3ix, p.smGroup, matid,0,0,0,0,1,0,hidden);
}
else if (hookPatches[px]&4)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];
int s1 = hookEdgeList[st+i*2+1];
int s2 = hookEdgeList[st+i*2+2];
int s3 = hookEdgeList[st+i*2+3];

MAKE_QUAD(e0ix, e0ix+e0dir, i0 + 1 + i, i0 + i, p.smGroup,
matid,1,0,0,0,0,hidden);
MAKE_QUAD(e1ix+e1dir, i1 + workSteps * (i + 1), i1 + workSteps * i,
e1ix, p.smGroup, matid,0,0,0,0,1,0,hidden);
MAKE_PENTA_EDGE3(i2 - 1 - i,i2 - i,s1,s2,s3, p.smGroup,
matid,0,0,1,0,hidden);
MAKE_QUAD(e3ix+e3dir, i3 - workSteps * (i + 1), i3 - workSteps * i,
e3ix, p.smGroup, matid,0,0,0,0,1,0,hidden);
}
else if (hookPatches[px]&8)
{
int index = hookPoints[px];
int st = hookEdgeStart[index];

```



```

if(getNumMapVerts (chan)) {
  mesh.setMapSupport (chan, TRUE);
  mesh.setNumMapFaces (chan, nFaces);
  int tface = 0;
  for(px = 0; px < numPatches; ++px) {
    Patch &p = patches[px];
    int baseTVert = (*patchTVOff[chan])[px]; // texture vertex base
    switch(p.type) {
      case PATCH_TRI: {
        int tvbase = baseTVert;
        // If only need to make 1 triangle, make it and blow!
        if(workSteps == 0) {
          if (hookPatches[px]==0)
            {
              MAKE_TTRI(chan, tvbase, tvbase+1, tvbase+2);
            }
          else
            {
              if (hookPatches[px]&1)
                {
                  MAKE_TQUAD_SPECIAL(chan, tvbase,tvertHookStart,
tvbase+1,tvbase+2);
                }
              else if (hookPatches[px]&2)
                {
                  MAKE_TQUAD_SPECIAL(chan,
tvbase+1,tvertHookStart,tvbase+2,tvbase );
                }
              else if (hookPatches[px]&4)
                {
                  MAKE_TQUAD_SPECIAL(chan,

```

```

tvbase+2,tvertHookStart,tvbase,tvbase+1 );
    }
    }
    break;
    }
    if(workSteps > 2) {
        // Make inside faces if necessary
        int ivbase = baseTVert + workSteps + 3;
        for(int ax = 0; ax < (workSteps - 2); ++ax) {
            int vix = ivbase;
            for(int bx = 0; bx < (workSteps - 2 - ax); ++bx, ++vix)
                MAKE_TTRI(chan, vix, vix + 1, vix + workSteps + 1 - ax);
            vix = ivbase;
            for(bx = 1; bx < (workSteps - 2 - ax); ++bx, ++vix)
                MAKE_TTRI(chan, vix + 1, vix + workSteps + 2 - ax, vix + workSteps
+ 1 - ax);
            ivbase += (workSteps + 1 - ax);
        }
    }
    // Create the corners
    int i0 = baseTVert + patchDivs + 2;
    int i1 = i0 + workSteps - 2;
    int i2 = baseTVert + triTVerts - 5;
    int e0start = baseTVert + 1;
    int e1start = baseTVert + patchDivs * 2;
    int e2start = baseTVert + triTVerts - 3;
    int e0end = baseTVert + workSteps;
    int e1end = e2start + 1;
    int e2end = baseTVert + patchDivs + 1;
    if (hookPatches[px]&1)
    {
        MAKE_TQUAD_SPECIAL(chan, baseTVert, tvertHookStart, e0start,

```

```

e2end);
    MAKE_TQUAD_SPECIAL(chan, e0end, tvertHookStart, baseTVert +
patchDivs, e1start);
    MAKE_TTRI(chan, baseTVert + triTVerts-1, e2start, e1end);
}
else if (hookPatches[px]&2)
{
    MAKE_TTRI(chan, baseTVert, e0start, e2end);
    MAKE_TQUAD_SPECIAL(chan, baseTVert + patchDivs,
tvertHookStart, e1start, e0end);
    MAKE_TQUAD_SPECIAL(chan, e1end,tvertHookStart , baseTVert +
triTVerts-1, e2start);
}
else if (hookPatches[px]&4)
{
    MAKE_TTRI(chan, baseTVert + patchDivs, e1start, e0end);
    MAKE_TQUAD_SPECIAL(chan, baseTVert + triTVerts-
1,tvertHookStart, e2start, e1end);
    MAKE_TQUAD_SPECIAL(chan, e2end,tvertHookStart, baseTVert,
e0start);
}
else
{
    MAKE_TTRI(chan, baseTVert, e0start, e2end);
    MAKE_TTRI(chan, baseTVert + patchDivs, e1start, e0end);
    MAKE_TTRI(chan, baseTVert + triTVerts-1, e2start, e1end);
}
// Create the edges, if necessary
if(workSteps > 1) {
    // Corner-to-interiors
    MAKE_TTRI(chan, e0start, i0, e2end);
    MAKE_TTRI(chan, e1start, i1, e0end);
    MAKE_TTRI(chan, e2start, i2, e1end);

```

```

// Now the edges
int e0ix = e0start, e1ix = e1start, e2ix = e2start;
int i0ix = i0, i1ix = i1, i2ix = i2;
int holder = tvertHookStart;
for(int i = 0; i < (workSteps - 1); ++i) {
    int e1dir = workSteps - i;
    int e2dir = -(3 + i);
    if (hookPatches[px]&1)
    {
        MAKE_TQUAD_SPECIAL(chan, e0ix, tvertHookStart, e0ix+1, i0ix);
        MAKE_TTRI(chan, e1ix, e1ix+e1dir, i1ix);
        MAKE_TTRI(chan, e2ix, e2ix+e2dir, i2ix);
    }
    else if (hookPatches[px]&2)
    {
        MAKE_TTRI(chan, e0ix, e0ix+1, i0ix);
        MAKE_TQUAD_SPECIAL(chan, e1ix, tvertHookStart, e1ix+e1dir,
i1ix);
        MAKE_TTRI(chan, e2ix, e2ix+e2dir, i2ix);
    }
    else if (hookPatches[px]&4)
    {
        MAKE_TTRI(chan, e0ix, e0ix+1, i0ix);
        MAKE_TTRI(chan, e1ix, e1ix+e1dir, i1ix);
        MAKE_TQUAD_SPECIAL(chan, e2ix, tvertHookStart, e2ix+e2dir,
i2ix);
    }
    else
    {
        MAKE_TTRI(chan, e0ix, e0ix+1, i0ix);
        MAKE_TTRI(chan, e1ix, e1ix+e1dir, i1ix);
        MAKE_TTRI(chan, e2ix, e2ix+e2dir, i2ix);
    }
}

```

```

    }
    i0ix++;
    i1ix += e1dir;
    i2ix -= (4 + i);
    e0ix++;
    e1ix += e1dir;
    e2ix += e2dir;
    }
// Make interleaving triangles
e0ix = e0start + 1;
e1ix = e1start + workSteps;
e2ix = e2start - 3;
i0ix = i0;
i1ix = i1;
i2ix = i2;
for(i = 1; i < (workSteps - 1); ++i) {
    int e1dir = workSteps - i;
    int e2dir = -(3 + i);
    MAKE_TTRI(chan, e0ix, i0ix + 1, i0ix);
    MAKE_TTRI(chan, e1ix, e1ix - 1, i1ix);
    MAKE_TTRI(chan, e2ix, i2ix + e2dir, i2ix);
    i0ix++;
    i1ix = i1ix + e1dir + 1;
    i2ix += e2dir;
    e0ix++;
    e1ix += e1dir;
    e2ix += e2dir;
    }
}
else
    MAKE_TTRI(chan, e0start, e1start, e2start);
}

```

```

break;
case PATCH_QUAD: {
int tvbase = baseTVert;
int qvstep = patchDivs + 1;
// If only need to make 1 quad, make it and blow!
if(workSteps == 0) {
if (hookPatches[px]==0)
{
MAKE_TQUAD(chan, tvbase, tvbase+1, tvbase+3, tvbase+2);
}
else
{
if (hookPatches[px]&1)
{
MAKE_TPENTA_EDGE1(chan, tvbase,tvertHookStart, tvbase+1,
tvbase+3, tvbase+2);
}
else if (hookPatches[px]&2)
{
MAKE_TPENTA_EDGE2(chan, tvbase, tvbase+1,tvertHookStart,
tvbase+3, tvbase+2);
}
else if (hookPatches[px]&4)
{
MAKE_TPENTA_EDGE3(chan, tvbase, tvbase+1,
tvbase+3,tvertHookStart, tvbase+2);
}
else if (hookPatches[px]&8)
{
MAKE_TPENTA_EDGE4(chan, tvbase, tvbase+1, tvbase+3,
tvbase+2,tvertHookStart);
}
}
}
}

```

```

    }
    break;
}
if(workSteps > 1) {
    int ivbase = baseTVert + qvstep + 1;
    // Make the inside faces if necessary
    for(int v = 0; v < (workSteps-1); ++v) {
        int vix = ivbase + v * qvstep;
        for(int u = 0; u < (workSteps-1); ++u, ++vix) {
            MAKE_TQUAD(chan, vix, vix + 1, vix + qvstep + 1, vix + qvstep);
        }
    }
}
int e0start = baseTVert + 1;
int e1start = baseTVert + qvstep * 2 - 1;
int e2start = baseTVert + quadTVerts - 2;
int e3start = baseTVert + qvstep * workSteps;
int e0end = e0start + workSteps - 1;
int e1end = e1start + qvstep * (workSteps - 1);
int e2end = e2start - (workSteps - 1);
int e3end = e3start - qvstep * (workSteps - 1);
// Create the corners
int i0 = baseTVert + qvstep + 1;
int i1 = i0 + (workSteps - 1);
int i2 = i1 + qvstep * (workSteps - 1);
int i3 = i2 - (workSteps - 1);
if (hookPatches[px]&1)
{
    MAKE_TPENTA_EDGE1(chan, baseTVert,tvertHookStart, e0start, i0,
e3end);
    MAKE_TPENTA_EDGE1(chan, e0end, tvertHookStart, baseTVert +
patchDivs,e1start, i1);

```

```

    MAKE_TQUAD(chan, baseTVert + quadTVerts - 1, e2start, i2, e1end);
    MAKE_TQUAD(chan, e3start, i3, e2end, baseTVert + qvstep *
patchDivs);
    }
    else if (hookPatches[px]&2)
    {
        MAKE_TQUAD(chan, baseTVert, e0start, i0, e3end);
        MAKE_TPENTA_EDGE2(chan, e0end, baseTVert +
patchDivs,tvertHookStart,e1start, i1);
        MAKE_TPENTA_EDGE2(chan, i2, e1end,tvertHookStart, baseTVert +
quadTVerts - 1, e2start);
        MAKE_TQUAD(chan, e3start, i3, e2end, baseTVert + qvstep *
patchDivs);
    }
    else if (hookPatches[px]&4)
    {
        MAKE_TQUAD(chan, baseTVert, e0start, i0, e3end);
        MAKE_TQUAD(chan, e1start, i1, e0end, baseTVert + patchDivs);
        MAKE_TPENTA_EDGE3(chan, i2, e1end, baseTVert + quadTVerts -
1,tvertHookStart, e2start);
        MAKE_TPENTA_EDGE3(chan, e3start, i3, e2end,tvertHookStart,
baseTVert + qvstep * patchDivs);
    }
    else if (hookPatches[px]&8)
    {
        MAKE_TQUAD(chan, e1start, i1, e0end, baseTVert + patchDivs);
        MAKE_TQUAD(chan, baseTVert + quadTVerts - 1, e2start, i2, e1end);
        MAKE_TPENTA_EDGE4(chan, e3start, i3, e2end, baseTVert + qvstep
* patchDivs,tvertHookStart);
        MAKE_TPENTA_EDGE4(chan, baseTVert, e0start, i0,
e3end,tvertHookStart);
    }
    else

```

```

{
MAKE_TQUAD(chan, baseTVert, e0start, i0, e3end);
MAKE_TQUAD(chan, e1start, i1, e0end, baseTVert + patchDivs);
MAKE_TQUAD(chan, baseTVert + quadTVerts - 1, e2start, i2, e1end);
MAKE_TQUAD(chan, e3start, i3, e2end, baseTVert + qvstep *
patchDivs);
}
// Create the edges, if necessary
if(workSteps > 1) {
int e0ix = e0start, e1ix = e1start, e2ix = e2start, e3ix = e3start;
for(int i = 0; i < (workSteps - 1); ++i) {
if (hookPatches[px]&1)
{
MAKE_TPENTA_EDGE1(chan, e0ix,tvertHookStart, e0ix+1, i0+1,
i0);
MAKE_TQUAD(chan, e1ix+qvstep, i1+qvstep, i1, e1ix);
MAKE_TQUAD(chan, e2ix, e2ix-1, i2-1, i2);
MAKE_TQUAD(chan, e3ix-qvstep, i3-qvstep, i3, e3ix);
}
else if (hookPatches[px]&2)
{
MAKE_TQUAD(chan, e0ix, e0ix+1, i0+1, i0);
MAKE_TPENTA_EDGE2(chan, i1, e1ix, tvertHookStart, e1ix+qvstep,
i1+qvstep);
MAKE_TQUAD(chan, e2ix, e2ix-1, i2-1, i2);
MAKE_TQUAD(chan, e3ix-qvstep, i3-qvstep, i3, e3ix);
}
else if (hookPatches[px]&4)
{
MAKE_TQUAD(chan, e0ix, e0ix+1, i0+1, i0);
MAKE_TQUAD(chan, e1ix+qvstep, i1+qvstep, i1, e1ix);
MAKE_TPENTA_EDGE3(chan, i2-1, i2,e2ix, tvertHookStart,e2ix-1);

```

```

    MAKE_TQUAD(chan, e3ix-qvstep, i3-qvstep, i3, e3ix);
}
else if (hookPatches[px]&8)
{
    MAKE_TQUAD(chan, e0ix, e0ix+1, i0+1, i0);
    MAKE_TQUAD(chan, e1ix+qvstep, i1+qvstep, i1, e1ix);
    MAKE_TQUAD(chan, e2ix, e2ix-1, i2-1, i2);
    MAKE_TPENTA_EDGE4(chan, e3ix-qvstep, i3-qvstep, i3,
e3ix,tvertHookStart);
}
else
{
    MAKE_TQUAD(chan, e0ix, e0ix+1, i0+1, i0);
    MAKE_TQUAD(chan, e1ix+qvstep, i1+qvstep, i1, e1ix);
    MAKE_TQUAD(chan, e2ix, e2ix-1, i2-1, i2);
    MAKE_TQUAD(chan, e3ix-qvstep, i3-qvstep, i3, e3ix);
}
e0ix++;
e1ix += qvstep;
e2ix--;
e3ix -= qvstep;
i0++;
i1 += qvstep;
i2--;
i3 -= qvstep;
}
}
}
}
}
assert(tfac == nFaces);
}

```

```

}

mesh.InvalidateGeomCache();
mesh.EnableEdgeList(1);
// The mesh is now valid!
meshValid = TRUE;
return;
}
// Compute the degree-4 alias control points
void Patch::ComputeAux(PatchMesh *pMesh, int index) {
    Point3 p1 = pMesh->getVert(v[index]).p;
    Point3 p2 = pMesh->getVert(v[(index+1)%3]).p;
    int vecIndex = index * 2;
    Point3 v1 = pMesh->getVec(vec[vecIndex]).p;
    Point3 v2 = pMesh->getVec(vec[(vecIndex+1)%6]).p;
    int auxIndex = index * 3;
    aux[auxIndex++] = p1 + (v1 - p1) * 0.75f;
    aux[auxIndex++] = v1 + (v2 - v1) * 0.5f;
    aux[auxIndex++] = v2 + (p2 - v2) * 0.25f;
#ifdef CHECK_TRI_PATCH_AUX
    auxSource[index*3] = p1;
    auxSource[index*3+1] = v1;
    auxSource[index*3+2] = v2;
#endif //CHECK_TRI_PATCH_AUX
}
// Compute interior vertices considering this patch only
void Patch::computeInteriors(PatchMesh *pMesh) {
    PatchVec *vecp = pMesh->vecs;
    switch(type) {
    case 3: {
        // Triangulars must also compute the degree 4 equivalent control points!
        // These get stored in the 'aux' array

```

```

ComputeAux(pMesh, 0);
ComputeAux(pMesh, 1);
ComputeAux(pMesh, 2);

// If the interior points are automatic, compute 'em!
if(flags & PATCH_AUTO) {
    Point3 a = pMesh->getVert(v[0]).p;
    Point3 b = pMesh->getVert(v[1]).p;
    Point3 c = pMesh->getVert(v[2]).p;
    vecp[interior[0]].p = pMesh->getVec(vec[5]).p + (pMesh-
>getVec(vec[0]).p - a);
    vecp[interior[1]].p = pMesh->getVec(vec[1]).p + (pMesh-
>getVec(vec[2]).p - b);
    vecp[interior[2]].p = pMesh->getVec(vec[3]).p + (pMesh-
>getVec(vec[4]).p - c);
    }
}
break;
case 4:
if(flags & PATCH_AUTO) {
    Point3 a = pMesh->getVert(v[0]).p;
    Point3 b = pMesh->getVert(v[1]).p;
    Point3 c = pMesh->getVert(v[2]).p;
    Point3 d = pMesh->getVert(v[3]).p;
    vecp[interior[0]].p = pMesh->getVec(vec[7]).p + (pMesh-
>getVec(vec[0]).p - a);
    vecp[interior[1]].p = pMesh->getVec(vec[1]).p + (pMesh-
>getVec(vec[2]).p - b);
    vecp[interior[2]].p = pMesh->getVec(vec[3]).p + (pMesh-
>getVec(vec[4]).p - c);
    vecp[interior[3]].p = pMesh->getVec(vec[5]).p + (pMesh-
>getVec(vec[6]).p - d);
    }

```

```

    break;
}
}
int PatchMesh::UpdateHooks()
{
//check to make sure all are valid hooks if not delete them
for (int i = 0; i < hooks.Count();i++)
{
    int j = hooks[i].hookEdge;
    int a = edges[j].v1;
    int b = edges[j].v2;

    if ( ((a == hooks[i].upperPoint) && (b == hooks[i].lowerPoint)) ||
        ((b == hooks[i].upperPoint) && (a == hooks[i].lowerPoint)) )
    {
        Point3 delta;
//build a spline based on that edge
        Spline3D work;
        work.AddKnot(SplineKnot(KTYPE_BEZIER, LTYPE_CURVE,
zeroPoint, zeroPoint, zeroPoint));
        work.AddKnot(SplineKnot(KTYPE_BEZIER, LTYPE_CURVE,
zeroPoint, zeroPoint, zeroPoint));
        PatchEdge &edge = edges[j];
        work.SetKnotPoint(0, verts[edge.v1].p);
        work.SetOutVec(0, vecs[edge.vec12].p);
        work.SetInVec(1, vecs[edge.vec21].p);
        work.SetKnotPoint(1, verts[edge.v2].p);
//refine from the spline refine
// Get the knot points
        Point3 v00 = work.GetKnotPoint(0);
        Point3 v30 = work.GetKnotPoint(1);
        Point3 point = work.InterpBezier3D(0, 0.5f);

```

```

Point3 v10 = work.GetOutVec(0);
Point3 v20 = work.GetInVec(1);
Point3 v01 = v00 + (v10 - v00) * 0.5f;
Point3 v21 = v20 + (v30 - v20) * 0.5f;
Point3 v11 = v10 + (v20 - v10) * 0.5f;
Point3 v02 = v01 + (v11 - v01) * 0.5f;
Point3 v12 = v11 + (v21 - v11) * 0.5f;
Point3 v03 = v02 + (v12 - v02) * 0.5f;
Point3 av,bv,cv,dv;
av = v01;
bv = v02;
cv = v12;
dv = v21;
PatchVert patchvert = getVert(hooks[i].hookPoint);
delta = point-patchvert.p;
setVert(hooks[i].hookPoint,point);
setVec(hooks[i].upperVec,av);
setVec(hooks[i].upperHookVec,bv);
setVec(hooks[i].lowerHookVec,cv);
setVec(hooks[i].lowerVec,dv);
for (int ct = 0; ct <patchvert.vectors.Count();ct++)
{
int vecIndex = patchvert.vectors[ct];
if ((vecIndex != hooks[i].upperHookVec) && (vecIndex !=
hooks[i].lowerHookVec))
{
PatchVec vp = getVec(vecIndex);
vp.p += delta;
setVec(vecIndex,vp.p);
}
}
}

```

```
else
{
hooks.Delete(i,1);
i--;
}
}
return 1;
}
```

## Working with Shapes and Splines

See Also: [Class SplineShape](#), [Class BezierShape](#).

## Overview

This section presents information about working with shapes and splines. It covers the main classes used, provides an overview of creating splines, and discusses the capping of shapes with meshes and patches.

A good example shape for study is the Donut plug-in. This shape has two circles, an inner and an outer that make up the shape. The code for this plug-in is in **`\MAXSDK\SAMPLES\OBJECTS\DONUTS.CPP`**.

## Overview of the Principal Classes

The following are the main classes used when working with shapes and splines:

### [Class ShapeObject](#)

These are open or closed hierarchical shape objects. This is the base class that **SimpleSpline**, **SimpleShape**, **SplineShape**, and **LinearShape** are derived from. This class is defined in `\MAXSDK\INCLUDE\OBJECT.H`.

### [Class SplineShape](#)

The **SplineShape** is the shape object flows down the geometry pipeline of MAX. The **SplineShape** contains a **BezierShape**. A **SplineShape** and its contained **BezierShape** are analogous to a **TriObject** which flows down the pipeline and its **Mesh**. This class is defined in `\MAXSDK\INCLUDE\SPLSHAPE.H`.

### [Class BezierShape](#)

The **BezierShape** is effectively a collection of Bezier Splines. For example, the 3ds max Donut object has two splines in a hierarchy to make a shape. The **BezierShape** contains these splines. The **BezierShape** is analogous to the **Mesh** of the **TriObject**. This class is defined in `\MAXSDK\INCLUDE\SHAPE.H`.

### [Class Spline3D](#)

This is a general 3D spline class. The **BezierShape** class has a list of these splines that make up the bezier shape. This class is defined in `\MAXSDK\INCLUDE\SPLINE3D.H`.

### [Class PolyShape](#)

This class is used in the caching of bezier shapes. This is used for doing a one time interpolation of a bezier shape into a form that is the same shape but doesn't require any further interpolation. In this way the system can do the complex calculations once, store the shape into this **PolyShape** representation, and not have to go through the cubic spline calculations to figure out where the points are in the future. This class maintains an array of **PolyLines**. This class is defined in `\MAXSDK\INCLUDE\POLYSH.P.H`.

### [Class PolyLine](#)

This class describes a single polygon in a **PolyShape** using linear segments. This class is defined in **\MAXSDK\INCLUDE\POLYSHP.H**.

#### [Class SimpleSpline](#)

This is a class used in the creation of shape plug-ins. Most of the 3ds max shapes and splines are derived from this class. For example, Line, Arc, Circle, Ellipse and Star are all **SimpleSplines**. This class is defined in **\MAXSDK\INCLUDE\SIMPSPL.H**.

#### [Class SimpleShape](#)

This class is used to make procedural shape primitives easier to create. The 3ds max Helix procedural shape is derived from this class. It's defined in **\MAXSDK\INCLUDE\SIMPSHP.H**.

#### [Class LinearShape](#)

This class is similar to a **SplineShape** except this class uses a **PolyShape** as its data while a **SplineShape** uses a **BezierShape** as its data. Therefore this is a shape made up of entirely linear segments. This class is defined in **\MAXSDK\INCLUDE\LINSHAPE.H**.

## Main Methods in Creating a Spline

Below is a section of code from the Circle plug-in. This plug-in is derived from class **SimpleSpline**. This code demonstrates the key methods used in building a shape. The method **SimpleSpline::BuildShape()** is called to build the shape at the specified time and store the results in **ashape**.

```
void CircleObject::BuildShape(TimeValue t, BezierShape& ashape) {  
    // Start the validity interval at forever and whittle it down.  
    ivalid = FOREVER;  
    float radius;  
    pblock->GetValue(PB_RADIUS, t, radius, ivalid);  
    LimitValue( radius, MIN_RADIUS, MAX_RADIUS );
```

The first thing to notice is the call to **NewShape()**. This ensures the shape is flushed out and emptied.

Next this method calls **MakeCircle()**. See the code for this method below.

```
    ashape.NewShape();  
    // Get parameters from SimpleSpline and place them in the BezierShape  
    int steps;  
    BOOL optimize,adaptive;  
    ipblock->GetValue(IPB_STEPS, t, steps, ivalid);  
    ipblock->GetValue(IPB_OPTIMIZE, t, optimize, ivalid);  
    ipblock->GetValue(IPB_ADAPTIVE, t, adaptive, ivalid);  
    ashape.steps = adaptive ? -1 : steps;  
    ashape.optimize = optimize;
```

Next this method calls **MakeCircle()**. See the code for this method below.

```
    MakeCircle(ashape,radius);
```

After **MakeCircle()** has returned, another important call is made. This is **UpdateSels()**. This should be called when you are done adding all the polygons to the shape. This method updates the selection set information maintained by the shape. It is vital to call this before you are done.

```
    ashape.UpdateSels(); // Make sure it readies the selection set info
```

Finally we clear any caches from the shape.

```

ashape.InvalidateGeomCache();
}
static void MakeCircle(BezierShape& ashape, float radius) {
float vector = CIRCLE_VECTOR_LENGTH * radius;

```

First create a new spline. This is done by calling the **NewSpline()** method on the shape. The shape adds a polygon to itself and returns a pointer to it.

```

Spline3D *spline = ashape.NewSpline();
// Now add all the necessary points
for(int ix=0; ix<4; ++ix) {
float angle = 6.2831853f * (float)ix / 4.0f;
float sinfac = (float)sin(angle), cosfac = (float)cos(angle);
Point3 p(cosfac * radius, sinfac * radius, 0.0f);
Point3 rotvec = Point3(sinfac * vector, -cosfac * vector, 0.0f);

```

Next points or knots are added to the spline by calling **AddKnot()**. This allows you to add different types of knots and line segments.

```

spline-
>AddKnot(SplineKnot(KTYPE_BEZIER,LTYPE_CURVE,
    p,p + rotvec,p - rotvec));
}

```

After adding four knots, the **SetClosed()** method is called to make sure it's a closed circle.

```

spline->SetClosed();

```

Next, an important call is made. The spline has a cached set of bezier points inside it. This needs to be called if you change the points of the spline. This methods updates the information internal to the spline.

```

spline->ComputeBezPoints();
}

```

## Capping a Shape with a Mesh

This section discusses the capping of shapes using a mesh. The SDK provides tools for easily creating these mesh caps for shapes.

Typically for an extruded object, the capping on the front of the object is exactly the same as the capping on the back of the object. For example, if you extrude the letter 'M'. It's redundant to have to cap this separately for the front and back (as they are the same). When dealing with complex shapes, having to go back and do the second cap is very time consuming. So the capping system tries to cache information to speed up the process.

Sample code for capping can be found in the extrude and lathe modifiers. These plug-ins are in `\MAXSDK\SAMPLES\MODIFIERS\SURFREV.CPP` and `\MAXSDK\SAMPLES\MODIFIERS\EXTRUDE.CPP`.

The following code fragments are taken from `EXTRUDE.CPP`. This demonstrates the basic process for capping a shape. This code is part of the method `ExtrudeMod::BuildMeshFromShape()`.

For this example, we are starting of with a `ShapeObject` **shape** that we want capped with a mesh. The first thing to do is convert the shape to a **PolyShape**. This greatly simplifies the mesh conversion. The code below shows how this is done.

```
// Make the shape convert itself to a PolyShape.  
// This makes our mesh conversion MUCH easier!  
PolyShape pShape;  
shape->MakePolyShape(t, pShape);
```

Next, we organize the curves into a hierarchy. This automatically figures out the shape nesting and directions for proper capping.

```
ShapeHierarchy hier = pShape.OrganizeCurves(t);
```

Next we need to reverse the shapes whose directions are incorrect for the hierarchy. The hierarchy calculated above contains a **BitArray** that describes which shapes need to be reversed. This is passed into the `Reverse()` method of the **PolyShape** to tell it which shapes to reverse.

```
// Need to flip the reversed curves in the shape!  
pShape.Reverse(hier.reverse);
```

At this point the **PolyShape** is all set up with the proper clockwise/counter-

clockwise ordering on all the polygons so all that needs to be done is to generate the faces. You may refer to the full source of **EXTRUDE.CPP** to see how this is done.

The next thing is just to create the caps. This begins by instantiating a **MeshCapInfo** class and asking the **PolyShape** to make a cap. This fills up the **MeshCapInfo** class with all the information it needs to create a cap. The type can be morph or grid capping. Morph capping only uses the existing vertices in the **PolyShape** to generate the cap. The capping code does the best job it can given this constraint, however it is possible to wind up with long sliver-like faces on the cap. This is referred to as a morph cap because if you cap a shape using this method it does not generate any new vertices and you can then morph between shapes with the same number of vertices. A Grid cap generates new vertices in the interior of the shape in a grid pattern. This helps to break up the shape and helps reduce slivering. Grid capping will generate different number of vertices based on the shape and thus the shapes are not morphable.

```
MeshCapInfo capInfo;  
pShape.MakeCap(t, capInfo, capType);
```

After this is done, the **MeshCapInfo** is cached within the shape. Therefore if this is needed again, no work needs to be done.

Next, a **MeshCapper** object is created. This is done by passing the **PolyShape** as an argument to the constructor. This gets the **MeshCapper** ready for the topology of the shape. Developers don't need to understand the inner workings of the **MeshCapper**, it's just a tool used to aide in capping.

```
// Build information for capping  
MeshCapper capper(pShape);
```

Below is the code where the start of the extrusion is capped. Inside the capper is a **MeshCapPoly**. There is one for each polygon in the shape. The **MeshCapPoly** needs to know the corresponding mesh vertex for each vertex in the **PolyLine**. This is done by calling **SetVert()** on the **MeshCapPoly**. For example, this might associate vertex 0 in the **PolyLine** with vertex 200 in the mesh, vertex 1 with mesh vertex 220, etc.

```
if(capStart) {  
vert = 0;  
for(poly = 0; poly < polys; ++poly) {
```

```

PolyLine &line = pShape.lines[poly];
MeshCapPoly &capline = capper[poly];
int lverts = line.numPts;
for(int v = 0; v < lverts; ++v)
    // Gives this vert's location in the mesh!
    capline.SetVert(v, vert++);
vert += lverts * levels;
}

```

The next thing that is done is used only for grid capping. A grid cap generates new vertices inside the shape that make up the grid. In the case of a SurfRev for example, the end cap might be rotated, or scaled in some manner. A matrix is required so the capper knows how to orient the vertices into the correct location. This matrix is ignored for non-grid capping.

```

// Create a work matrix for grid capping
Matrix3 gridMat = TransMatrix(offset1);

```

The final step is to cap the mesh. This is done using a method of the capper named **CapMesh()**. This method is passed the output mesh, the **MeshCapInfo**, a flag that indicates if the cap should be flipped, the smoothing group number for all the faces in the cap, and a pointer to the orientation matrix.

```

capper.CapMesh(mesh, capInfo, TRUE, 16, &gridMat);

```

Once this is done the shape has been capped with a mesh.

## Capping a Shape with a Patch

This section discusses the capping of shapes with patches. This is very similar to the mesh capping discussed above.

Again, the following code fragments are taken from **EXTRUDE.CPP**. This demonstrates the basic process for capping a shape with a patch.

Here we try to use a **BezierShape** to create the cap. First a **BezierShape** object is created. Then the shape to be capped is asked if it can make a transformation from itself to a **BezierShape**. If it can, we just ask it to make a **BezierShape**. If it cannot, we are forced to use a **PolyShape**. For patch capping this is much less desirable, but if we can't make a **BezierShape**, then this is used. The shape is asked to convert to a **PolyShape**, and then the **PolyShape** is converted to a **BezierShape**. This is a very poor conversion, as all it does is make a linear spline out of the **PolyShape**.

```
// If the shape can convert itself to a BezierShape, have it do so!  
BezierShape bShape;  
if (shape->CanMakeBezier())  
    shape->MakeBezier(t, bShape);  
else {  
    PolyShape pShape;  
    shape->MakePolyShape(t, pShape);  
    bShape = pShape; // UGH -- Convert it from a PolyShape -- not  
    good!  
}
```

Next, the curves are organized into a hierarchy. This automatically figures out the shape nesting and directions for proper capping.

```
ShapeHierarchy hier;  
bShape.OrganizeCurves(t, &hier);
```

Next we need to reverse the shapes whose direction is incorrect for the hierarchy. The hierarchy calculated above contains a **BitArray** that describes which shapes need to be reversed. This is passed into the reverse method of the **BezierShape** to tell it which shapes to reverse.

```
// Need to flip the reversed polys...  
bShape.Reverse(hier.reverse);
```

At this point the **BezierShape** is all set up with the proper clockwise/counter-clockwise ordering on all the polygons. Next the extrude modifier generates the patches. You may refer to the full source of **EXTRUDE.CPP** to see how this is done.

The next step is to create the caps. This begins by instantiating a **PatchCapInfo** class and asking the **BezierShape** to make a cap. This fills up the **PatchCapInfo** class with all the information it needs to create a cap.

```
PatchCapInfo capInfo;  
bShape.MakeCap(t, capInfo);
```

After this is done, the **PatchCapInfo** is cached within the shape. Therefore if this is needed again, no work needs to be done.

Next, a **PatchCapper** object is created. This is done by passing the **BezierShape** as an argument to the constructor. This gets the **PatchCapper** ready for the topology of the shape. Developers don't need to understand the inner workings of the **PatchCapper**, it's just a tool used to aide in capping.

```
// Build information for capping  
PatchCapper capper(bShape);
```

Below is the code where the start of the extrusion is capped. Inside the capper is a **PatchCapPoly**. There is one for each polygon in the shape. The **PatchCapPoly** needs to know the vertex number in the patch mesh that corresponds to the given knot in the bezier spline. This is done by calling **SetVert()** on the **PatchCapPoly**. After this is done, the same thing happens for each of the vectors when **SetVec()** is called.

```
if(capStart) {  
  vert = 0;  
  int baseVec = 0;  
  for(poly = 0; poly < polys; ++poly) {  
    Spline3D *spline = bShape.splines[poly];  
    PatchCapPoly &capline = capper[poly];  
    int lverts = spline->KnotCount();  
    for(int v = 0; v < lverts; ++v)  
      // Gives this vert's location in the mesh!  
      capline.SetVert(v, vert++);  
    vert += lverts * levels;  
  }  
}
```

```

vec = baseVec;
int lvecs = spline->Segments() * 2;
for(v = 0; v < lvecs; ++v)
  // Gives this vec's location in the mesh!
  capline.SetVec(v, vec++);
  baseVec += lvecs * (levels + 1) + spline->KnotCount() * levels * 2;
}

```

There are vectors generated inside the patch cap. In the case of a SurfRev for example, the end cap might be rotated, or scaled in some manner. A matrix is required so the capper knows how to orient the vectors into the correct location.

```

// Create a work matrix for capping
Matrix3 mat = TransMatrix(offset1);

```

The final step is to cap the patch mesh. This is done using a method of the capper named **CapPatchMesh()**. This method is passed the output patch mesh, the **PatchCapInfo**, a flag that indicates if the cap should be flipped, the smoothing group number for all the patches in the cap, and a pointer to the orientation matrix.

```

capper.CapPatchMesh(pmesh, capInfo, TRUE, 16, &mat);

```

Once this is done the shape has been capped with a patch.

## Modifier Pipeline Note

The ShapeObject class has a special method, **CopyBaseData()**, which is used by derived classes to make sure that, when they copy themselves, they are also copying the data in the **ShapeObject**. This is usually used in the assignment operator of derived classes. It is also used in the **ShallowCopy** method of objects that are passed up the modifier pipeline. In this case, it is VITAL that you wrap the **CopyBaseData** call in Suspend and Resume calls to the undo mechanism:

```

theHold.Suspend();
CopyBaseData(*fob);
theHold.Resume();

```

If this is not done, when certain undo functions are performed, a crash will occur

due to the temporary pipeline object being destroyed. Just remember to wrap the call as shown in any code that is used as part of the modifier pipeline evaluation.

## **Common Problems and Solutions**

This section provides a series of commonly-encountered problems and their solutions.

**Problem:**

Your DLL compiles and links fine, but when you run 3ds max it complains that **LibVersion()** is not implemented. You know that you implemented this, so what is going on?

**Solution:**

Did you remember to create a **.DEF** file with this listed, and if so did you remember to include the file in your project?

**Problem:**

When you link **InitCommonControls** comes up unresolved.

**Solution:**

Did you remember to link in **COMCTL32.LIB** (which is not included by default by Visual C++ in the list of libraries to link)?

**Problem:**

You have a dialog proc to process a button in your user interface. The code is in place to process the button but it never gets called.

**Solution:**

You may have a macro redefinition for one of the buttons in another header file in your project. The compiler takes the first definition and uses it instead of your definition. Therefore yours never gets processed. The compiler will flag this condition as a warning but not as an error.

**Problem:**

When you de-allocate memory the system crashes. You have checked that the item being de-allocated exists and is OK to delete. What is happening?

**Solution:**

In Visual C++ the debug and release C runtime libraries use different heap management. This means that allocating an object in debug mode and de-allocating it in release mode (and vice versa) can cause a crash. You need to change your settings for your DEBUG configuration. Choose **Build/Settings...** from the pulldown menus. From the **Settings For:** section choose your Debug configuration. From the **C/C++** tab, under the **Categories:** drop down list choose **Code Generation**. From the **Use runtime library:** drop down list choose **Multithreaded DLL** instead of **Debug Multithreaded DLL**.

**Problem:**

You have a plug-in which when compiled under the hybrid settings works fine. When you go to make it in release mode, it works fine except for any time it is used and 3ds max is exited you get a "Runtime error. R6017 unexpected multithread lock error".

**Solution:**

The project settings for the Release configuration in the code generation section for run-time library are set to DLL instead of Multithreaded DLL. Changing to Multithreaded DLL corrects the problem.

**Problem:**

You have created a plug-in that runs fine on certain machines but on other machines it simply won't load.

**Solution:**

You probably have a missing system DLL. The easiest way to see what DLLs your plug-in requires is to use the **DUMPBIN.EXE** program (which comes with Visual C++ and is usually found in **\MSDEV\BIN**) and use the **/IMPORTS** switch. This will tell you all of the DLLs that your DLL depends upon. Make sure the machines that need to run the plug-in have the proper DLLs.

For example: **dumpbin /imports utility.dlu**

**Problem:**

You have a plug-in whose interface won't show up in the command panel yet everything seems fine code wise. What's wrong?

**Solution:**

You may have forgotten to enter the class names for the custom controls in the VC++ dialog editor. See the Advanced Topic [Custom User Interface Controls](#) -- especially the section How to a Create a Rollup Page using the Custom Controls for a description of the values to be used in the Class field of the Custom Control Properties dialog.

# Class Interface

See Also: [Class FPStaticInterface](#), [Class ViewExp](#), [Class INode](#), [Class INodeTab](#), [Class CommandMode](#), [Class Interval](#), [Class Renderer](#), [Class Modifier](#), [Class Control](#), [Class Atmospheric](#), [Class Point3](#), [Class Matrix3](#), [Class ModContext](#), [Class ReferenceTarget](#), [Template Class Tab](#)

```
class Interface : public FPStaticInterface
```

## Description:

This class provides an interface for calling functions that are exported from the 3DStudio MAX executable. All the methods in this class are implemented by MAX itself.

Methods are provided for putting up many standard MAX dialogs, working with command modes, working with viewports, controlling the prompt, toolbar and status areas, and working with selection sets. There are also methods for creating objects and nodes in the scene, setting and getting the current time and animation ranges, working with the standard directories of MAX, and many more. See the Method Groups listed below for a breakdown of the various kinds of methods available.

## Important Note for Finding Specific Methods:

To find documentation for a specific method choose the **Help Topics** button and the **Index** tab, then type:

**methods**, followed by the name of the method. Note the space after the comma -- this is required.

For example, enter

**methods, ForceCompleteRedraw**

This will jump directly to the **Interface::ForceCompleteRedraw()** method.

Note: When editing in the command panel, a developer gets passed an interface pointer during **BeginEditParams()**. This pointer is only valid before **EndEditParams()** is finished. A developer should not hang on to this pointer and call methods on it after **EndEditParams()** has returned.

## Method Groups

These hyperlinks take you to the start of groups of related methods within the class.

[Action Table and Menu Manager Methods](#)  
[Add / Delete Class Methods](#)  
[Ambient/Atmosphere/Background Access](#)  
[Animation / Time / Playback Methods](#)  
[Auto Backup Time Related Methods](#)  
[Axis System Related Methods](#)  
[Bitmap/Texmap Related Methods](#)  
[Callback / Notification Registration](#)  
[Creation Related Methods](#)  
[Command Modes](#)  
[Command Panel and Rollup Page Methods](#)  
[Delete Key Notification](#)  
[Deferred Loading Related Methods](#)  
[Dialogs -- Methods to Display MAX Dialogs](#)  
[Dialogs -- Register Windows](#)  
[Directory Access -- Plug-In / Bitmap Paths](#)  
[DLL Directory Access](#)  
[Error Logging](#)  
[Environment Access](#)  
[Execute MAX Commands](#)  
[Execute -- Generic Expansion Function](#)  
[Extended Display Modes](#)  
[Filenames and Pathnames](#)  
[File Open / Merge / Save / Reset / Hold / Fetch](#)  
[Fonts / Cursor Related Methods](#)  
[Grid Related Methods](#)  
[Import / Export Related Methods](#)  
[Keyboard Accelerators](#)  
[Keyboard Shortcut Related Methods](#)  
[Light Related Methods](#)  
[Licensing Methods](#)  
[Material / Texmap Related Methods](#)  
[Modifier Related Methods](#)  
[Node Grouping](#)  
[Node Names -- Creating Unique](#)  
[Node Picking \(Interactive Selection\)](#)  
[Node Related Methods](#)

[Node Selection Sets](#)  
[IObjCreate and IObjParam Pointer Casting](#)  
[Object Snap Methods](#)  
[Plug-In Renderer Access](#)  
[Preview Creation](#)  
[Progress Bar Methods](#)  
[Property Set Access](#)  
[RAMPlayer Access](#)  
[Redraw Viewports](#)  
[Renderer Access \(also see \[Plug-In Renderer Access\]\(#\)\)](#)  
[Render Effects Methods](#)  
[Right Click Menu Related Methods](#)  
[Selection Sets \(Named\)](#)  
[Selection Sets \(Nodes\)Scene Access](#)  
[Show End Result Related Methods](#)  
[Slave / Server Mode Method](#)  
[Snap Related Methods](#)  
[Sound Object Access](#)  
[Sub-Object Related Methods](#)  
[Sub-Object Selection Sets \(Named\)](#)  
[Status Panel / Prompt Related Methods](#)  
[Texmap / Material Related Methods](#)  
[Time Configurations Key Steps Settings Access](#)  
[Track Bar and Track View Related Methods](#)  
[Transform Gizmo Related Methods](#)  
[Undo / Redo Related Methods](#)  
[User Interface Controls and Properties](#)  
[Video Post Related Methods](#)  
[Viewport Access \(Redrawing, etc\)](#)  
[Viewport Background Properties](#)  
[Window Handle of MAX](#)  
[Windows Messages](#)  
[XRef Methods](#)

## **Methods:**

## **Animation / Time / Playback Related Methods**

**Prototype:****virtual TimeValue GetTime()=0;****Remarks:**

Returns the current time, i.e. the frame slider position. See the Advanced Topics section on [Time](#) for an overview of time in MAX.

**Prototype:****virtual void SetTime(TimeValue t, BOOL redraw=TRUE)=0;****Remarks:**

Set the current time, updates the frame slider, and optionally redraws the viewports.

**Parameters:****TimeValue t**

The time to set as current.

**BOOL redraw=TRUE**

If set to FALSE, the current time will be set to the specified time but the viewports will not be redrawn.

**Prototype:****virtual Interval GetAnimRange()=0;****Remarks:**

Returns the current setting of the animation interval. This can be used to get the total number of frames in the animation. Note: The values stored in the interval returned represent ticks not frames.

**Prototype:****virtual void SetAnimRange(Interval range)=0;****Remarks:**

Sets the animation interval. Note: The values passed in the interval are ticks not frames.

**Parameters:****Interval range**

Specifies the new animation range to set.

**Prototype:**

**virtual void StartAnimPlayback(int selOnly=FALSE)=0;**

**Remarks:**

Begins animation playback. The animation may be played for all objects, or just the selected ones.

**Parameters:**

**int selOnly=FALSE**

If TRUE only the selected objects are updated as the animation is played; otherwise all objects are.

**Prototype:**

**virtual void EndAnimPlayback()=0;**

**Remarks:**

Terminates the animation playback.

**Prototype:**

**virtual BOOL IsAnimPlaying()=0;**

**Remarks:**

Returns TRUE if the animation is currently playing; otherwise FALSE.

**Prototype:**

**virtual void EnableAnimateButton(BOOL enable)=0;**

**Remarks:**

Sets the state of the Animate button to enabled or disabled. When disabled the user cannot turn on Animate mode.

Note: Developers have additional functions available for controlling the state of the animate button. These functions are defined in **\MAXSDK\INCLUDE\CONTROL.H**. These functions can be used to determine if animating is on or off, or toggle it on and off without affecting the appearance of the Animate button in the user interface. In this way, a user will not be aware anything is happening. See [List of Additional Controller Related](#)

[Functions](#) for documentation.

**Parameters:**

**BOOL enable**

Pass TRUE to enable the button; FALSE to disable it.

**Prototype:**

**virtual BOOL IsAnimateEnabled()=0;**

**Remarks:**

Returns TRUE if the Animate button is enabled; otherwise FALSE. See the note in the method above as well.

**Prototype:**

**virtual void SetAnimateButtonState(BOOL onOff)=0;**

**Remarks:**

Turns the animate button (and animate mode) on or off.

**Parameters:**

**BOOL onOff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual BOOL GetRealTimePlayback()=0;**

**Remarks:**

Returns the state of the real-time animation playback toggle.

**Prototype:**

**virtual void SetRealTimePlayback(BOOL realTime)=0;**

**Remarks:**

Sets the state of the real-time animation playback toggle.

**Parameters:**

**BOOL realTime**

TRUE if frames should be dropped if necessary for the animation to play back in real time. FALSE specifies that every frame should be played.

**Prototype:****virtual BOOL GetPlayActiveOnly()=0;****Remarks:**

This method returns the flag controlling which viewports are updated when the animation is played.

**Return Value:**

TRUE if all the viewports are updated during play; FALSE if only the active viewport is updated.

**Prototype:****virtual void SetPlayActiveOnly(BOOL playActive)=0;****Remarks:**

This method sets the flag controlling which viewports are updated when the animation is played. This may be all the viewports, or just the active one.

**Parameters:****BOOL playActive**

If TRUE, only the active viewport is updated as the animation is played; otherwise all the viewports are updated.

**Auto Backup Time Related Methods****Prototype:****virtual float GetAutoBackupTime()=0;****Remarks:**

This method is available in release 2.0 and later only.  
Returns the auto backup interval in minutes.

**Prototype:****virtual void SetAutoBackupTime(float minutes)=0;****Remarks:**

This method is available in release 2.0 and later only.  
Sets the auto backup time interval.

**Parameters:**

**float minutes**

The time to set in minutes.

**Prototype:**

**virtual BOOL AutoBackupEnabled()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if auto backup mode is enabled; FALSE if it's disabled.

**Prototype:**

**virtual void EnableAutoBackup(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Enables or Disables the auto backup system.

**Parameters:**

**BOOL onOff**

TRUE to turn it on; FALSE to turn it off.

## **Action Table and Menu Manager Methods**

**Prototype:**

**virtual IActionManager\* GetActionManager()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the Action Manager interface class. The action manager is used to manage ActionTables which plug-ins can use to export operations that can be tied to UI elements like keyboard shortcuts, menus and toolbars. See [Class IActionManager](#).

**Prototype:**

**virtual IMenuManager\* GetMenuManager()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the manager for customizable menus. See [Class IMenuManager](#).

## Axis System Related Methods

### Prototype:

```
virtual Matrix3 GetTransformAxis(INode *node,int  
subIndex,BOOL* local = NULL)=0;
```

### Remarks:

An item that is doing sub-object hit testing gets to specify what their sub-object axes systems are. For example a mesh may have separate coordinate systems for every face or group of selected faces, while, for instance, a bend modifier has its own axes system for the gizmo. This method gets the axes system for a particular node. Each node may have several axes systems identified by an index.

See **EDITMESH.CPP** for an example of use.

### Parameters:

**INode \*node**

The node to get the axis coordinates system of.

**int subIndex**

The index of the axis system of the node.

**BOOL\* local = NULL**

If 'local' is not NULL, it will be set to TRUE if the center of the axis is the pivot point of the node, FALSE otherwise.

### Return Value:

The axis system of the node.

### Prototype:

```
virtual int GetNumAxis()=0;
```

### Remarks:

This returns the number of axis tripods in the scene. When transforming multiple sub-objects, in some cases each sub-object is transformed in a different space.

**Return Value:**

One of the following values:

**NUMAXIS\_ZERO**

Nothing to transform.

**NUMAXIS\_ALL**

Use only one axis.

**NUMAXIS\_INDIVIDUAL**

Do all, one at a time.

**Prototype:**

```
virtual Matrix3 GetTransformAxis(INode *node, int subIndex,  
BOOL* local = NULL)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the axis which defines the space in which transforms should take place.

**Parameters:**

**INode \*node**

The object the axis system should be based on.

**int subIndex**

The sub object which the axis system should be based on (the thing the user clicked on).

**BOOL\* local = NULL**

If 'local' is not NULL, it will be set to TRUE if the center of the axis is the pivot point of the node, FALSE otherwise.

**Return Value:**

A matrix representing the axis system that transforms take place in.

**Prototype:**

```
virtual void LockAxisTripods(BOOL onOff)=0;
```

**Remarks:**

This method locks axis tripods so that they will not be updated.

**Parameters:**

**BOOL onOff**

TRUE to lock; FALSE to unlock.

**Prototype:**

**virtual BOOL AxisTripodLocked()=0;**

**Remarks:**

This method returns TRUE if axis tripods are locked.

## **Bitmap / Texmap Related Methods**

**Prototype:**

**virtual void FreeSceneBitmaps()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method traverses the scene reference hierarchy, calling

**Animatable::FreeAllBitmaps()** on every Animatable. This will free up all the memory used by bitmaps.

**Prototype:**

**virtual void EnumAuxFiles(NameEnumCallback& nameEnum,  
DWORD flags)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method may be used to enumerate all the bitmap files in the scene. The **flags** allow control over which files are enumerated.

**Parameters:**

**NameEnumCallback& nameEnum**

The callback, called once for each bitmap. See [Class NameEnumCallback](#).

**DWORD flags**

See [List of EnumAuxFiles\(\) Flags](#).

**Prototype:**

**virtual void RenderTexmap(Texmap \*tex, Bitmap \*bm, float scale3d=1.0f, BOOL filter=FALSE, BOOL display=FALSE)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method may be called to render a texmap (or an entire texmap tree) to the specified bitmap.

**Parameters:**

**Texmap \*tex**

The Texmap to render to a bitmap.

**Bitmap \*bm**

A pointer to a bitmap to render to. This bitmap must be created at the resolution you wish to render to.

**float scale3d=1.0f**

This is a scale factor applied to 3D Texmaps. This is the scale of the surface in 3d space that is mapped to UV. This controls how much of the texture appears in the bitmap representation.

**BOOL filter=FALSE**

If TRUE the bitmap is filtered. It is quite a bit slower to rescale bitmaps with filtering on.

**BOOL display=FALSE**

If TRUE the resulting bitmap is displayed using the virtual frame buffer; otherwise it is not.

**Prototype:**

**virtual bool CanImportBitmap(const TCHAR\* filename)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method will check if the system can import the specified bitmap file.

**Parameters:**

**const TCHAR\* filename**

The file name to check.

**Return Value:**

TRUE if the specified file is a bitmap file of a format that is supported by one of the bitmap reader plug-ins; otherwise FALSE.

**Prototype:**

**virtual bool CaptureSubObjectRegistration(bool OnOff, Class\_ID cid)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used to lock and unlock subobject mode registrations and is primarily used by the FileLink wrapper classes acting as proxies to other classes in 3D Studio VIZ. When a class calls this method with OnOff set to TRUE, then other classes are prevented from registering new subobject modes. This continues until the original class "releases" by calling CaptureSubObjectModes(FALSE, myClassID). The second argument insures that only the class which does the capture can do the release. Note that this is used only by certain VIZ plugins.

**Parameters:**

**bool OnOff**

TRUE to prevent other classes from registering new subobject modes.

**Class\_ID cid**

The class ID.

**Return Value:**

TRUE is successful, otherwise FALSE.

**Prototype:**

**virtual bool DownloadUrl(HWND hwnd, const TCHAR\* url, const TCHAR\* filename, DWORD flags = 0)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method simplifies downloading files from any given URL and displays a floating progress dialog.

**Parameters:**

**HWND hwnd**

The window handle for owner window (required for the progress floating dialog).

**const TCHAR\* url**

The string for the resource/file to download.

**const TCHAR\* filename**

The target location and filename for the downloaded file.

**DWORD flags = 0**

Additional controls to the download behavior. Currently only one flag is supported, **DOWNLOADDLG\_NOPLACE**, which hides an option in the progress dialog that allows the user to place (move) a dropped object immediately after being dropped.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

```
virtual INode* GetImportCtxNode(void)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

On drag-and-drop, if the drop type is a file, the drop handler searches for an importer plugin that can handle the file (based on its extension). Some drop operations, such as bitmaps and material XML files, can or must be dropped on to an object in the scene.

**Return Value:**

The node that the operation is performed on, if one is "hit" at the drop location. It returns NULL if no nodes were found at the drop location.

**Callback / Notification Registration**

Related to callbacks is [Structure NotifyInfo](#) and its associated functions.

Please see this topic for additional ways a developer can register a callback for events like the user performing a File/Reset or File/New, the system unit settings changing, etc.

**Prototype:**

**virtual void RegisterTimeChangeCallback(TimeChangeCallback \*tc)=0;**

**Remarks:**

Registers a callback object that will get called every time the user changes the MAX frame slider.

**Parameters:**

**TimeChangeCallback \*tc**

Points to the callback object to register. See: [Class TimeChangeCallback](#).

**Prototype:**

**virtual void**

**UnRegisterTimeChangeCallback(TimeChangeCallback \*tc)=0;**

**Remarks:**

This method un-registers the time change callback.

**Parameters:**

**TimeChangeCallback \*tc**

Points to the callback object to un-register. See: [Class TimeChangeCallback](#).

**Prototype:**

**virtual void**

**RegisterCommandModeChangedCallback(CommandModeChange \*cb)=0;**

**Remarks:**

Register a callback object that will get called when the user changes the command mode.

**Parameters:**

**CommandModeChangedCallback \*cb**

Points to the callback object to register. See [Class CommandModeChangedCallback](#).

**Prototype:**

**virtual void**

**UnRegisterCommandModeChangedCallback(CommandModeChar \*cb)=0;**

**Remarks:**

Un-registers the command mode change callback object.

**Parameters:**

**CommandModeChangedCallback \*cb**

Points to the callback object to un-register. See [Class CommandModeChangedCallback](#).

**Prototype:**

**virtual void RegisterViewportDisplayCallback(BOOL preScene, ViewportDisplayCallback \*cb)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Registers a **ViewportDisplayCallback** whose **Display()** method will be called to allow a plug-in to draw in the MAX viewports.

**Parameters:**

**BOOL preScene**

If TRUE the callback will be called before objects are rendered (typically, but not always); if FALSE the callback is called after the objects are rendered.

In some cases redrawing the viewports may take two passes; once to re-render the background plane and once to rerender the foreground plane. In this case the order of events would be:

Call pre callbacks

Render scene

Call post callbacks

Call pre callbacks

Render scene

Call post callbacks

The two calls to callbacks in the middle are neither pre nor post callbacks. However you could also look at this as two separate redraws.

**ViewportDisplayCallback \*cb**

Points to the callback object. See [Class ViewportDisplayCallback](#).

**Prototype:**

```
virtual void UnRegisterViewportDisplayCallback(BOOL  
preScene, ViewportDisplayCallback *cb)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Call this method to un-register the viewport display callback object.

**Parameters:**

**BOOL preScene**

If TRUE the callback will be called before object are rendered (typically, but not always); if FALSE the callback is called after the objects are rendered.

**ViewportDisplayCallback \*cb**

Points to the callback object. See [Class ViewportDisplayCallback](#).

**Prototype:**

```
virtual void NotifyViewportDisplayCallbackChanged(BOOL  
preScene, ViewportDisplayCallback *cb)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to inform MAX that the viewport callback has changed. This is similar to when an object that is part of the reference hierarchy changes and it needs to call **NotifyDependents()** with the message **REFMSG\_CHANGE** to inform the items that depend on it. In a sense a **ViewportDisplayCallback** is like an object in the scene in that it has a **Display()** method that is called. However, the callback is not actually an object that is part of the reference hierarchy so it cannot send a **REFMSG\_CHANGE** message when it changes. So this method provides the equivalent functionality. If the callback changes this method needs to be called.

To understand why this is needed consider that if a plug-in called **RedrawViews()** five times in a row, the viewports may be redrawn the first time (if something has changed), but the next four calls won't do anything.

This is because MAX maintains some flags that indicate if things have changed or not and these flags are reset on the first redraw to indicate that everything is up to date. Therefore, when a **ViewportDisplayCallback** changes, it needs to call this method to let MAX know that changes have been made and the viewports indeed need to be redrawn the next time **RedrawViews()** is called.

**Parameters:**

**BOOL preScene**

If TRUE the callback will be called before objects are rendered (typically, but not always); if FALSE the callback is called after the objects are rendered.

**ViewportDisplayCallback \*cb**

Points to the callback object. See [Class ViewportDisplayCallback](#).

**Prototype:**

```
virtual void RegisterExitMAXCallback(ExitMAXCallback  
*cb)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Registers a **ExitMAXCallback** whose **Exit()** method will be called when MAX is about to exit. The return value from the callback allows the plug-in to decide if MAX exits or not.

**Parameters:**

**ExitMAXCallback \*cb**

Points to the callback object. See [Class ExitMAXCallback](#).

**Prototype:**

```
virtual void UnRegisterExitMAXCallback(ExitMAXCallback  
*cb)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Un-registers the exit callback so it's no longer called.

**Parameters:**

## **ExitMAXCallback \*cb**

Points to the callback object. See [Class ExitMAXCallback](#).

### **Prototype:**

```
virtual void RegisterAxisChangeCallback(AxisChangeCallback  
*cb)=0;
```

### **Remarks:**

Registers a callback object that will get called any time the user changes the reference coordinate system by:

Changing the transform coordinate system drop-down menu.

Changing the state of the transform center fly-off.

Changing X, Y, Z, XY, YZ, ZX constraint buttons/fly-off.

Using an accelerator or anything else that changes the above.

### **Parameters:**

**AxisChangeCallback \*cb**

Points to the callback to register. See [AxisChangeCallback](#).

### **Prototype:**

```
virtual void UnRegisterAxisChangeCallback(AxisChangeCallback  
*cb)=0;
```

### **Remarks:**

Un-registers the axis change callback.

### **Parameters:**

**AxisChangeCallback \*cb**

Points to the callback to un-register. See [AxisChangeCallback](#).

### **Prototype:**

```
virtual void  
RegisterRedrawViewsCallback(RedrawViewsCallback *cb)=0;
```

### **Remarks:**

Registers a call back object that gets called every time the viewports are redrawn. The **proc()** method is called after the views are finished redrawing.

**Parameters:**

**RedrawViewsCallback \*cb**

Points to the callback object whose **proc()** method is called when the viewports are redrawn. See [Class RedrawViewsCallback](#).

**Prototype:**

**virtual void**

**UnRegisterRedrawViewsCallback(RedrawViewsCallback \*cb)=0;**

**Remarks:**

Un-registers the viewport redraw callback.

**Parameters:**

**RedrawViewsCallback \*cb**

Points to the callback object to un-register. See [Class RedrawViewsCallback](#).

**Prototype:**

**virtual void RegisterSelectFilterCallback(SelectFilterCallback \*cb)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Registers a call back object that gets called to filter the selection of nodes.

**Parameters:**

**SelectFilterCallback \*cb**

Points to the callback object to register. See [Class SelectFilterCallback](#).

**Prototype:**

**virtual void UnRegisterSelectFilterCallback(SelectFilterCallback \*cb)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Un-registers the select filter callback.

**Parameters:**

### **SelectFilterCallback \*cb**

Points to the callback object to un-register. See [Class SelectFilterCallback](#).

#### **Prototype:**

```
virtual void RegisterDisplayFilterCallback(DisplayFilterCallback *cb)=0;
```

#### **Remarks:**

This method is available in release 4.0 and later only.

Registers a call back object that gets called to filter the display of nodes.

#### **Parameters:**

### **DisplayFilterCallback \*cb**

Points to the callback object to register. See [Class DisplayFilterCallback](#).

#### **Prototype:**

```
virtual void UnRegisterDisplayFilterCallback(DisplayFilterCallback *cb)=0;
```

#### **Remarks:**

This method is available in release 4.0 and later only.

Un-registers the display filter callback.

#### **Parameters:**

### **DisplayFilterCallback \*cb**

Points to the callback object to un-register. See [Class DisplayFilterCallback](#).

#### **Prototype:**

```
virtual PickModeCallback* GetCurPickMode()=0;
```

#### **Remarks:**

This method is available in release 4.0 and later only.

If a Pick Mode is on top of the command stack, then this function will return the **PickModeCallback**. If the Pick Mode is not on top of the command stack the it returns NULL.

## Creation Related Methods

### Prototype:

```
virtual INode *CreateObjectNode( Object *obj )=0;
```

### Remarks:

Creates a new node in the scene with the given object. Normally a developer will use the standard creation methods for procedural objects and this method is not used. However if the developer wants to handle the creation process on their own they may need to use this method. See Also: [Object Creation Methods](#).

### Parameters:

**Object \*obj**

Pointer to the Object to create.

### Return Value:

Pointer to the node created.

### Prototype:

```
virtual void *CreateInstance(SClass_ID superID, Class_ID classID)=0;
```

### Remarks:

Creates an instance of a registered class. This will call **Create()** on the class descriptor.

### Parameters:

**SClass\_ID superID**

The super class ID of the item to create an instance of.

**Class\_ID classID**

The class ID of the item to create an instance of. See [Class Class\\_ID](#).

### Return Value:

Pointer to the created instance.

### Also Note:

There is a global method that duplicates the functionality of this class method to let you create an instance of any registered class wherever you are (without the interface pointer):

**void \*CreateInstance(SClass\_ID superID, Class\_ID classID);**

**Prototype:**

**virtual void NonMouseCreate(Matrix3 tm)=0;**

**Remarks:**

This creates a new object/node without going through the usual create mouse proc sequence. The matrix is relative to the construction plane. This must be called during the creation phase of an object. For example, the procedural sphere uses it when the user clicks on the 'Create' button after they type in the parameters for the sphere. See [Object Creation Methods](#).

**Parameters:**

**Matrix3 tm**

The transformation matrix relative to the construction plane.

**Prototype:**

**virtual void NonMouseCreateFinish(Matrix3 tm)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is presently only used in the Line object. It's used for repositioning the pivot point of the object at the end of the creation process. You simply pass it the new matrix, and the creation manager sets the TM for the node being created to the given TM \* the construction plane TM.

**Parameters:**

**Matrix3 tm**

The transformation matrix relative to the construction plane for the node.

**Prototype:**

**virtual void StopCreating()=0;**

**Remarks:**

When a plug-in object implements its own **ClassDesc::BeginCreate()** / **ClassDesc::EndCreate()** it can cause **EndCreate()** to be called by calling this method. See [Object Creation Methods](#).

**Prototype:**

```
virtual int BindToTarget(INode *laNode, INode *targNode)=0;
```

**Remarks:**

This method binds a node to a target using a lookat controller.

**Parameters:**

**INode \*laNode**

Pointer to the node to assign the lookat controller to.

**INode \*targNode**

Pointer to the target node.

**Return Value:**

Returns TRUE if the node was bound; otherwise FALSE.

**Sample Code:**

This method is used in

**\MAXSDK\SAMPLES\OBJECTS\CAMERA.CPP.**

## Command Modes

For more information on CommandModes, see the Advanced Topics section [Command Modes and Mouse Procs.](#)

**Prototype:**

```
virtual void PushCommandMode(CommandMode *m)=0;
```

**Remarks:**

This method pushes the specified command mode on the stack. Typically this is used by developers handling their own creation using

**ClassDesc::BeginCreate()**. See [Class ClassDesc](#) for more details.

Note: This method works as documented but a developer may not want to use it. The problem is that other modes can be pushed on the stack (such as viewport transformation modes) and it becomes complicated to track when it is OK to pop your mode. See the methods below for alternatives (such as **SetCommandMode**).

**Parameters:**

**CommandMode \*m**

A pointer to the command mode to push.

**Prototype:**

**virtual void SetCommandMode(CommandMode \*m)=0;**

**Remarks:**

This method sets the top of the stack to the specified command mode. A developer should call **DeleteMode()** to delete their command mode when done using it.

**Parameters:**

**CommandMode \*m**

The command mode to set.

**Prototype:**

**virtual void PopCommandMode()=0;**

**Remarks:**

Pops the command mode off the top of the stack.

**Prototype:**

**virtual CommandMode\* GetCommandMode()=0;**

**Remarks:**

Returns the current mode on the top of the stack.

**Prototype:**

**virtual int GetCommandStackSize();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of command modes in the command mode stack.

**Prototype:**

**virtual CommandMode\* GetCommandStackEntry(int entry);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the command mode at the specified position in the command mode stack. A developer may use this to determine if their

command mode is in the stack.

**Parameters:**

**int entry**

The index into the command mode stack of the entry to get. Pass 0 to get the current command mode.

**Prototype:**

**virtual void SetStdCommandMode(int cid)=0;**

**Remarks:**

This is the typical method called by the developer to handle mouse interaction. It allows the developer to set the command mode to one of the standard command modes. For example: **CID\_OBJMOVE, CID\_OBJROTATE, CID\_OBJSCALE, CID\_OBJUSCALE, CID\_OBJsquash, CID\_OBJSELECT**, etc.

**Parameters:**

**int cid**

The index of the command mode to set. See [List of Standard Command Modes](#) to review the full list.

**Prototype:**

**virtual void PushStdCommandMode(int cid)=0;**

**Remarks:**

Allows the developer to push one of the standard command modes on the command stack. For example: **CID\_OBJMOVE, CID\_OBJROTATE, CID\_OBJSCALE, CID\_OBJUSCALE, CID\_OBJsquash, CID\_OBJSELECT**, etc.

**Parameters:**

**int cid**

The index of the command mode to set. See [List of Standard Command Modes](#) to review the full list.

**Prototype:**

**virtual void RemoveMode(CommandMode \*m)=0;**

**Remarks:**

Removes the specified command mode from the stack. This method pops items off the command mode stack up to and including the specified mode. The top item in the stack is then set as the active command mode. As usual, **ExitMode()** is called on the specified mode before it is popped and **EnterMode()** is called on the newly active mode.

**Parameters:**

**CommandMode \*m**

Points to the command mode to remove.

**Prototype:**

**virtual void DeleteMode(CommandMode \*m)=0;**

**Remarks:**

If the developer sets or pushes a command mode, this method should be called when the developer is done with the mode to ensure that it is no longer anywhere in the stack. If the mode is already deleted this method does nothing.

Note: It is normal for a developer to set the 'Select and Move' command mode to be the active one if their mode was at the top of the stack and is being deleted. For instance:

```
ip->SetStdCommandMode(CID_OBJMOVE);
```

```
ip->DeleteMode(&myCMode);
```

**Parameters:**

**CommandMode \*m**

The command mode to delete.

## **Command Panel and Rollup Page methods**

**Prototype:**

```
virtual HWND AddRollupPage(HINSTANCE hInst, TCHAR  
*dlgTemplate, DLGPROC dlgProc, TCHAR *title, LPARAM  
param=0, DWORD flags=0, int category =
```

**ROLLUP\_CAT\_STANDARD)=0;**

**Remarks:**

This method is used to add a rollup page to the command panel. It returns the window handle of the rollup page.

**Parameters:**

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**TCHAR \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the messages sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar of the rollup page.

**LPARAM param=0**

Any specific data to pass along may be stored here. This may be later retrieved using the **GetWindowLong()** call from the Windows API if it was set in the window using **SetWindowLong()**.

For example, at the beginning of the dialog proc do something like:

```
BOOL CALLBACK MyDlgProc(  
    HWND hWnd, UINT message, WPARAM wParam, LPARAM  
    lParam) {
```

```
    MyUtil *u = (MyUtil *)GetWindowLong(hWnd,  
    GWL_USERDATA);  
    if (!u && message != WM_INITDIALOG ) return FALSE;  
    ...
```

Then inside the code block handling the **WM\_INITDIALOG** message do something like:

```
switch (message) {  
case WM_INITDIALOG:  
    u = (MyUtil *)lParam;  
    SetWindowLong(hWnd, GWL_USERDATA, (LONG)u);  
    ...
```

Then later in the dialog you can access the object passed in -- in the case above an instance of **MyUtil** (using **u->**).

### **DWORD flags=0**

The following flag value may be used:

#### **APPENDROLL\_CLOSED**

Starts the page in the rolled up (closed) state.

### **int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

### **Sample Code:**

```
hBendParams = ip->AddRollupPage(  
    hInstance,  
    MAKEINTRESOURCE(IDD_BEND_ROLLUP),  
    BendParamDialogProc,  
    _T("Parameters"),  
    (LPARAM)this );
```

### **Return Value:**

The window handle of the rollup page.

### **Prototype:**

```
virtual HWND AddRollupPage(HINSTANCE hInst,  
    DLGTEMPLATE *dlgTemplate, DLGPROC dlgProc, TCHAR
```

**\*title, LPARAM param=0, DWORD flags=0, int category =  
ROLLUP\_CAT\_STANDARD)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used to add a rollup page to the command panel. It returns the window handle of the rollup page. This method is currently not being used.

**Parameters:**

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**DLGTEMPLATE \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the messages sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar of the rollup page.

**LPARAM param=0**

Any specific data to pass along may be stored here. This may be later retrieved using the **GetWindowLong()** call from the Windows API if it was set in the window using **SetWindowLong()**.

**DWORD flags=0**

The following flag value may be used:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up (closed) state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**. When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the

**RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The window handle of the rollup page.

**Prototype:**

**virtual void DeleteRollupPage(HWND hRollup)=0;**

**Remarks:**

Removes a rollup page and destroys it.

**Parameters:**

**HWND hRollup**

The window handle of the rollup window. This is the handle returned from **AddRollupPage()**.

**Prototype:**

**virtual HWND ReplaceRollupPage(HWND hOldRollup, HINSTANCE hInst, TCHAR \*dlgTemplate, DLGPROC dlgProc, TCHAR \*title, LPARAM param=0, DWORD flags=0, int category = ROLLUP\_CAT\_STANDARD)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This replaces an existing rollup with another one and (deletes the original).

**Parameters:**

**HWND hOldRollup**

The window handle of the old rollup.

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**TCHAR \*dlgTemplate**

The dialog template for the rollup page.

## **DLGPROC dlgProc**

The dialog proc to handle the messages sent to the rollup page.

## **TCHAR \*title**

The title displayed in the title bar of the rollup page.

## **LPARAM param=0**

Any specific data to pass along may be stored here.

## **DWORD flags=0**

The following flag value may be used:

### **APPENDROLL\_CLOSED**

Starts the page in the rolled up (closed) state.

## **int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

## **Return Value:**

The window handle of the rollup.

## **Prototype:**

```
virtual HWND ReplaceRollupPage(HWND hOldRollup,  
HINSTANCE hInst, TCHAR *dlgTemplate, DLGPROC dlgProc,  
TCHAR *title, LPARAM param=0,DWORD flags=0, int category  
= ROLLUP_CAT_STANDARD)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This replaces an existing rollup with another one and (deletes the original).

This method is currently not being used.

**Parameters:****HWND hOldRollup**

The window handle of the old rollup.

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**DLGTEMPLATE \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the messages sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar of the rollup page.

**LPARAM param=0**

Any specific data to pass along may be stored here.

**DWORD flags=0**

The following flag value may be used:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up (closed) state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the

**RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags

argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The window handle of the rollup.

**Prototype:**

**virtual IRollupWindow \*GetCommandPanelRollup()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns a rollup window interface to the command panel rollup. This interface provides methods for showing and hiding rollups, adding and removing rollup pages, etc. Note: This interface does not need to be released with **ReleaseIRollup()** as MAX takes care of this when it shuts down.

See [Class IRollupWindow](#).

**Prototype:**

**virtual void RollupMouseMessage(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam )=0;**

**Remarks:**

This method allows hand cursor scrolling in the command panel when the user clicks and drags the mouse in an unused area of the dialog. When the user mouse-es down in dead area of the command panel, the plug-in should pass mouse messages to this function which will pass them on to the rollup.

Note: This method is obsolete in MAX 2.0 and later. These messages no longer need to be passed along as this is handled internally.

**Parameters:**

**HWND hDlg**

The window handle of the dialog.

**UINT message**

The message sent to the dialog proc.

**WPARAM wParam**

Passed in to the dialog proc. Pass along to this method.

**LPARAM lParam**

Passed in to the dialog proc. Pass along to this method.

**Sample Code:**

```
case WM_LBUTTONDOWN: case WM_LBUTTONUP: case  
WM_MOUSEMOVE:  
ip->RollupMouseMessage(hDlg,message,wParam,lParam);
```

**Prototype:**

```
virtual int GetCommandPanelTaskMode()=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns a value to indicate which branch of the command panel is currently active.

**Return Value:**

One of the following values:

```
TASK_MODE_CREATE  
TASK_MODE_MODIFY  
TASK_MODE_HIERARCHY  
TASK_MODE_MOTION  
TASK_MODE_DISPLAY  
TASK_MODE_UTILITY
```

**Prototype:**

```
virtual void SetCommandPanelTaskMode(int mode)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the branch of the command panel that is currently active.

**Parameters:**

**int mode**

One of the following values:

```
TASK_MODE_CREATE  
TASK_MODE_MODIFY
```

**TASK\_MODE\_HIERARCHY**  
**TASK\_MODE\_MOTION**  
**TASK\_MODE\_DISPLAY**  
**TASK\_MODE\_UTILITY**

## **Delete Key Notification**

### **Prototype:**

**virtual void RegisterDeleteUser(EventUser \*user)=0;**

### **Remarks:**

Registers a callback invoked when the user presses the Delete Key. This allows the developer to override the default processing of the Delete Key.

### **Parameters:**

**EventUser \*user**

Pointer to an instance of the [EventUser](#) class.

### **Sample Code:**

See **\\MAXSDK\\SAMPLES\\MODIFIERS\\EDITSPL.CPP**.

### **Prototype:**

**virtual void UnRegisterDeleteUser(EventUser \*user)=0;**

### **Remarks:**

Un-registers a Delete Key callback.

### **Parameters:**

**EventUser \*user**

Pointer to an instance of the [EventUser](#) class.

## **Dialogs -- Methods to Display MAX dialogs**

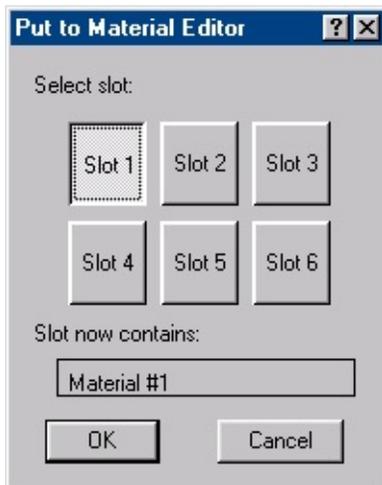
This section describes methods to put up many of the standard dialog boxes used by MAX. Some dialogs are accessed outside of this class. For the Arc Rotate dialog box see [Class ArcballDialog](#). For the modal or modeless color selector dialog see [Class ColorPicker](#). For a Win32 MessageBox() like function that supports 'Hold' and 'Don't show this dialog again' items see [List of Miscellaneous Utility Functions](#).

**Prototype:**

```
virtual void PutMtlToMtlEditor(MtlBase *mb , int slot=-1)=0;
```

**Remarks:**

Puts the specified material into the material editor. The material is put to the specified slot, or if -1 is passed, the following dialog is presented which allows the user to choose a sample slot for the material.

**Parameters:**

**MtlBase \*mb**

The material to put to the material editor.

**int slot=-1**

This parameter is available in release 3.0 and later only.

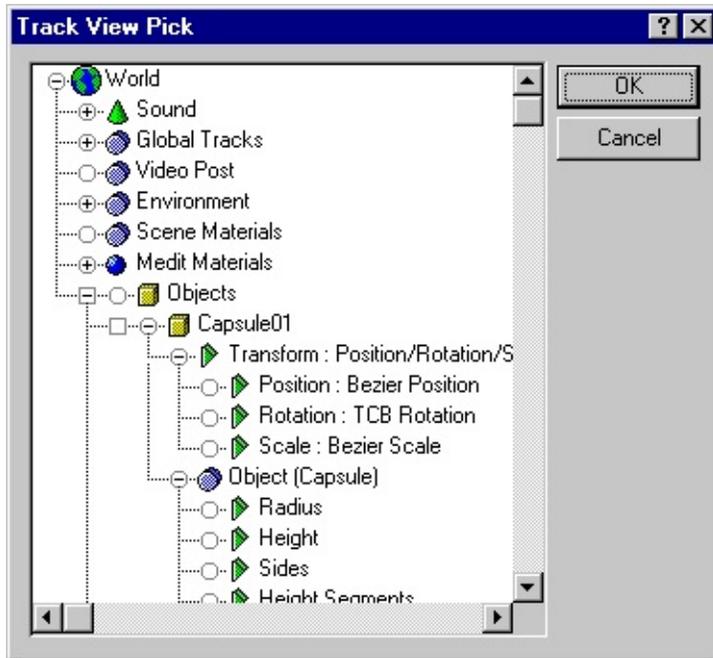
The Materials Editor slot number (a value in the range 0 to 23). If a slot number is specified, then this method will replace that material in the Materials Editor without user interaction. If -1 is passed (the default) then the function brings up the put dialog.

**Prototype:**

```
virtual BOOL TrackViewPickDlg(HWND hParent,  
TrackViewPick *res, TrackViewFilter *filter=NULL, DWORD  
flags=0)=0;
```

**Remarks:**

This method brings up the track view pick dialog. This dialog appears below:



### Parameters:

#### **HWND hParent**

The handle of the parent window.

#### **TrackViewPick \*res**

The item chosen by the user. See [Class TrackViewPick](#)

#### **TrackViewFilter \*filter=NULL**

The call back object to filter selection in the track view. See [Class TrackViewFilter](#).

#### **DWORD flags=0**

Currently not used.

### Return Value:

TRUE if the user selected OK to exit the dialog; otherwise FALSE.

### Sample Code:

This code brings up the Track View Pick Dialog and filters the input to MAX's controllers. After the controller is selected GetValue() is called on it.

```

class MyTVFilter : public TrackViewFilter {
    BOOL proc(Animatable *anim, Animatable *client,int subNum) {
        Control *c = (Control*)anim->GetInterface(I_CONTROL);
        return (c) ? TRUE : FALSE;
    }
};

```

```

void DoTest() {
    TrackViewPick res;
    MyTVFilter tvf;
    BOOL okay = IP->TrackViewPickDlg(IP->GetMAXHwnd(), &res,
    &tvf);
    if (!okay) return;
    Control *c = (Control *)res.anim;
    SClass_ID sid = c->SuperClassID();
    GetSetMethod method = CTRL_ABSOLUTE;
    switch(sid) {
    case CTRL_FLOAT_CLASS_ID:
        float r;
        Interval ivalid;
        c->GetValue(IP->GetTime(), &r, ivalid, method);
        ...
    }
}

```

**Prototype:**

```

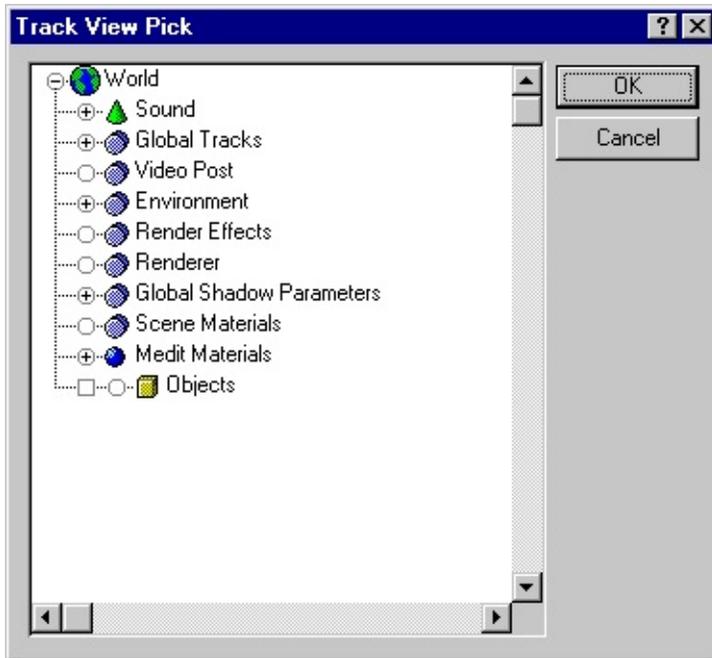
virtual BOOL TrackViewPickMultiDlg(HWND hParent,
    Tab<TrackViewPick> *res, TrackViewFilter
    *filter=NULL,DWORD flags=0)=0;

```

**Remarks:**

This method is available in release 3.0 and later only.

This methods brings up a dialog that allows one to select multiple tracks. This method works much like the **TrackViewPickDlg** method above except it is passed a pointer to a table of a **TrackViewPick** items instead.



### Parameters:

#### **HWND hParent**

The handle of the parent window.

#### **Tab<TrackViewPick> \*res**

This is a table (See [Template Class Tab](#)) of items chosen by the user. See [Class TrackViewPick](#)

#### **TrackViewFilter \*filter=NULL**

The call back object to filter selection in the track view. See [Class TrackViewFilter](#).

#### **DWORD flags=0**

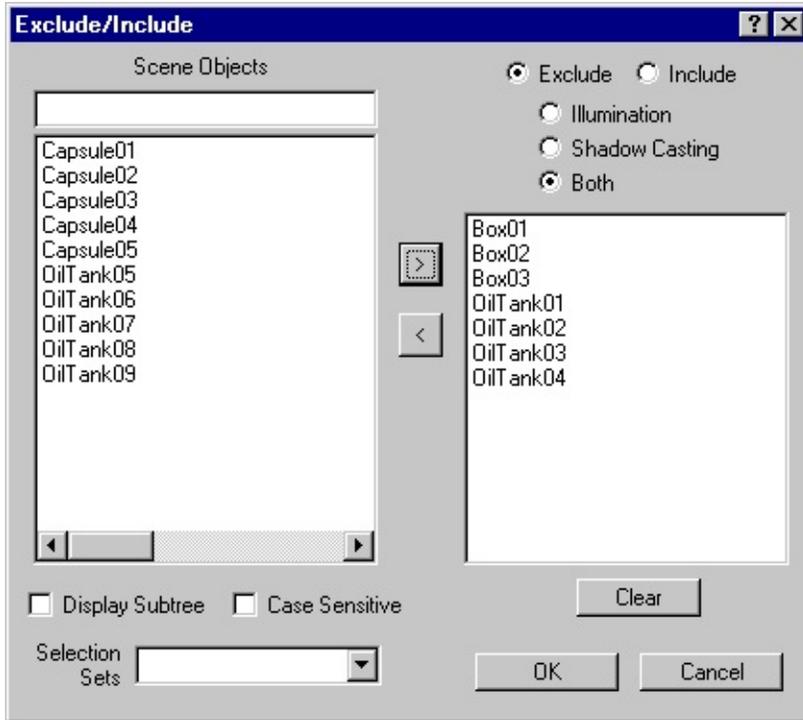
Currently not used.

### Prototype:

```
virtual int DoExclusionListDialog(ExclList *nl, BOOL  
doShadows=TRUE)=0;
```

### Remarks:

This brings up the standard Exclude / Include dialog box used for light exclusion / inclusion lists. This dialog appears below:



### Parameters:

#### **ExclList \*nl**

If the user selects OK, this is the list of names chosen by the user. See [Class ExclList](#).

#### **BOOL doShadows=TRUE**

The shadows switch.

### Return Value:

Nonzero if the user selected OK to exit the dialog; otherwise 0.

### Prototype:

```
virtual void ConvertNameTabToExclList(const NameTab *nt,  
ExclList *excList)=0;
```

### Remarks:

This method is available in release 4.0 and later only.

This method will convert a NameTab to an ExclList (the new format for Exclusion lists).

### Parameters:

**const NameTab \*nt**

A pointer to the name table.

**ExclList \*excList**

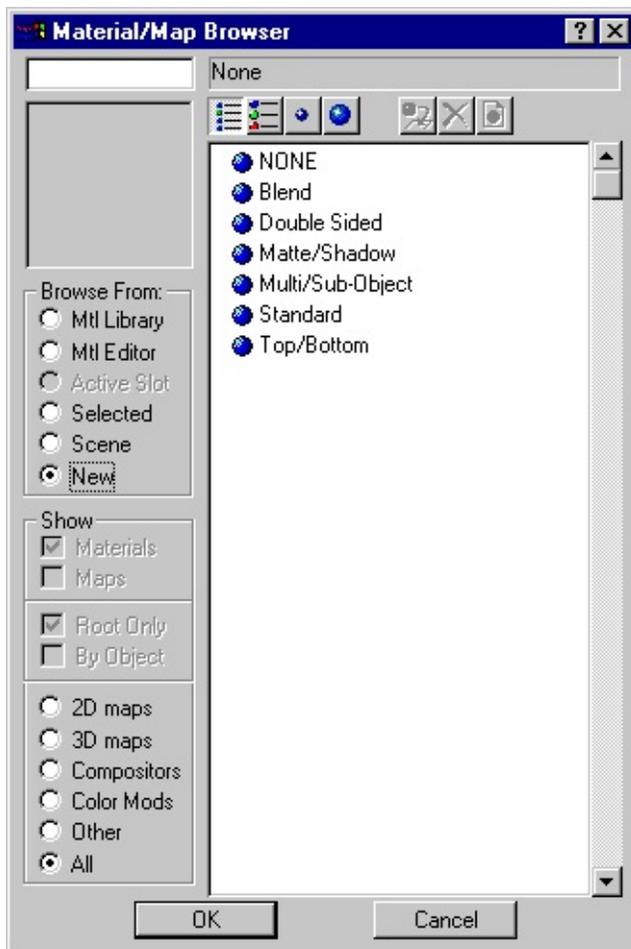
A pointer to the resulting exclusion list.

**Prototype:**

**virtual MtlBase \*DoMaterialBrowseDlg(HWND hParent,DWORD flags,BOOL &newMat,BOOL &cancel)=0;**

**Remarks:**

This method brings up the Material / Map Browser dialog box. This dialog appears below:



**Parameters:**

**HWND hParent**

The parent window handle.

**DWORD flags**

See [List of Material Browser Flags](#).

**BOOL &newMat**

TRUE if the user has selected a new item; otherwise FALSE. If TRUE it is safe to modify this item. If FALSE the item may be an instance and a developer should not modify this as other materials may be using this same item.

**BOOL &cancel**

TRUE if the user canceled the dialog; otherwise FALSE.

**Return Value:**

If **cancel** is FALSE, the item chosen by the user is returned. See [Class MtlBase](#).

**Prototype:**

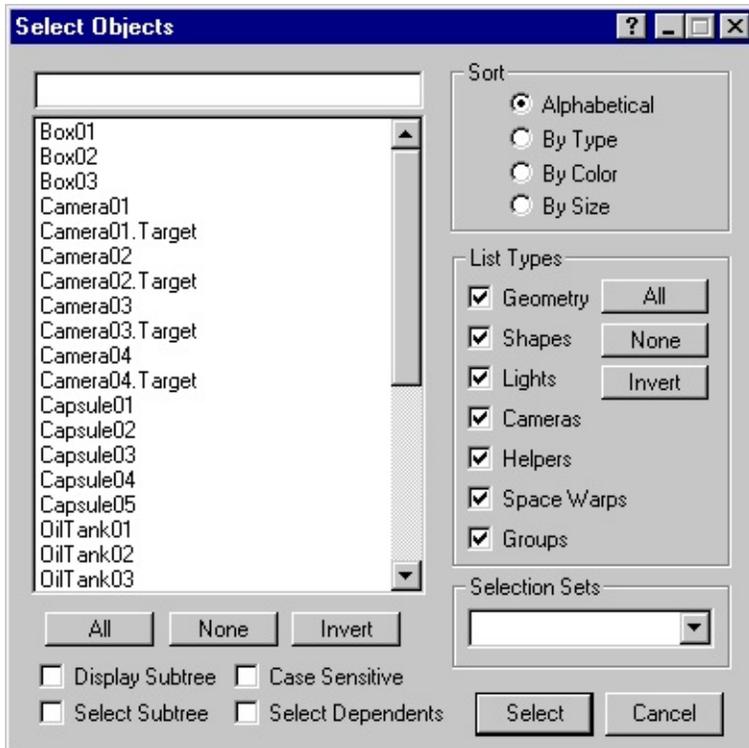
**virtual BOOL DoHitByNameDialog(HitByNameDlgCallback \*hbncb=NULL)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to put up the standard MAX Hit By Name dialog. This dialog appears below:

If the callback is NULL this method does a standard select by name. The nodes chosen by the user are selected if the user selects 'Select' from the dialog to exit. Use **Interface::GetSelNodeCount()** and **GetSelNode(i)** to retrieve the results.



**Parameters:**

**HitByNameDlgCallback \*hbnCb=NULL**

Points to the callback object. See [Class HitByNameDlgCallback](#). Developers should delete this callback when done.

**Return Value:**

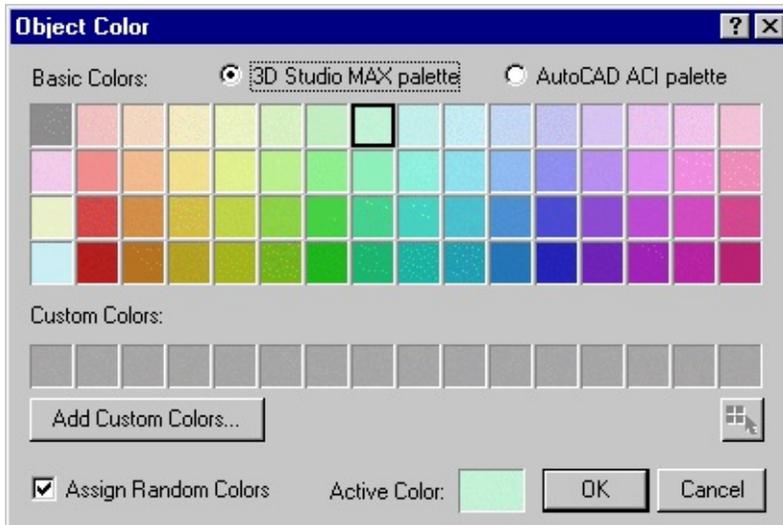
TRUE if the user selects 'Select' from the dialog; otherwise FALSE.

**Prototype:**

**virtual BOOL NodeColorPicker(HWND hWnd,DWORD &col)=0;**

**Remarks:**

This method brings up the standard MAX object color picker dialog. This dialog appears below:



### Parameters:

#### **HWND hWnd**

The parent window handle.

#### **DWORD &col**

If the user picks a color then this will be set to the chosen color. This is stored in a 32 bit format, with the high order 8 bits as 0's, the next 8 bits as the Blue amount, the next 8 bits as the Green amount, and the low order 8 bits as the Red amount (**0x00BBGGRR**). See [COLORREF - DWORD Color Format](#).

### Return Value:

TRUE if the user picks a color and FALSE if the user cancels the dialog.

### Prototype:

```
virtual void ChooseDirectory(HWND hWnd, TCHAR *title,  
TCHAR *dir, TCHAR *desc=NULL)=0;
```

### Remarks:

This method is available in release 2.0 and later only.

This methods puts up the Choose Directory dialog box to allow the user to select a directory. The choosen directory including the drive and path is stored in **dir**. This dialog appears below:



**Parameters:**

**HWND hWnd**

The parent window handle.

**TCHAR \*title**

The title in the dialog box.

**TCHAR \*dir**

The chosen directory is stored here. This points to an empty string on cancel.

**TCHAR \*desc=NULL**

The string to go into the Label field of the dialog. This string may be changed by the user and is returned here.

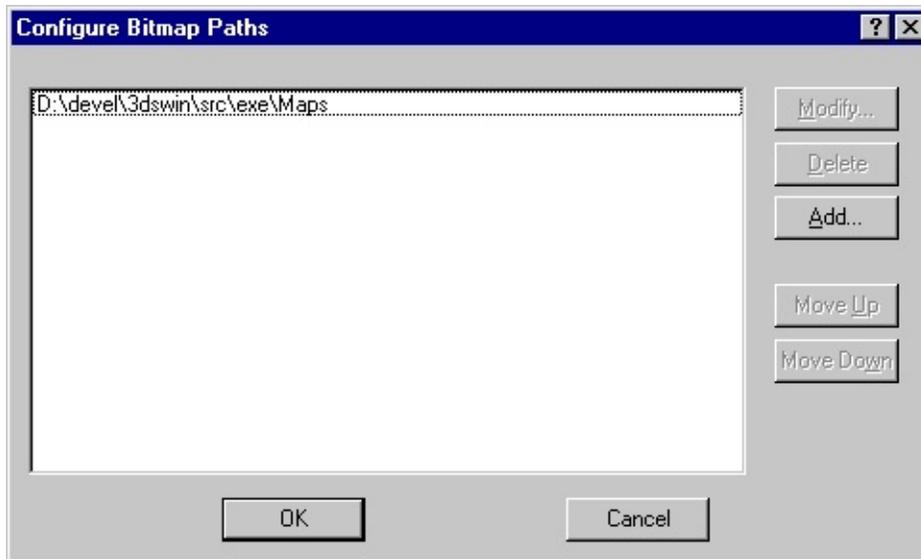
**Prototype:**

**virtual int ConfigureBitmapPaths()**=0;****

**Remarks:**

This method is available in release 3.0 and later only.

This method puts up the dialog to let the user configure the bitmap loading paths. This dialog appears below:



**Return Value:**

Nonzero on user selecting OK, zero on Cancel.

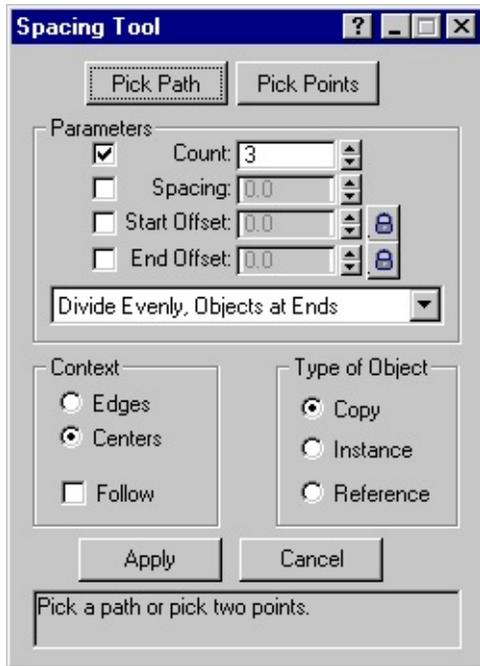
**Prototype:**

```
virtual BOOL DoSpaceArrayDialog(SpaceArrayCallback  
*sacb=NULL)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Puts up the space array dialog. If the callback is NULL it just does the standard space array tool.



### Parameters:

**SpaceArrayCallback \*sacb=NULL**

The callback. See [Class SpaceArrayCallback](#).

### Return Value:

Returns TRUE if the user OKs the dialog, otherwise FALSE.

## RAM Player

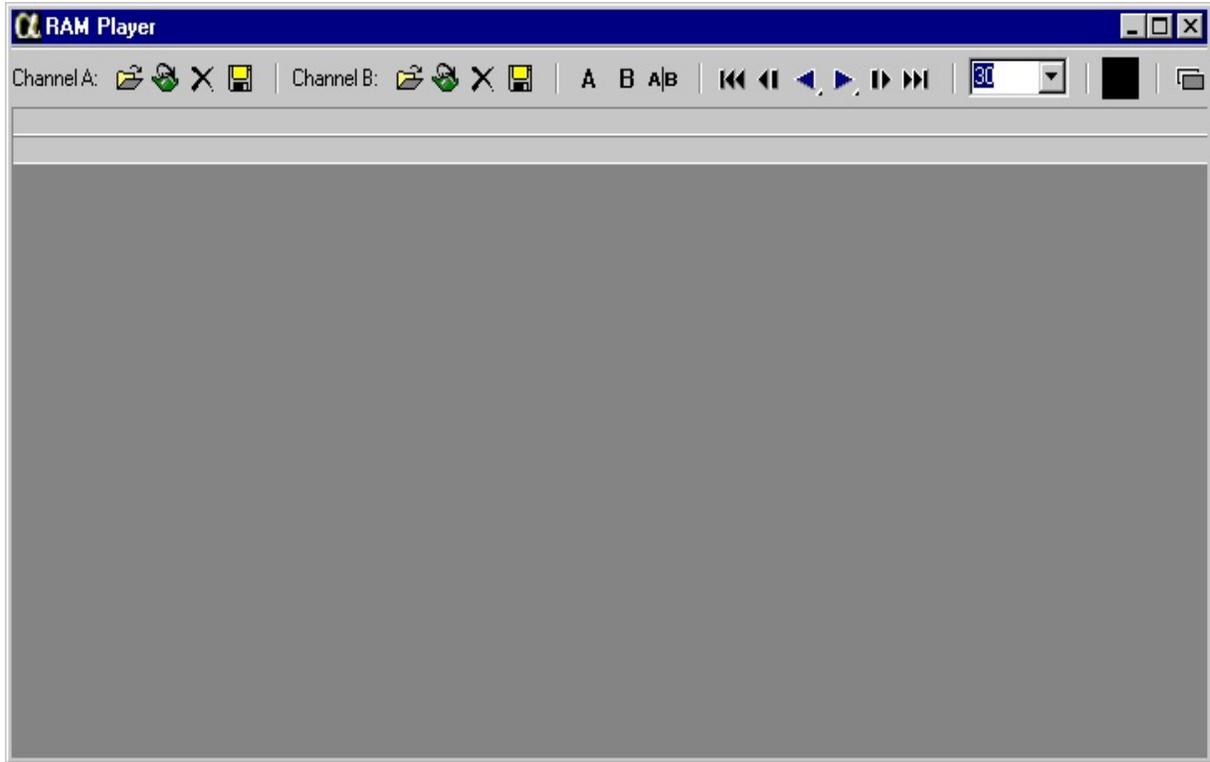
### Prototype:

```
virtual void RAMPlayer(HWND hWndParent, TCHAR*  
szChanA=NULL, TCHAR* szChanB=NULL)=0;
```

### Remarks:

This method is available in release 3.0 and later only.

Brings up the RAMPlayer dialog and optionally loads one, or both channels with the supplied files.



### Parameters:

#### **HWND hWndParent**

The parent window handle.

#### **TCHAR\* szChanA=NULL**

The file to load for channel A (for example, `_T("movie.avi")`). If NULL is passed no file is loaded into the channel.

#### **TCHAR\* szChanB=NULL**

The file to load for channel B. If NULL is passed no file is loaded into the channel.

## Dialogs -- Register Windows

### Prototype:

```
virtual void RegisterDlgWnd(HWND hDlg)=0;
```

### Remarks:

Registers a dialog window so **IsDialogMessage()** gets called for it. This is not required if you add rollup pages to the command panel as this is done automatically, however if you create a floating, modeless dialog you **must** call this method.

**Important Note:** ALL modeless dialogs in MAX must be registered to the application window with this method so that it, and any sub dialogs, will behave as they should.

**Parameters:**

**HWND hDlg**

The window handle of the dialog.

**Prototype:**

**virtual int UnRegisterDlgWnd(HWND hDlg)=0;**

**Remarks:**

Un-registers a dialog window so **IsDialogMessage()** is no longer called for it.

**Parameters:**

**HWND hDlg**

The window handle of the dialog.

**Return Value:**

Nonzero if successful; otherwise 0.

## **Directory Access -- Plug-In and Bitmap Paths**

**Prototype:**

**virtual TCHAR \*GetDir(int which)=0;**

**Remarks:**

Returns the pathname of a directory used by 3ds max.

**Parameters:**

**int which**

One of the following directories. See [List of Directory Names](#).

**Return Value:**

The pathname of the directory.

**Prototype:**

**virtual int GetPlugInEntryCount()=0;**

**Remarks:**

This is the number of entries in **PLUGIN.INI**. **PLUGIN.INI** contains a list of descriptions and directories used by plug-in DLLs. See Also: [Plug-In Directory Search Mechanism](#).

**Return Value:**

The number of entries in **PLUGIN.INI**.

**Prototype:**

```
virtual TCHAR *GetPlugInDesc(int i)=0;
```

**Remarks:**

Returns the 'i-th' description string from **PLUGIN.INI**. See Also: [Plug-In Directory Search Mechanism](#).

**Parameters:**

**int i**

Specifies which description to return.

**Return Value:**

The 'i-th' description string from **PLUGIN.INI**.

**Prototype:**

```
virtual TCHAR *GetPlugInDir(int i)=0;
```

**Remarks:**

Returns the pathname string for the 'i-th' plug-in directory from **PLUGIN.INI**. See Also: [Plug-In Directory Search Mechanism](#).

**Parameters:**

**int i**

Specifies which directory to return.

**Return Value:**

The 'i-th' pathname string from **PLUGIN.INI**.

## DLL Directory Access

**Prototype:**

**virtual DllDir& GetDllDir()=0;**

**Remarks:**

Returns a reference to the central DLL directory. See [Class DllDir](#).

**Prototype:**

**virtual DllDir \*GetDllDirectory()=0;**

**Remarks:**

Returns a pointer to the central DLL directory. See [Class DllDir](#).

## **Error Logging**

The following methods allow a developer to write to the error log file created in each network server's directory. These three methods allow more detailed information to be recorded than the single line error message that is returned to the network "master" when a render fails. If there is any place in your code where an Alert is put up that might be encountered during rendering, you can print the information out to the error log instead.

**Prototype:**

**virtual LogSys \*Log()=0;**

**Remarks:**

Returns a pointer which may be used for calling methods to write information to the system log. See [Class LogSys](#) for details.

## **Environment Access**

**Prototype:**

**virtual Texmap \*GetEnvironmentMap()=0;**

**Remarks:**

Returns the current environment map.

**Prototype:**

**virtual void SetEnvironmentMap(Texmap \*map)=0;**

**Remarks:**

Sets the current environment map to the specified map. See **\MAXSDK\SAMPLES\UTILITIES\UTILTEST.CPP** for sample code.

**Parameters:**

**Texmap \*map**

The map to set.

**Prototype:**

**virtual BOOL GetUseEnvironmentMap()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the 'Use Map' checkbox is checked in the Environment / Background dialog; otherwise FALSE.

**Prototype:**

**virtual void SetUseEnvironmentMap(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the state of the 'Use Map' checkbox in the Environment / Background dialog.

**Parameters:**

**BOOL onOff**

TRUE for checked; FALSE for unchecked.

**Prototype:**

**virtual Point3 GetAmbient(TimeValue t,Interval &valid)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Retrieves the color of the ambient light at the time passed and updates the validity interval passed to reflect the validity of the ambient light.

**Parameters:**

**TimeValue t**

The time to retrieve the ambient light color.

## **Interval &valid**

The validity interval to update.

### **Return Value:**

The color as a **Point3**.

### **Prototype:**

```
virtual void SetAmbient(TimeValue t, Point3 col)=0;
```

### **Remarks:**

This method is available in release 2.0 and later only.

Sets the color of the ambient light in the scene to the color passed at the specified time.

### **Parameters:**

#### **TimeValue t**

The time to set the color.

#### **Point3 col**

The new color for the ambient light.

### **Prototype:**

```
virtual Control *GetAmbientController()=0;
```

### **Remarks:**

This method is available in release 2.0 and later only.

Retrieves a pointer to the controller use to animate the ambient light.

### **Prototype:**

```
virtual void SetAmbientController(Control *c)=0;
```

### **Remarks:**

This method is available in release 2.0 and later only.

Sets the controller used for handling the animation of the ambient light.

### **Parameters:**

#### **Control \*c**

The controller to set.

**Prototype:**

**virtual Point3 GetLightTint(TimeValue t,Interval &valid)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the rendering environment global lighting tint color at the specified time and updates the validity interval passed to reflect the validity of the tint color controller.

**Parameters:****TimeValue t**

The time at which to return the color.

**Interval &valid**

The validity interval that is updated.

**Prototype:**

**virtual void SetLightTint(TimeValue t, Point3 col)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the rendering environment global lighting tint color at the specified time to the color passed.

**Parameters:****TimeValue t**

The time at which to set the color.

**Point3 col**

The color to set.

**Prototype:**

**virtual Control \*GetLightTintController()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the controller use to animate the tint color.

**Prototype:**

**virtual void SetLightTintController(Control \*c)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the controller use to animate the tint color.

**Parameters:**

**Control \*c**

Points to the controller to set.

**Prototype:**

**virtual float GetLightLevel(TimeValue t,Interval &valid)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the rendering environment global lighting level at the specified time and updates the validity interval passed to reflect the validity of the lighting level controller.

**Parameters:**

**TimeValue t**

The time at which to return the level.

**Interval &valid**

The validity interval that is updated.

**Prototype:**

**virtual void SetLightLevel(TimeValue t, float lev)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the rendering environment global lighting level at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the lighting level.

**float lev**

The level to set.

**Prototype:**

**virtual Control \*GetLightLevelController()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the controller use to animate the lighting level.

**Prototype:**

**virtual void SetLightLevelController(Control \*c)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the controller use to animate the lighting level.

**Parameters:**

**Control \*c**

Points to the controller to set.

**Prototype:**

**virtual int NumAtmospheric()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the number of atmospheric effects currently assigned.

**Prototype:**

**virtual Atmospheric \*GetAtmospheric(int i)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the 'i-th' atmospheric effect.

**Parameters:**

**int i**

Specifies which atmospheric effect to retrieve.

**Prototype:**

**virtual void SetAtmospheric(int i,Atmospheric \*a)=0;**

**Remarks:**

This method is available in release 2.0 and later only.  
Sets the 'i-th' atmospheric effect.

**Parameters:**

**int i**

Specifies which effect to set.

**Atmospheric \*a**

A pointer to the atmospheric effect.

**Prototype:**

**virtual void AddAtmosphere(Atmospheric \*atmos)=0;**

**Remarks:**

This method is available in release 2.0 and later only.  
Adds the specified atmospheric effect to the list of effects.

**Parameters:**

**Atmospheric \*a**

A pointer to the atmospheric effect to add.

**Prototype:**

**virtual void DeleteAtmosphere(int i)=0;**

**Remarks:**

This method is available in release 2.0 and later only.  
Deletes the specified atmospheric effect.

**Parameters:**

**int i**

The index of the atmospheric effect to delete.

**Prototype:**

**virtual void EditAtmosphere(Atmospheric \*a, INode \*gizmo=NULL)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method selects the specified atmosphere's gizmo and displays the parameters for it (if any).

**Parameters:**

**Atmospheric \*a**

Points to the Atmospheric plug-in. See [Class Atmospheric](#).

**INode \*gizmo=NULL**

Points to the gizmo node associated with the plug-in.

**Render Effects Methods****Prototype:**

**virtual int NumEffects()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of Render Effects currently assigned.

**Prototype:**

**virtual Effect \*GetEffect(int i)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the 'i-th' Render Effect. See [Class Effect](#).

**Parameters:**

**int i**

The zero based index of the effect to return.

**Prototype:**

**virtual void SetEffect(int i, Effect \*e)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified Render Effect to the one passed.

**Parameters:**

**int i**

The zero based index of the effect to set.

**Effect \*e**

Points to the Renderer Effect to set.

**Prototype:**

**virtual void AddEffect(Effect \*eff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Adds the specified Renderer Effect to the existing list of effects.

**Parameters:**

**Effect \*eff**

Points to the render effect to add. See [Class Effect](#).

**Prototype:**

**virtual void DeleteEffect(int i)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes the specified Renderer Effect.

**Parameters:**

**int i**

The zero based index of the effect to delete.

**Prototype:**

**virtual void EditEffect(Effect \*e, INode \*gizmo=NULL)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method selects the specified gizmo and displays the parameters for it (if any).

**Parameters:**

**Effect \*e**

Points to the Effect plug-in. See [Class Effect](#).

**INode \*gizmo=NULL**

Points to the gizmo node associated with the effect.

**Prototype:**

**virtual Point3 GetBackGround(TimeValue t,Interval &valid)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Retrieves the background color at the specified time and updates the validity interval passed to reflect the validity of the background color.

**Parameters:**

**TimeValue t**

The time to retrieve the color.

**Interval &valid**

The validity interval to update.

**Prototype:**

**virtual void SetBackGround(TimeValue t,Point3 col)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the background color to the specified color at the specified time.

**Parameters:**

**TimeValue t**

The time to set the color.

**Point3 col**

The color to set.

**Prototype:**

**virtual Control \*GetBackGroundController()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the controller animating the background color.

**Prototype:**

**virtual void SetBackGroundController(Control \*c)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the controller used for animating the background color.

**Parameters:**

**Control \*c**

Specifies which controller to set.

## Execute MAX Commands

**Prototype:**

**virtual void ExecuteMAXCommand(int id)=0;**

**Remarks:**

This method may be used to execute a MAX command. These are the same commands that may be assigned using the MAX

**Customize/Preferences.../Keyboard Tab** key assignment system.

For MAX version 1.1 or later, this method may also be used to set various aspects of the preview display. The **id** to pass for these options is shown below:

**ExecuteMAXCommand(MAXCOM\_API\_PVW\_GRID\_OFF)**

This turns off the preview grid display.

**ExecuteMAXCommand(MAXCOM\_API\_PVW\_GRID\_ON)**

This turns on the preview grid display.

**ExecuteMAXCommand(MAXCOM\_API\_PVW\_SMOOTH\_MODE)**

This sets the preview rendering mode to "smooth".

**ExecuteMAXCommand(MAXCOM\_API\_PVW\_FACET\_MODE)**

This sets the preview rendering mode to "facet".

**ExecuteMAXCommand(MAXCOM\_API\_PVW\_WIRE\_MODE)**

This sets the preview rendering mode to "wireframe".

Note: In MAX 2.0 and later there is an alternate (better) way to do this. See the method:

**virtual void CreatePreview(PreviewParams \*pvp=NULL)=0;**

**Parameters:**

**int id**

The command to execute. See [List of MAX Command IDs](#).

## **Execute -- General API Expansion Function**

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG\_PTR arg1=0, ULONG\_PTR arg2=0, ULONG\_PTR arg3=0, ULONG\_PTR arg4=0, ULONG\_PTR arg5=0, ULONG\_PTR arg6=0)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The MAX development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

Note: In R4 the return value changed from int to INT\_PTR

**New in R5.1** An additional Command was added to allow the setting of certain directories such as Plugins. It looks like this:

**Execute(I\_EXEC\_SET\_DIR, (ULONG\_PTR)(int) which, ULONG\_PTR (TCHAR \*)dir)**

where 'which' designates the particular Max directory to be changed (like the corresponding argument of Interface::GetDir()), and 'dir' is the path as a string.

**Parameters:**

**int cmd**

The index of the command to execute. See [List of Interface::Execute Command Options](#).

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**ULONG arg4=0**

This parameter is available in release 4.0 and later only.

Optional argument 4.

**ULONG arg5=0**

This parameter is available in release 4.0 and later only.

Optional argument 5.

**ULONG arg6=0**

This parameter is available in release 4.0 and later only.

Optional argument 6.

**Return Value:**

An INT\_PTR return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

## Extended Display Modes

**Prototype:**

**virtual void SetExtendedDisplayMode(int flags)=0;**

**Remarks:**

This method is used internally to set the extended display mode.

**Parameters:**

**int flags**

See [List of Extended Display Modes](#).

**Prototype:**

**virtual int GetExtendedDisplayMode()=0;**

**Remarks:**

Returns the extended display mode flags. This method provides a mechanism to retrieve some additional information about an object that is more dependent on MAX than on the particular object. For example, when a spotlight is selected, it can use this method to detect that it is the only item selected, and display its cone. It checks this using and extended display mode

**EXT\_DISP\_ONLY\_SELECTED.** See [List of Extended Display Modes](#).

## **Filenames and Pathnames**

### **Prototype:**

**virtual TSTR &GetCurFileName()=0;**

### **Remarks:**

Returns the name of the current MAX file (but not the path). For example, if the currently loaded file is "D:\3DSMAX\SCENES\Expgears.max" this method returns " **Expgears.max**".

### **Prototype:**

**virtual TSTR &GetCurFilePath()=0;**

### **Remarks:**

Returns the file and path of the current MAX file. For example, if the currently loaded file is "D:\3DSMAX\SCENES\Expgears.max" this method returns "D:\3DSMAX\SCENES\Expgears.max ".

### **Prototype:**

**virtual TCHAR \*GetMatLibFileName()=0;**

### **Remarks:**

Returns the current material library file name.

## **File Open/Merge/Save/Reset/Hold/Fetch Methods**

For file IO using the MAX import / export mechanism see the section: [Import / Export Related Methods](#).

### **Prototype:**

**virtual bool IsMaxFile(const TCHAR\* filename)=0;**

### **Remarks:**

This method is available in release 4.0 and later only.

Returns true if the specified file is a valid MAX file; otherwise false.

### **Parameters:**

**const TCHAR\* filename**

The name of the file to check.

**Prototype:**

**virtual bool IsInternetCachedFile(const TCHAR\* filename)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns true if the specified file is an internet cached file; otherwise false.

**Parameters:**

**const TCHAR\* filename**

The name of the file to check.

**Prototype:**

**virtual void SetMAXFileOpenDlg(MAXFileOpenDialog\* dlg)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method allows a custom file open dialog to be registered.

**Parameters:**

**MAXFileOpenDialog\* dlg**

Points to the file open dialog object to use. See [Class MAXFileOpenDialog](#).

**Prototype:**

**virtual void SetMAXFileSaveDlg(MAXFileSaveDialog\* dlg)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method allows a custom file save dialog to be registered.

**Parameters:**

**MAXFileSaveDialog\* dlg**

Points to the file save dialog object to use. See [Class MAXFileSaveDialog](#).

**Prototype:**

**virtual void FileOpenMatLib(HWND hWnd)=0;**

**Remarks:**

This method brings up the File Open dialog box and allows the user to select a material library to load.

**Parameters:**

**HWND hWnd**

The parent window handle.

**Prototype:**

**virtual void FileSaveMatLib(HWND hWnd)=0;**

**Remarks:**

If the current material library has been saved previously (has been named) this method saves the material library to the same file. Otherwise it brings up the standard Save File As dialog box to allow the user to save the current material library.

**Parameters:**

**HWND hWnd**

The parent window handle.

**Prototype:**

**virtual void FileSaveAsMatLib(HWND hWnd)=0;**

**Remarks:**

Brings up the standard Save File As dialog box to allow the user to save the current material library.

**Parameters:**

**HWND hWnd**

The parent window handle.

**Prototype:**

**virtual void LoadDefaultMatLib()=0;**

**Remarks:**

This method loads the default material library **3DSMAX.MAT** (if this file

exists).

**Prototype:**

**virtual int LoadFromFile(const TCHAR \*name, BOOL refresh=TRUE)=0;**

**Remarks:**

Loads the specified MAX file. A developer should normally specify a complete path name. This method does not bring up a file dialog.

**Parameters:**

**const TCHAR \*name**

The MAX file to load.

**BOOL refresh=TRUE**

This parameter is available in release 2.0 and later only.

Set this to FALSE to prevent the viewports from automatically being refreshed.

**Return Value:**

Nonzero if the file was loaded; otherwise 0.

**Prototype:**

**virtual int SaveToFile(const TCHAR \*fname)=0;**

**Remarks:**

Saves the current scene to the specified MAX file. This method does not bring up a file dialog.

**Parameters:**

**const TCHAR \*name**

The MAX file to save.

**Return Value:**

Nonzero if the library was saved; otherwise 0.

**Prototype:**

**virtual void FileSaveSelected(TCHAR \*fname)=0;**

**Remarks:**

This method is available in release 2.0 and later only.  
Saves the selected nodes to the specified file.

**Parameters:**

**TCHAR \*fname**

The MAX file to save.

**Prototype:**

**virtual void FileSaveNodes(INodeTab\* nodes, TCHAR \*fname)=0;**

**Remarks:**

This method is available in release 2.0 and later only.  
Saves the specified nodes to the specified file.

**Parameters:**

**INodeTab\* nodes**

Points to the table of nodes to save. See [Class INodeTab](#).

**TCHAR \*fname**

The MAX file to save.

**Prototype:**

**virtual int LoadMaterialLib(const TCHAR \*name, MtlBaseLib \*lib=NULL)=0;**

**Remarks:**

Loads the specified material library. This method does not bring up a file dialog.

**Parameters:**

**const TCHAR \*name**

The material library to load.

**MtlBaseLib \*lib=NULL**

This parameter is available in release 2.0 and later only.

Points to the material library to load into. If NULL the library is loaded into the current material library. See [Class MtlBaseLib](#).

Note: You need to call **MtlBaseLib::DeleteAll()** on the library during a MAX reset operation. This will remove all its references to the materials and

set its count to zero.

**Return Value:**

Nonzero if the library was loaded; otherwise 0.

**Prototype:**

**virtual int SaveMaterialLib(const TCHAR \*name, MtlBaseLib \*lib=NULL)=0;**

**Remarks:**

Saves the specified material library to the specified file. This method does not bring up a file dialog.

**Parameters:**

**const TCHAR \*name**

The material library to save.

This parameter is available in release 2.0 and later only.

Points to the material library to save from. If NULL the library is saved from the current material library. See [Class MtlBaseLib](#).

**Return Value:**

Nonzero if the library was saved; otherwise 0.

**Prototype:**

**virtual int MergeFromFile(const TCHAR \*name, BOOL mergeAll=FALSE, BOOL selMerged=FALSE, BOOL refresh=TRUE, int dupAction = MERGE\_DUPS\_PROMPT, NameTab\* mrgList=NULL)=0;**

**Remarks:**

Merges the specified MAX file into the current scene. In MAX 2.0 and later additional parameters allow automatic viewport updates to be optionally be turned off, cases of duplicate objects being merged may be handled, a table of names of the merged objects may be generated, and only specific named objects from the file may be merged.

**Parameters:**

**const TCHAR \*name**

The MAX file to merge.

**BOOL mergeAll=FALSE**

If TRUE all the items in the file are merged; otherwise the selector dialog appears allowing the user to choose.

**selMerged=FALSE**

If TRUE the nodes are selected when they are merged.

**BOOL refresh=TRUE**

This parameter is available in release 2.0 and later only.

Set this to FALSE to prevent the viewports from automatically being refreshed.

**int dupAction = MERGE\_DUPS\_PROMPT**

This parameter is available in release 2.0 and later only.

Determines what to do when duplicate named objects are encountered during the merge. One of the following values specifies what to do:

**MERGE\_DUPS\_PROMPT**

Prompt the user for what to do when duplicates are encountered.

**MERGE\_DUPS\_MERGE**

Merge new objects, but keep original objects when duplicates are encountered.

**MERGE\_DUPS\_SKIP**

Skip any duplicates encountered (don't merge them).

**MERGE\_DUPS\_DELOLD**

Delete the old objects and merge the new ones when duplicates are encountered.

**MERGE\_LIST\_NAMES**

Specifies that the name table **mrgList** below should be filled in with the names of the merged objects.

**NameTab\* mrgList=NULL**

This parameter is available in release 2.0 and later only.

When you specify a pointer to a **NameTab** for this parameter, and don't set **dupAction** to **MERGE\_LIST\_NAMES**, then this method will merge the nodes whose names are listed in the **mrgList**.

If **dupAction==MERGE\_LIST\_NAMES**, (and **mergeAll==TRUE**,

and **mrgList** != NULL) then this method puts a list of the nodes in the file into **mrgList**, and simply returns (no merging is done).

See [Class NameTab](#).

**Return Value:**

Nonzero if the file was merged; otherwise 0.

**Prototype:**

**virtual void FileOpen()=0;**

**Remarks:**

This brings up the standard MAX file open dialog and allows the user to load a new scene.

**Prototype:**

**virtual BOOL FileSave()=0;**

**Remarks:**

This saves the current file. If the file has not been saved yet (and is thus unnamed) this brings up the standard MAX file Save As dialog box and allows the user to choose a name.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual BOOL CheckForSave()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method can be used to write out a .MAX file if needed. First it ends animation if it's in progress. Next it determines whether a save operation is required (change marked in scene, or undo operations present).

**Return Value:**

If the save is not required, it returns TRUE; otherwise, it puts up a dialog box asking if the user wants to save. If the user picks No, it returns TRUE. If the user picks Cancel it returns FALSE. If the user picks Yes then the method proceeds as for **FileSave()** above.

**Prototype:****virtual BOOL FileSaveAs()=0;****Remarks:**

This brings up the standard MAX file saveas dialog box and allows the user to save the current scene to a new file.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:****virtual void FileSaveSelected()=0;****Remarks:**

This brings up the standard MAX file save selected dialog box and allows the user to save the selected items from the current scene.

**Prototype:****virtual void FileReset(BOOL noPrompt=FALSE)=0;****Remarks:**

This resets MAX to its startup condition after a confirmation prompt. This performs the same operation as choosing File / Reset from the MAX menus.

**Parameters:****BOOL noPrompt=FALSE**

If TRUE the confirmation prompt is not presented.

**Prototype:****virtual void FileMerge()=0;****Remarks:**

This allows the user to merge another MAX file with the current scene. This performs the same operation as choosing File / Merge from the MAX menus.

**Prototype:****virtual void FileHold()=0;****Remarks:**

This saves the current state of the scene to a temporary hold buffer (same as Edit / Hold). This state may later be restored using **FileFetch()** or (Edit / Fetch from MAX).

**Prototype:**

**virtual void FileFetch()=0;**

**Remarks:**

This restores the current state of the scene from the temporary hold buffer created using **FileHold()** (or from the MAX menu command Edit / Hold).

## **File Import / Export Related Methods**

**Prototype:**

**virtual BOOL FileImport()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method simply brings up the 'Select File to Import' dialog just as if the user picked this option from the File / Import... pulldown menu.

**Return Value:**

TRUE if the file was imported successfully; otherwise FALSE.

**Prototype:**

**virtual BOOL FileExport()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method simply brings up the 'Select File to Export' dialog just as if the user picked this option from the File / Export... pulldown menu.

**Return Value:**

TRUE if the export file was written successfully; otherwise FALSE.

**Prototype:**

**virtual BOOL ImportFromFile(const TCHAR \*name, BOOL suppressPrompts=FALSE, Class\_ID \*importerID=NULL)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method allows the import of the specified file by any of the supported import formats. This is done by specifying a full filename thus bypassing the file browser. If **suppressPrompts** is set to TRUE, the import mechanism will not display any prompts requiring user action.

Note: Developers of Import plug-ins need to support the **suppressPrompts** mechanism for this to work properly. Developers of such plug-ins should see the [Class SceneImport](#) for details.

**Parameters:**

**const TCHAR \*name**

The full file name (including extension -- which identifies the importer used) of the input file.

**BOOL suppressPrompts=FALSE**

If TRUE the default choices in the import plug-in are used and no options dialogs are presented. If FALSE any options dialogs provided by the importer are presented to the user.

**Class\_ID \*importerID=NULL**

This parameter is available in release 3.0 and later only.

The parameter specifies the Class\_ID of the import module to use. This is for those cases where more than one import module uses the same file extension. Omitting this class ID operates in the pre-R3 mode, i.e. the first importer found is used.

**Return Value:**

TRUE if the export file was written successfully; otherwise FALSE.

**Prototype:**

```
virtual BOOL ExportToFile(const TCHAR *name, BOOL  
suppressPrompts=FALSE, DWORD options, Class_ID  
*exporterID=NULL)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method allows the export of the current MAX file to any of the supported export formats. This is done by specifying a full filename thus bypassing the

file browser. If **suppressPrompts** is set to TRUE, the export mechanism will not display any prompts requiring user action.

Note: Developers of Export plug-ins need to support the **suppressPrompts** mechanism for this to work properly. Developers of such plug-ins should see the [Class SceneExport](#) for details.

#### **Parameters:**

##### **const TCHAR \*name**

The full file name (including extension -- which identifies the exporter used) of the output file.

##### **BOOL suppressPrompts=FALSE**

If TRUE the default choices in the export plug-in are used and no options dialogs are presented. If FALSE any options dialogs provided by the exporter are presented to the user.

##### **DWORD options**

This option is available in release 4.0 and later only.

There is currently one option; **SCENE\_EXPORT\_SELECTED** which allows you to export only selected nodes.

##### **Class\_ID \*exporterID=NULL**

This parameter is available in release 3.0 and later only.

The parameter specifies the Class\_ID of the export module to use. This is for those cases where more than one export module uses the same file extension. Omitting this class ID operates in the pre-R3 mode, i.e. the first exporter found is used.

#### **Return Value:**

TRUE if the export file was written successfully; otherwise FALSE.

#### **Prototype:**

```
virtual bool CanImportFile(const TCHAR* filename)=0;
```

#### **Remarks:**

This method is available in release 4.0 and later only.

This method will check is the specified file can be imported.

#### **Parameters:**

```
const TCHAR* filename
```

The file name to check.

**Return Value:**

TRUE if the specified file can be imported by one of the import plug-ins; otherwise FALSE.

**Prototype:**

**virtual BOOL GetImportZoomExtents()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This returns the state of the system zoom extents flag. Note that individual SceneImport plug-ins can override this in their **ZoomExtents()** method. See [Class SceneImport](#).

**Return Value:**

TRUE indicates that zoom extents will occur after imports, FALSE indicates that no zoom extents.

**Prototype:**

**virtual void SetImportZoomExtents(BOOL onOff)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Sets the state of the system zoom extents flag.

**Parameters:**

**BOOL onOff**

TRUE indicates that zoom extents will occur after imports, FALSE indicates that no zoom extents.

## **Fonts / Cursor Related Methods**

**Prototype:**

**virtual HFONT GetAppHFont()=0;**

**Remarks:**

Returns the handle of the font used by MAX. The default font is the same one returned from **GetAppHFont()**. So if a developer makes a dialog in the

rollup page the correct font will be used automatically.

**Prototype:**

**virtual HCURSOR GetSysCursor( int id )=0;**

**Remarks:**

This method returns the cursor handle for the standard MAX cursors. Use **SetCursor()** from the Windows API to set the cursor.

**Parameters:**

**int id**

One of the following values:

**SYSCUR\_MOVE, SYSCUR\_ROTATE, SYSCUR\_USCALE,  
SYSCUR\_NUSCALE,  
SYSCUR\_SQUASH, SYSCUR\_SELECT,  
SYSCUR\_DEFARROW.**

**Return Value:**

The handle of the cursor.

**Prototype:**

**virtual void SetCrossHairCur(BOOL onOff)=0;**

**Remarks:**

This method is reserved for future use.

**Prototype:**

**virtual BOOL GetCrossHairCur()=0;**

**Remarks:**

This method is reserved for future use.

## **Grid Related Methods**

**Prototype:**

**virtual float GetGridSpacing()=0;**

**Remarks:**

This method returns the grid spacing value that the user specifies in the Views / Grid and Snap Settings dialog in the Home Grid tab under Grid Spacing.

**Prototype:**

**virtual int GetGridMajorLines()=0;**

**Remarks:**

This method returns the value that the user specifies in the Views / Grid and Snap Settings dialog in the Home Grid tab under Major Lines every Nth.

**Prototype:**

**virtual void SetActiveGrid(INode \*node)=0;**

**Remarks:**

Sets the given node as the active grid object. This is used with grid helper objects to allow them to take effect. This method may also be used to activate the home grid.

Note: This API is not working in the SDK prior to version 2.5.

**Parameters:**

**INode \*node**

The node to set as the active grid object. To activate the home grid pass NULL.

**Prototype:**

**virtual INode \*GetActiveGrid()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the active grid node or NULL if the home grid is in use.

**Prototype:**

**virtual void AddGridToScene(INode \*node)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method is not operative in MAX (only in 3D Studio VIZ).

## Hide By Category Flags

### Prototype:

**virtual DWORD GetHideByCategoryFlags()=0;**

### Remarks:

Returns the state of the hide by category flags.

### Return Value:

One or more of the following values:

**HIDE\_OBJECTS**  
**HIDE\_SHAPES**  
**HIDE\_LIGHTS**  
**HIDE\_CAMERAS**  
**HIDE\_HELPERS**  
**HIDE\_WSMS**  
**HIDE\_SYSTEMS**  
**HIDE\_ALL**  
**HIDE\_NONE**

### Prototype:

**virtual void SetHideByCategoryFlags(DWORD f)=0;**

### Remarks:

Sets the state of the hide by category flags.

### Parameters:

**DWORD f**

One or more of the following values:

**HIDE\_OBJECTS**  
**HIDE\_SHAPES**  
**HIDE\_LIGHTS**  
**HIDE\_CAMERAS**  
**HIDE\_HELPERS**  
**HIDE\_WSMS**  
**HIDE\_SYSTEMS**  
**HIDE\_ALL**

## HIDE\_NONE

### Keyboard Accelerators

See the Advanced Topics section [Keyboard Accelerators and Dialog Messages](#) for more details. Also see the [Keyboard Shortcut System](#) topic for details of the system used in release 3.0 and later.

#### Prototype:

```
virtual void RegisterAccelTable(HWND hWnd, HACCEL  
hAccel)=0;
```

#### Remarks:

Registers a keyboard accelerator table.

#### Parameters:

**HWND hWnd**

The window handle.

**HACCEL hAccel**

The handle of the accelerator table (from the Windows API). See the Windows API for more details on accelerator tables.

#### Prototype:

```
virtual int UnRegisterAccelTable(HWND hWnd, HACCEL  
hAccel)=0;
```

#### Remarks:

Un-registers a keyboard accelerator table.

#### Parameters:

**HWND hWnd**

The window handle.

**HACCEL hAccel**

The handle of the accelerator table (from the Windows API). See the Windows API for more details on accelerator tables.

#### Return Value:

Nonzero if successful; otherwise 0.

**Prototype:**

**virtual BOOL GetPluginKeysEnabled()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Note that this method has been removed in release 4.0 and later.

In the MAX UI there is a icon button for toggling between the MAX system keyboard accelerators and those registered by the plug-in. This icon has a tooltip 'Plug-in Keyboard Shortcut Toggle'.

**Return Value:**

TRUE if the state is set such that the plug-in accelerators are enabled and FALSE if the system accelerators are enabled.

**Prototype:**

**virtual void SetPluginKeysEnabled(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Note that this method has been removed in release 4.0 and later.

In the MAX UI there is a icon button for toggling between the MAX system keyboard accelerators and those registered by plug-ins. This method may be used to set the state such that the plug-in accelerators are enabled or disabled.

**Note: Normally the user should decide which accelerators are enabled. The user will probably find it disconcerting to have a plug-in change the state of this button. Therefore developers should simply register their accelerators and let the user toggle them on and off.**

**Parameters:**

**BOOL onOff**

TRUE to enable plug-in accelerators; FALSE to enable system accelerators.

**Keyboard Shortcut Table Related Methods**

The keyboard shortcut system used prior to release 4 of MAX has been replaced by a more comprehensive system. See the Advanced Topics section [UI Customization](#) for details.

## Light Related Methods

### Prototype:

**virtual float GetLightConeConstraint()=0;**

### Remarks:

Returns the light cone constraint angle (in radians). This is the hotspot / falloff separation angle.

### Prototype:

**virtual void AddLightToScene(INode \*node)=0;**

### Remarks:

Adds a light to the scene and registers the light with the viewports.

### Parameters:

**INode \*node**

The light to add.

### Prototype:

**virtual void AddSFXRollupPage(ULONG flags=0)=0;**

### Remarks:

This method is available in release 3.0 and later only.

This method should be called in an light's **BeginEditParams()** method, after adding rollups to the modify panel: it puts up a rollup containing a list of all Atmospherics and Effects that use the current selected node as a "gizmo".

### Parameters:

**ULONG flags=0**

These are reserved for future use.

### Prototype:

**virtual void DeleteSFXRollupPage()=0;**

### Remarks:

This method is available in release 3.0 and later only.

This is called in a light's **EndEditParams()** when removing rollups.

**Prototype:**

**virtual void RefreshSFXRollupPage()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

An Atmospheric or Rendering Effect calls this when it adds or removes a "gizmo" reference. This is called to refresh the Special Effects rollup. The Atmospherics and Rendering Effects may also use the **REFMSG\_SFX\_CHANGE** message, though calling this method would have the same effect.

**Material Related Methods****Prototype:**

**virtual MtlBaseLib& GetMaterialLibrary()=0;**

**Remarks:**

This method provides access to the currently loaded material library.

**Return Value:**

See [Class MtlBaseLib](#).

**Prototype:**

**MtlBaseLib\* GetSceneMtls();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a list of all the materials used in the scene.

**Return Value:**

See [Class MtlBaseLib](#).

**Prototype:**

**MtlBase\* GetMtlSlot(int slot);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the material in the specified slot in the Materials Editor.

**Parameters:**

**int slot**

The number of the slot in the Materials Editor (a value in the range 0 to 23).

**Prototype:**

```
virtual BOOL OkMtlForScene(MtlBase *m)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Before assigning material to scene, call this to avoid duplicate names.

**Return Value:**

TRUE if it is okay to assign the material; FALSE if not.

**Prototype:**

```
virtual void AssignNewName(Mtl *m)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Modifies the name of the material to make it unique. The name is of the form "**Material #1**" where the number is incremented as required to make ensure it's unique.

**Parameters:**

**Mtl \*m**

The material whose name is modified.

**Prototype:**

```
virtual void AssignNewName(Texmap *m)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Modifies the name of the texture to make it unique. . The name is of the form "**Map #1**" where the number is incremented as required to make ensure it's unique.

**Parameters:**

**Texmap \*m**

The texmap whose name is modified.

**Prototype:**

```
void ActivateTexture(Texture *tx, Mtl *mtl, int subNum=-1);
```

**Remarks:**

This method is available in release 2.0 and later only.

The method activates the texture map in the viewports.

Note: In the MAX 2.0 SDK a bug prevents this function from being used -- it results in a link error. This is fixed in the 2.5 SDK.

**Parameters:**

**Texture \*tx**

Points to the texture map to activate.

**Mtl \*mtl**

The **top level** material containing the texture map.

**int subNum=-1**

If **mtl** above is a Multi-material, this specifies which sub-branch of the material contains **tx**.

**Prototype:**

```
void DeActivateTexture(Texture *tx, Mtl *mtl, int subNum=-1);
```

**Remarks:**

This method is available in release 2.0 and later only.

The method deactivates the texture map in the viewports.

Note: In the MAX 2.0 SDK a bug prevents this function from being used -- it results in a link error. This is fixed in the 2.5 SDK.

**Parameters:**

**Texture \*tx**

Points to the texture map to deactivate.

**Mtl \*mtl**

The **top level** material containing the texture map.

**int subNum=-1**

If **mtl** above it a Multi-material, this specifies which sub-branch of the material contains **tx**.

**Prototype:**

```
virtual void UpdateMtlEditorBrackets()=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

This method makes sure the Materials Editor slots correctly reflect which materials are used in the scene, which are used by selected objects, etc. This is used internally for the drag-and-drop of materials to nodes -- there is no reason why a plug-in developer should need to call it.

**Prototype:**

```
virtual void ConvertMtl(TimeValue t, Material &gm, Mtl *mtl,  
BOOL doTex, int subNum, float vis, BOOL &needDecal, INode  
*node, BitArray *needTex, GraphicsWindow *gw)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This function converts a material (class Mtl) to a viewport material (class Material).

**Parameters:****TimeValue t**

The time to convert the material.

**Material &gm**

The viewport material (output). See [Class Material](#).

**Mtl \*mtl**

The material to convert (input). See [Class Mtl](#).

**BOOL doTex**

Determines whether or not to include textures. TRUE for yes; FALSE for no.

**int subNum**

If the input material **mtl** is a sub-material then pass its subnum; otherwise pass 0.

**float vis**

The visibility value in the range of 0.0 (completely transparent) to 1.0 (fully opaque) for the viewport representation.

**BOOL &needDecal**

Pass TRUE if the texture needs decal mapping; otherwise FALSE.

### **INode \*node**

Points to the node that the material is assigned to. See [Class INode](#).

### **BitArray \*needTex**

A BitArray that returns which map channels are needed. The BitArray is enlarged if needed. See [Class BitArray](#).

### **GraphicsWindow \*gw**

The GraphicsWindow to do the conversion for. See [Class GraphicsWindow](#).

## **Modifier Related Methods**

### **Prototype:**

```
virtual void GetModContexts(ModContextList& list, INodeTab& nodes)=0;
```

### **Remarks:**

A modifier may be applied to several objects in the scene. This method retrieves a list of all the ModContexts for the modifier at the current place in the modifier stack. It also gets a list of the nodes the modifier is applied to. For example a modifier may store data into the local data portion of the ModContext for each object affected by the modifier. In order to get at this data it needs to get each ModContext.

Note: this method returns only the ModContext of the currently selected objects, even if the modifier is applied to several objects.

### **Parameters:**

#### **ModContextList& list**

A reference to the list of ModContexts. See [ModContextList](#).

#### **INodeTab& nodes**

A reference to the table of each of the nodes. See [INodeTab](#).

### **Prototype:**

```
virtual BaseObject* GetCurEditObject()=0;
```

### **Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the object (or modifier) that is currently being edited in

the modifier panel. See [Class BaseObject](#).

## Node Grouping

### Prototype:

```
virtual INode *GroupNodes(INodeTab *nodes=NULL,TSTR  
*name=NULL,BOOL selGroup=TRUE)=0;
```

### Remarks:

Combines the specified nodes into a MAX group.

### Parameters:

**INodeTab \*nodes=NULL**

The table of nodes to combine into a MAX group. If NULL is passed, the current selection set is used.

**TSTR \*name=NULL**

The name of the group to create. If the name is NULL, a dialog box will prompt the user to select a name.

**BOOL selGroup=TRUE**

If TRUE, the group of nodes will be selected after the operation is completed.

### Return Value:

A pointer to the group of nodes created.

### Prototype:

```
virtual void UngroupNodes(INodeTab *nodes=NULL)=0;
```

### Remarks:

Un-groups the specified nodes.

### Parameters:

**INodeTab \*nodes=NULL**

The table of nodes to un-group. If NULL is passed, the current selection set is used.

### Prototype:

```
virtual void ExplodeNodes(INodeTab *nodes=NULL)=0;
```

**Remarks:**

Explodes the grouped nodes. This completely un-groups nested groups.

**Parameters:**

**INodeTab \*nodes=NULL**

The table of nodes to explode. If NULL is passed, the current selection set is used.

**Prototype:**

**virtual void OpenGroup(INodeTab \*nodes=NULL,BOOL clearSel=TRUE)=0;**

**Remarks:**

Opens the grouped nodes. Items in an opened group may be edited individually.

**Parameters:**

**INodeTab \*nodes=NULL**

The table of nodes comprising a group that will be opened. If NULL is passed, the current selection set is used.

**BOOL selGroup=TRUE**

If TRUE, the group of nodes will be selected after the operation is completed.

**Prototype:**

**virtual void CloseGroup(INodeTab \*nodes=NULL,BOOL selGroup=TRUE)=0;**

**Remarks:**

Closes the specified group of nodes. Items in a closed group cannot be edited individually.

**Parameters:**

**INodeTab \*nodes=NULL**

The table of nodes to close. If NULL is passed, the current selection set is used.

**BOOL selGroup=TRUE**

If TRUE, the group of nodes will be selected after the operation is completed.

## Node Names -- Creating Unique

### Prototype:

```
virtual void MakeNameUnique(TSTR &name)=0;
```

### Remarks:

Given a name, this method will modify it to ensure it is unique. It does this by appending a two digit number to the end of the name, or incrementing the existing number, until the name created is unique. Important Note: See **NewNameMaker()** below for a much faster version of this method when creating **many** nodes whose names must be unique.

### Parameters:

**TSTR &name**

The name to make unique.

### Prototype:

```
virtual NameMaker* NewNameMaker(BOOL initFromScene =  
TRUE)=0;
```

### Remarks:

Returns a class used for efficiently creating unique names. To use, do the following:

```
NameMaker *nm = interface->NewNameMaker();  
for(;;) {  
    ...  
    nm->MakeUniqueName(nodename);  
    ...  
}  
delete nm;
```

### Parameters:

**BOOL initFromScene = TRUE**

This parameter is available in release 2.0 and later only.

If FALSE then the name maker is not seeded with the names of the objects in the current scene.

### Return Value:

See [Class NameMaker](#) for details.

## Node Related Methods

### Prototype:

```
virtual INode *GetINodeByName(const TCHAR *name)=0;
```

### Remarks:

This method may be called to retrieve the **INode** pointer for the node with the given name.

### Parameters:

**const TCHAR \*name**

The name of the node to find.

### Prototype:

```
virtual INode *GetINodeByHandle(ULONG handle)=0;
```

### Remarks:

This method is available in release 4.0 and later only.

Returns a pointer to the node whose handle is specified. In 3dsmax version 4.0 and later each node is assigned a unique handle.

### Parameters:

**ULONG handle**

The handle of the node to find.

### Prototype:

```
virtual void DeleteNode(INode *node, BOOL redraw=TRUE,  
BOOL overrideSlaves=FALSE)=0;
```

### Remarks:

This deletes the specified node from the scene and handles setting up for an undo.

In order to be able to undo this deletion properly the undo system must be in a holding state. A developer can do this by calling **theHold.Begin()** before calling **DeleteNode()** and calling **theHold.Accept()** after. For example, to delete the first node in the current selection set, you could use the following

code:

```
theHold.Begin();  
ip->DeleteNode(ip->GetSelNode(0));  
theHold.Accept(_T("Delete"));
```

The string used in the **Accept()** method is what appears to the user in the Undo menu choice.

Note that the object reference of the node is deleted as well if the only item referencing that object is the node. This is because when the node is deleted, it first deletes all its references. Whenever a reference is deleted, if the item is the last reference, then the system deletes the item as well.

### **Parameters:**

**INode \*node**

The node to delete.

**BOOL redraw=TRUE**

This parameter is available in release 2.0 and later only.

If FALSE the viewports will not be redrawn after the node is deleted. This allows several nodes to be deleted at once without the viewports having to be redrawn after each one.

**BOOL overrideSlaves=FALSE**

This parameter is available in release 2.0 and later only.

If TRUE then this method **will** delete nodes whose TM controllers implement the **Control::PreventNodeDeletion()** method to return TRUE. If a TM controller implements that method then normally **DeleteNode()** won't work on the nodes associated with that controller. However if this parameter is TRUE then these nodes still can be deleted. This allows a master controller to easily delete slaves nodes if it needs to.

### **Prototype:**

```
virtual INode *GetRootNode()=0;
```

### **Remarks:**

Returns a pointer to the root node. From the root node one can retrieve the children (using **INode::NumberOfChildren()**, and **INode::GetChildNode(i)**). This allows a developer enumerated the scene

by enumerating the node tree.

**Prototype:**

**virtual void NodeInvalidateRect( INode \*node )=0;**

**Remarks:**

This invalidates the rectangle in the viewports that the node occupies. Rectangles flagged as invalid will be updated on the next screen redraw.

**Parameters:**

**INode \*node**

Specifies the node whose rectangle is invalidated.

**Prototype:**

**virtual void SetNodeTMRelConstPlane(INode \*node, Matrix3& mat)=0;**

**Remarks:**

Sets the nodes transform relative to the current construction plane. This may be used during creating so you can set the position of the node in terms of the construction plane and not in world units.

**Parameters:**

**INode \*node**

The node whose transform will be set.

**Matrix3& mat**

The transformation matrix.

## **Node Selection Sets**

**Prototype:**

**virtual int GetSelNodeCount()=0;**

**Remarks:**

Returns the number of nodes in the selection set.

**Prototype:**

**virtual INode \*GetSelNode(int i)=0;**

**Remarks:**

Selection sets are handled as a virtual array and accessed using an index starting at 0. This method returns a pointer to the 'i-th' node in the current selection set. See also: **GetSelNodeCount()**.

**Parameters:****int i**

Index of the node in the selection set.

**Return Value:**

Pointer to the 'i-th' node in the selection set.

**Prototype:****virtual void GetSelectionWorldBox(TimeValue t,Box3 &box)=0;****Remarks:**

This method retrieves the world space bounding box of the current selection set.

**Parameters:****TimeValue t**

The time to retrieve the bounding box.

**Box3 &box**

The bounding box is returned here.

**Prototype:****virtual void SelectNode(INode \*node, int clearSel = 1)=0;****Remarks:**

This selects the specified node, either adding it to the current selection set, or creating a new selection set with the node.

Note: A developer should call **theHold.Begin()** before this method and **theHold.Accept()** after this call. Otherwise, MAX may crash if the user selects undo after this call has been issued. See the sample code below. Also see [Undo / Redo](#) for more details on this system.

**Parameters:****INode \*node**

The node to select.

**int clearSel = 1**

If zero, the node is added to the current selection set. If nonzero, the node replaces the selection set.

**Sample Code:**

```
theHold.Begin();  
ip->SelectNode(node);  
TSTR undostr; undostr.printf("Select");  
theHold.Accept(undostr);
```

**Prototype:**

```
virtual void DeSelectNode(INode *node)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method unselects the specified node.

**Parameters:**

**INode \*node**

The node to deselect.

**Prototype:**

```
virtual void SelectNodeTab(INodeTab &nodes,BOOL sel,BOOL  
redraw=TRUE)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method may be used to select or deselect a group of nodes all at once and optionally redraw the viewports when done.

**Parameters:**

**INodeTab &nodes**

The Table of nodes to select or deselect.

**BOOL sel**

If TRUE the nodes are selected; otherwise they are deselected.

**BOOL redraw=TRUE**

If TRUE the viewports are redrawn when done; otherwise they are not.

**Prototype:**

**virtual void ClearNodeSelection(BOOL redraw=TRUE)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method may be used to clear the current selection set and optionally redraw the viewports.

**Parameters:**

**BOOL redraw=TRUE**

If TRUE the viewports are redrawn when done; otherwise they are not.

**Prototype:**

**virtual BOOL SelectionFrozen()=0;**

**Remarks:**

In MAX the space bar freezes the selection set. This keeps the selection set from being inadvertently changed. This method access the status of the selection set frozen state.

**Return Value:**

TRUE if the selection set is frozen; otherwise FALSE.

**Prototype:**

**virtual void FreezeSelection()=0;**

**Remarks:**

Toggles the selection set to the frozen state.

**Prototype:**

**virtual void ThawSelection()=0;**

**Remarks:**

Toggles the selection set to the thawed state.

**Prototype:**

```
virtual bool CloneNodes(INodeTab& nodes, Point3& offset, bool  
expandHierarchies = true, CloneType cloneType = NODE_COPY,  
INodeTab* resultSource = NULL, INodeTab* resultTarget =  
NULL) = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to clone nodes and node hierarchies.

**Parameters:**

**INodeTab& nodes**

The node table containing the nodes you wish to clone.

**Point3& offset**

The position offset you wish to apply to the cloned nodes.

**bool expandHierarchies = true**

This determines if children will be cloned in hierarchies.

**CloneType cloneType = NODE\_COPY**

The type of cloning you wish to do, which is one of the following;

**NODE\_COPY**, **NODE\_INSTANCE** or **NODE\_REFERENCE**.

**INodeTab\* resultSource = NULL**

This node table will be filled in with the original nodes to be cloned. The reason for this is that there can be dependencies between nodes that cause other nodes to be added to the list. For example light/camera targets, nodes part of systems, belonging to groups or expanded hierarchies etc.

**INodeTab\* resultTarget = NULL**

This node table will be filled in with the new cloned nodes. There is a one to one relationship between the nodes in the resultSource and the resultTarget.

**Return Value:**

TRUE if the nodes were cloned successfully, otherwise FALSE. Note that with R4.0 the return value will always be TRUE since no actual error checking is undertaken.

## **Node Picking (interactive selection in the scene)**

**Prototype:**

```
virtual INode *PickNode(HWND hWnd, IPoint2
```

**pt, PickNodeCallback \*filt=NULL)=0;**

**Remarks:**

This method hit tests the screen position for nodes and returns an INode pointer if one is hit, NULL otherwise.

**Parameters:**

**HWND hWnd**

Handle of the window to check.

**IPoint2 pt**

Point to check in screen coordinates.

**PickNodeCallback \*filt=NULL**

This callback may be used to filter nodes from being picked. See [Class PickNodeCallback](#).

**Return Value:**

INode pointer of node that was hit; NULL if no node was found.

**Prototype:**

**virtual void BoxPickNode(ViewExp \*vpt, IPoint2 \*pt, BOOL crossing, PickNodeCallback \*filt=NULL)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method performs a node level hit test on the specified rectangular region. Use either the **ViewExp::GetClosestHit()** or **ViewExp::GetHit()** method to access the result.

**Parameters:**

**ViewExp \*vpt**

The viewport to perform the hit testing.

**IPoint2 \*pt**

These points specify the box region to hit test. The first point in the array is the lower left. The second point is the upper right.

**BOOL crossing**

If TRUE Crossing selection is used; otherwise Window selection is used.

**PickNodeCallback \*filt=NULL**

This callback may be used to filter nodes from being picked. See [Class PickNodeCallback](#).

**Prototype:**

```
virtual void CirclePickNode(ViewExp *vpt,IPoint2 *pt,BOOL crossing,PickNodeCallback *filt=NULL)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method performs a node level hit test on the specified circular region. Use either the **ViewExp::GetClosestHit()** or **ViewExp::GetHit(int i)** method to access the result.

**Parameters:**

**ViewExp \*vpt**

The viewport to perform the hit testing.

**IPoint2 \*pt**

These points specify the circular region to hit test. The first point in the array is the center. The second point is a point on the radius.

**BOOL crossing**

If TRUE Crossing selection is used; otherwise Window selection is used.

**PickNodeCallback \*filt=NULL**

This callback may be used to filter nodes from being picked. See [Class PickNodeCallback](#).

**Prototype:**

```
virtual void FencePickNode(ViewExp *vpt,IPoint2 *pt,BOOL crossing,PickNodeCallback *filt=NULL)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method performs a node level hit test on the specified arbitrary polygonal region. Use either the **ViewExp::GetClosestHit()** or **ViewExp::GetHit()** method to access the result.

**Parameters:**

## **ViewExp \*vpt**

The viewport to perform the hit testing.

## **IPoint2 \*pt**

These points specify the fence region to hit test. It is assumed the last point is connected to the first point to close the region. The fence knows when it's out of points when it hits a point that has negative **x** and **y** values. So, the initializer **IPoint2(-1, -1)** added to the bottom of this **IPoint2** list signals the end.

## **BOOL crossing**

If TRUE Crossing selection is used; otherwise Window selection is used.

## **PickNodeCallback \*filt=NULL**

This callback may be used to filter nodes from being picked. See [Class PickNodeCallback](#).

### **Prototype:**

```
virtual void SetPickMode(PickModeCallback *pc)=0;
```

### **Remarks:**

This will set the command mode to a standard pick mode. The callback implements hit testing and a method that is called when the user actually picks an item. Note that this method, if called a second time, will cancel the pick mode and put the user into 'Select and Move' mode. This can be used to handle the case where a user clicks on a user interface control a second time to cancel to picking.

Sample code using this API is available in

**\MAXSDK\SAMPLES\OBJECTS\MORPHOBJ.CPP**. A utility plug-in that uses this API is

**\MAXSDK\SAMPLES\UTILITIES\ASCIIOUT.CPP**.

### **Parameters:**

#### **PickModeCallback \*pc**

A pointer to an instance of the class [PickModeCallback](#).

### **Prototype:**

```
virtual void ClearPickMode()=0;
```

**Remarks:**

This method is called to make sure there are no pick modes in the command stack.

**Prototype:**

```
virtual void FlashNodes(INodeTab *nodes)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to 'flash' a group of nodes. This is usually used as a confirmation of some operation (for example as an indication of the completion of a pick node operation.) The nodes are briefly erased and then redrawn in the viewport to flash them.

**Parameters:**

```
INodeTab *nodes
```

Pointer to the table of nodes to 'flash'.

**Sample Code:**

```
INodeTab flash;  
INode *node;  
for (int i=0; i<ip->GetSelNodeCount(); i++) {  
    node = ip->GetSelNode(i);  
    flash.Append(1,&node,10);  
}  
ip->FlashNodes(&flash);
```

**IObjCreate and IObjParam Pointer Casting****Prototype:**

```
virtual IObjCreate *GetIObjCreate()=0;
```

**Remarks:**

Returns this interface pointer cast as a **IObjCreate** pointer.

**Prototype:**

```
virtual IObjParam *GetIObjParam()=0;
```

**Remarks:**

Returns this interface pointer cast as a **IObjParam** pointer.

**Object Snap Methods****Prototype:**

**virtual IOsnapManager \*GetOsnapManager()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns an interface pointer to the object snap manager. See [Class IOsnapManager](#). Also see the Advanced Topics section on [Snapping](#).

**Prototype:**

**virtual MouseManager \*GetMouseManager()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This is used internally by the Osnap Manager. Plug-Ins don't need to use this method.

**Prototype:**

**virtual void InvalidateOsnapdraw()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This is used internally to invalidate the osnap drawing mechanism. Plug-Ins don't need to use this method.

**Preview Creation****Prototype:**

**virtual void CreatePreview(PreviewParams \*pvp=NULL)=0;**

**Remarks:**

This method is used to render a preview image (or animation) from the currently active viewport using the real-time (viewport) renderer.

This method is available in release 2.0 and later only.

**Parameters:**

**PreviewParams \*pvp=NULL**

This class defines the way the preview is generated (via its data members). If this is passed as NULL the parameters from the preview rendering dialog box are used. See [Class PreviewParams](#).

## Progress Bar Methods

**Prototype:**

**virtual BOOL ProgressStart(TCHAR \*title, BOOL dispBar, LPTHREAD\_START\_ROUTINE fn, LPVOID arg)=0;**

**Remarks:**

This method puts up a progress bar in the status panel. The function **fn** is then called and is passed the argument **arg**. This function **fn** does the processing work of the plug-in. See the sample code below.

Note: It is not possible to use the progress bar APIs in the create or modify branches of the command panel. In the create or modify branch of the command panel, **EndEditParams()** gets called from this method. This is because most of MAX is "shut down" during a progress operation. For example, it is not appropriate for users to be moving objects during an IK calculation, or changing the camera lens while a preview is being created (all these operations use the **ProgressStart()** API). It is for this same reason that the Transform Type-in, Medit, Track View and Video Post windows are hidden during a progress operation. Since **EndEditParams()** is called, the Interface pointer a plug-in maintains is no longer valid after the operation is started. This method is useful for Utility plug-ins, and Import/Export plug-ins however.

**Parameters:**

**TCHAR \*title**

The title string for the progress bar to let the user know what is happening.

**BOOL dispBar**

If FALSE the progress bar is not displayed; if TRUE the progress bar is displayed.

## **LPTHREAD\_START\_ROUTINE fn**

This is a pointer to a function that returns a **DWORD** and takes a single argument. When **ProgressStart()** is called, this function is called with the argument **arg** passed. This function should be declared as follows:

```
DWORD WINAPI fn(LPVOID arg)
```

## **LPVOID arg**

This is the argument to the function **fn**.

## **Return Value:**

TRUE means the progress mode was entered successfully. FALSE means that there was a problem. Currently the return value is always TRUE. When compiled for multi-threading (which is turned off in the 1.0 build), FALSE is returned if the new thread could not be created.

## **Sample Code:**

This code demonstrates the use of the **ProgressStart()**, **ProgressEnd()**, **GetCancel()** and **SetCancel()** APIs. There are two ways to use the APIs. One is to use a dummy function for **fn** in **ProgressStart()**. Then just call **ProgressUpdate()** from whatever function you want to do the processing. The other way is to use **fn** to do the processing.

In this version, the **fn** function passed to **ProgressStart()** performs the work.

```
DWORD WINAPI fn(LPVOID arg) {  
    int i, percent;  
    Interface *ip = theUtility.ip;  
    for (i = 0; i < 1000; i++) {  
        percent = i/10;  
        ip->ProgressUpdate(percent);  
        if (ip->GetCancel()) {  
            switch(MessageBox(ip->GetMAXHwnd(), _T("Really Cancel"),  
                _T("Question"), MB_ICONQUESTION | MB_YESNO)) {  
                case IDYES:  
                    return(0);  
                case IDNO:  
                    ip->SetCancel(FALSE);  
            }  
        }  
    }
```

```

}
return(0);
}
void Utility::Test1() {
    Interface *ip = theUtility.ip;
    LPVOID arg;
    ip->ProgressStart(_T("Title String"), TRUE, fn, arg);
    ip->ProgressEnd();
}

```

In this version, the **fn** function passed to **ProgressStart()** is a dummy function, and the processing is done outside it.

```

DWORD WINAPI fn(LPVOID arg) {
    return(0);
}
void Utility::Test1() {
    int i, percent, retval;
    Interface *ip = theUtility.ip;
    LPVOID arg;
    ip->ProgressStart(_T("Title String"), TRUE, fn, arg);
    for (i = 0; i < 1000; i++) {
        percent = i/10;
        ip->ProgressUpdate(percent);
        if (ip->GetCancel()) {
            retval = MessageBox(ip->GetMAXHwnd(), _T("Really Cancel"),
                _T("Question"), MB_ICONQUESTION | MB_YESNO);
            if (retval == IDYES)
                break;
            else if (retval == IDNO)
                ip->SetCancel(FALSE);
        }
    }
    ip->ProgressEnd();
}

```

**Prototype:**

```

virtual void ProgressUpdate(int pct, BOOL showPct = TRUE,

```

**TCHAR \*title = NULL)=0;**

**Remarks:**

This method updates the progress bar. As the function **fn** passed in **ProcessStart()** above is working it should periodically call this method to report its progress.

**Parameters:**

**int pct**

The percentage complete (0 to 99). This is what causes the progress bar to move.

**BOOL showPct = TRUE**

If TRUE, then the **title** parameter is ignored, and a percent is displayed to the right of the progress bar. If FALSE, then the **title** parameter is displayed next to the progress bar. This is for operations that are discrete -- the title might change from "extruding" to "capping" to "welding" for example. Note that currently the Cancel button is not shown if **showPct** is set to FALSE, however the ESC key may be used to cancel.

**TCHAR \*title = NULL**

If **showPct** is FALSE, this string is displayed next to the progress bar.

**Prototype:**

**virtual void ProgressEnd()=0;**

**Remarks:**

This method removes the progress bar and frees the memory that was allocated internally to handle the processing.

**Prototype:**

**virtual BOOL GetCancel()=0;**

**Remarks:**

This method returns the progress bar cancel button status. Also see **SetCancel()** below.

**Return Value:**

TRUE if the user pressed the cancel button; otherwise FALSE.

**Prototype:**

```
virtual void SetCancel(BOOL sw)=0;
```

**Remarks:**

Sets the canceled status returned from **GetCancel()**. This may be used if you want to give the user a confirmation dialog box asking if they really want to cancel. For example, when a MAX user creates an preview animation this API is used. If the user presses cancel, the preview code reads this via

**GetCancel()**. Then a confirmation dialog is displayed asking the user if they indeed want to cancel. If the user selects that they don't want to cancel, this method is called passing FALSE. This sets the class variable that is returned by **GetCancel()**. The cancel request is ignored and processing continues.

Note that this will not cancel unless you implement code to process **GetCancel()**. It merely sets the state returned from **GetCancel()**.

**Parameters:**

**BOOL sw**

TRUE to set the cancel flag; FALSE to clear the cancel flag.

**Renderer Access**

The following APIs provide a simplistic method to call the renderer and render frames (to gain more control over the renderer, see the methods after these first three). The renderer just uses the current user specified parameters. Note that the renderer uses the width, height, and aspect of the specified bitmap so the caller can control the size of the rendered image rendered. Also Note: These methods drive the renderer and not video post.

**Prototype:**

```
virtual int OpenCurRenderer(INode *camNode,ViewExp  
*view,RenderType t = RENDTYPE_NORMAL, int w=0, int h=0)=0;
```

**Remarks:**

This method is called to open the current renderer. It must be opened before frames can be rendered. Either **camNode** or **view** must be non-NULL, but not both. Remember to close the renderer when you are done (using **CloseCurRenderer()**).

**Parameters:**

**INode \*camNode**

A pointer to the camera node to render, or NULL if a viewport should be rendered.

**ViewExp \*view**

A pointer to the view to render, or NULL if the camera should be rendered.

**RendType t = RENDTYPE\_NORMAL**

This parameter is available in release 3.0 and later only.

This provides an optional way to specify the view when opening the renderer.

This specifies the type of render. See [List of Render Types](#).

**int w=0**

This parameter is available in release 3.0 and later only.

This specifies the width of the rendering.

**int h=0**

This parameter is available in release 3.0 and later only.

This specifies the height of the rendering.

**Return Value:**

Nonzero indicates success; failure is zero.

**Prototype:**

```
virtual int OpenCurRenderer(ViewParams *vpar,RendType t =  
RENDTYPE_NORMAL, int w=0, int h=0)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to open the current renderer. It provides an optional way to specify the view when opening.

**Parameters:****ViewParams \*vpar**

This class describes the properties of a view that is being rendered. See [Class ViewParams](#).

**RendType t = RENDTYPE\_NORMAL**

This parameter is available in release 3.0 and later only.

This provides an optional way to specify the view when opening the renderer.

This specifies the type of render. See [List of Render Types](#).

**int w=0**

This parameter is available in release 3.0 and later only.

This specifies the width of the rendering.

**int h=0**

This parameter is available in release 3.0 and later only.

This specifies the height of the rendering.

**Return Value:**

Nonzero indicates success; failure is 0.

**Prototype:**

**virtual void CloseCurRenderer()=0;**

**Remarks:**

This method is called to close the renderer. The renderer must be closed when you are finished with it.

**Prototype:**

**virtual int CurRendererRenderFrame(TimeValue t,Bitmap \*bm,RendProgressCallback \*prog=NULL, float frameDur = 1.0f , ViewParams \*vp=NULL, RECT \*regionRect = NULL)=0;**

**Remarks:**

This method is called to render a frame to the given bitmap. The renderer uses the width, height, and aspect ratio of the specified bitmap to control the size of the rendered image.

**Parameters:**

**TimeValue t**

The time to render the image.

**Bitmap \*bm**

The bitmap to render to. This bitmap defines the size and aspect ratio of the render. See [Class Bitmap](#).

**RendProgressCallback \*prog=NULL**

The RendProgressCallback is an optional callback. See [Class RendProgressCallback](#).

**float frameDur = 1.0f**

This parameter should always be set to 1.0.

**ViewParams \*vp=NULL**

This parameter allows you to specify a different view transformation on each render call. For instance, you can render a given scene at a given time from many different viewpoints, without calling **Render::Open()** for each one.

**RECT \*regionRect = NULL**

This parameter is available in release 3.0 and later only.

This value, if passed, defines the region to be rendered. This only works for **RENDTYPE\_REGION** and **RENDTYPE\_REGIONCROP**.

**Return Value:**

The result of the render - Nonzero if success; otherwise 0.

**Prototype:**

**virtual IScanRenderer \*CreateDefaultScanlineRenderer()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method creates a default scanline renderer. This renderer must be deleted by calling **IScanRenderer::DeleteThis()**.

**Return Value:**

A pointer to a new IScanRenderer object.

**Prototype:**

**virtual BOOL DisplayActiveCameraViewWithMultiPassEffect() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method will draw the active view (if it is a camera view with a multi-pass effect enabled) with that effect active.

**Return Value:**

TRUE if the active view is a camera view with a multi-pass effect enabled, FALSE otherwise.

## Plug-In Renderer Access

Below is a set of functions parallel to those above to work with any `Renderer` instance.

### Prototype:

```
virtual int OpenRenderer(Renderer *pRenderer, INode  
*camNode, ViewExp *view, RendType type =  
RENDTYPE_NORMAL, int w=0, int h=0)=0;
```

### Remarks:

This method is available in release 4.0 and later only.

This method is called to open the specified renderer. It must be opened before frames can be rendered. Either **camNode** or **view** must be non-NULL, but not both. Remember to close the renderer when you are done (using **CloseRenderer()**).

### Parameters:

#### **Renderer \*pRenderer**

Points to the renderer to open.

#### **INode \*camNode**

A pointer to the camera node to render, or NULL if a viewport should be rendered.

#### **ViewExp \*view**

A pointer to the view to render, or NULL if the camera should be rendered.

#### **RendType type = RENDTYPE\_NORMAL**

This parameter is available in release 3.0 and later only.

This provides an optional way to specify the view when opening the renderer.

This specifies the type of render. See [List of Render Types](#).

#### **int w=0**

This specifies the width of the rendering.

#### **int h=0**

This specifies the height of the rendering.

### Return Value:

Nonzero indicates success; failure is zero.

**Prototype:**

```
virtual int OpenRenderer(Renderer *pRenderer, ViewParams *vpar, RendType type = RENDTYPE_NORMAL, int w=0, int h=0)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is called to open the specified renderer. It provides an optional way to specify the view when opening.

**Parameters:****Renderer \*pRenderer**

Points to the renderer to do the rendering.

**ViewParams \*vpar**

This class describes the properties of a view that is being rendered. See [Class ViewParams](#).

**RendType type = RENDTYPE\_NORMAL**

This parameter is available in release 3.0 and later only.

This provides an optional way to specify the view when opening the renderer.

This specifies the type of render. See [List of Render Types](#).

**int w=0**

This specifies the width of the rendering.

**int h=0**

This specifies the height of the rendering.

**Return Value:**

Nonzero indicates success; failure is 0.

**Prototype:**

```
virtual int RendererRenderFrame(Renderer *pRenderer, TimeValue t, Bitmap *bm, RendProgressCallback *prog=NULL, float frameDur = 1.0f, ViewParams *vp=NULL, RECT *regionRect = NULL)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is called to render a frame with the specified renderer to the given

bitmap. The renderer uses the width, height, and aspect ratio of the specified bitmap to control the size of the rendered image.

**Parameters:**

**Renderer \*pRenderer**

Points to the renderer which will do the rendering.

**TimeValue t**

The time to render the image.

**Bitmap \*bm**

The bitmap to render to. This bitmap defines the size and aspect ratio of the render. See [Class Bitmap](#).

**RendProgressCallback \*prog=NULL**

The RendProgressCallback is an optional callback. See [Class RendProgressCallback](#).

**float frameDur = 1.0f**

This parameter should always be set to 1.0.

**ViewParams \*vp=NULL**

This parameter allows you to specify a different view transformation on each render call. For instance, you can render a given scene at a given time from many different viewpoints, without calling **Render::Open()** for each one.

**RECT \*regionRect = NULL**

This parameter is available in release 3.0 and later only.

This value, if passed, defines the region to be rendered. This only works for **RENDTYPE\_REGION** and **RENDTYPE\_REGIONCROP**.

**Return Value:**

The result of the render - Nonzero if success; otherwise 0.

**Prototype:**

**virtual void CloseRenderer(Renderer \*pRenderer)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Closes the specified renderer.

**Parameters:**

**Renderer \*pRenderer**

Points to the renderer to close.

To get more control over the renderer, the renderer can be called directly. The following methods give access to the current renderer and the user's current rendering settings. Note: These methods drive the renderer and not video post.

**Prototype:**

**virtual Renderer \*GetCurrentRenderer()=0;**

**Remarks:**

Retrieves a pointer to the renderer currently set as the active renderer. This will be either the production renderer or the draft renderer depending upon which is active. A developer can determine which renderer this is by calling the **ClassID()** method of the renderer.

**Return Value:**

A pointer to the renderer.

**Prototype:**

**virtual Renderer \*GetProductionRenderer()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Retrieves a pointer to the renderer currently set as the production renderer. A developer can determine which renderer this is by calling the **ClassID()** method of the renderer.

**Return Value:**

A pointer to the renderer.

**Prototype:**

**virtual Renderer \*GetDraftRenderer()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Retrieves a pointer to the renderer currently set as the draft renderer. A developer can determine which renderer this is by calling the **ClassID()** method of the renderer.

**Return Value:**

A pointer to the renderer.

**Prototype:**

```
virtual Renderer *GetRenderer(RenderSettingID  
renderSettingID)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the specified renderer.

**Parameters:**

**RenderSettingID renderSettingID**

One of these values: See [List of Render Setting IDs](#)

**Return Value:**

A pointer to the renderer.

**Prototype:**

```
virtual void AssignCurRenderer(Renderer *rend)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Assigns the renderer passed for use as either the draft renderer or the production renderer depending upon which is active.

**Parameters:**

**Renderer \*rend**

The renderer to assign.

**Prototype:**

```
virtual void AssignProductionRenderer(Renderer *rend)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Assigns the renderer passed as the production renderer.

**Parameters:**

**Renderer \*rend**

The renderer to assign.

**Prototype:**

**virtual void AssignDraftRenderer(Renderer \*rend)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Assigns the renderer passed as the draft renderer.

**Parameters:**

**Renderer \*rend**

The renderer to assign.

**Prototype:**

**virtual void AssignRenderer(RenderSettingID renderSettingID,  
Renderer \*rend)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Assigns the renderer passed as one of the standard MAX rendering options (Production, Draft, etc).

**Parameters:**

**RenderSettingID renderSettingID**

One of these values: See [List of Render Setting IDs](#)

**Renderer \*rend**

Points to the renderer to assign.

**Prototype:**

**virtual void SetUseDraftRenderer(BOOL b)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Specifies which renderer is active -- draft or production. Pass TRUE to use the draft renderer and FALSE to get the production renderer.

**Prototype:**

**virtual BOOL GetUseDraftRenderer()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Determines which renderer is active -- draft or production.

**Return Value:**

TRUE for the draft renderer and FALSE for the production renderer.

**Prototype:**

**virtual void ChangeRenderSetting(RenderSettingID  
renderSettingID)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Sets the specified renderer as active.

**Parameters:**

**RenderSettingID renderSettingID**

One of these values: See [List of Render Setting IDs](#).

**Prototype:**

**virtual RenderSettingID GetCurrentRenderSetting()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a value which indicates which renderer is current. See [List of Render Setting IDs](#).

**Prototype:**

**virtual IRenderElementMgr \*GetCurRenderElementMgr()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns a pointer to the current render elements manager interface.

**Prototype:**

## **virtual IRenderElementMgr**

**\*GetRenderElementMgr(RenderSettingID renderSettingID)=0;**

### **Remarks:**

This method is available in release 4.0 and later only.

This method returns a pointer to the production or draft render element manager -- passing in renderSettingID = RS\_IReshade will return NULL.

### **Parameters:**

**RenderSettingID renderSettingID**

One of these values: See [List of Render Setting IDs](#).

### **Prototype:**

**virtual void SetupRendParams(RendParams &rp, ViewExp \*vpt, RendType t = RENDTYPE\_NORMAL)=0;**

### **Remarks:**

This method is called to fill in a **RendParams** structure that can be passed to the renderer with the user's current rendering settings. This is whatever was last used, or the default settings.

In MAX 1.x note the following (this needs not be done in MAX 2.0 or later):

In order to open a renderer using this method to setup the **RendParams** class the following code should be used:

```
RendParams rp;  
SetupRendParams(rp, );  
rp.atmos = NULL;  
rp.envMap = NULL;
```

As shown above, this method does not automatically set the values for **envMap** and **atmos**. You must do this manually if you are using the **RendParams** object to initialize the renderer. Then you can call **Renderer::Open(, rp, )**. In MAX 2.0 and later, **atmos** and **envMap** are properly initialized without the above code.

### **Parameters:**

**RendParams &rp**

This is the class instance whose data is filled in. See [Class RendParams](#).

### **ViewExp \*vpt**

This pointer only needs to be passed in if the RendType is **RENDTYPE\_REGION** or **RENDTYPE\_BLOWUP**. In these cases it will set up the **RendParams regxmin, regxmax, regymin, regymax** from values stored in the viewport. See [Class ViewExp](#).

### **RendType t = RENDTYPE\_NORMAL**

See [List of Render Types](#).

### **Prototype:**

```
virtual void GetViewParamsFromNode(INode* vnode,  
ViewParams& vp, TimeValue t)=0;
```

### **Remarks:**

This method is available in release 4.0 and later only.

This method fills in the specified ViewParams structure based on type of node passed (camera or light). It can be used when instantiating a renderer and calling Open(), Render(), and Close() directly on it.

### **Parameters:**

#### **INode\* vnode**

Points to the node to initialize from.

#### **ViewParams& vp**

The ViewParams structure to initialize.

#### **TimeValue t**

The time at which to initialize the structure.

### **Prototype:**

```
virtual BOOL CheckForRenderAbort()=0;
```

### **Remarks:**

This method is available in release 3.0 and later only.

This method may be called during a rendering to check if user has cancelled the render.

### **Return Value:**

TRUE if user has cancelled; otherwise FALSE. If not rendering the method returns FALSE.

These methods give access to the individual user specified render parameters. These are either parameters that the user specifies in the render dialog or the renderer page of the preferences dialog.

**Prototype:**

**virtual int GetRendTimeType()=0;**

**Remarks:**

Retrieves the type of time range to be rendered.

**Return Value:**

One of the following values:

**REND\_TIMESINGLE**

A single frame.

**REND\_TIMESEGMENT**

The active time segment.

**REND\_TIMERANGE**

The user specified range.

**REND\_TIMEPICKUP**

The user specified frame pickup string (for example "1,3,5-12").

**Prototype:**

**virtual void SetRendTimeType(int type)=0;**

**Remarks:**

Sets the type of time range rendered.

**Parameters:**

**int type**

One of the following values:

**REND\_TIMESINGLE**

A single frame.

**REND\_TIMESEGMENT**

The active time segment.

**REND\_TIMERANGE**

The user specified range.

**REND\_TIMEPICKUP**

The user specified frame pickup string (for example "1,3,5-12").

**Prototype:**

**virtual TimeValue GetRendStart()=0;**

**Remarks:**

Retrieves the renderer's start time setting.

**Prototype:**

**virtual void SetRendStart(TimeValue start)=0;**

**Remarks:**

Sets the renderer's start time setting.

**Parameters:**

**TimeValue start**

The time to begin rendering.

**Prototype:**

**virtual TimeValue GetRendEnd()=0;**

**Remarks:**

Retrieves the renderer's end time setting.

**Prototype:**

**virtual void SetRendEnd(TimeValue end)=0;**

**Remarks:**

Sets the renderer's end time setting.

**Parameters:**

**TimeValue end**

The time to end rendering.

**Prototype:**

**virtual int GetRendNThFrame()=0;**

**Remarks:**

Returns the renderer's 'n-th' frame setting.

**Prototype:**

**virtual void SetRendNThFrame(int n)=0;**

**Remarks:**

Sets the renderer's 'n-th' frame setting.

**Parameters:**

**int n**

The n-th frame setting.

**Prototype:**

**virtual BOOL GetRendShowVFB()=0;**

**Remarks:**

Retrieves the state of the renderer's show virtual frame buffer flag. Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendShowVFB(BOOL onOff)=0;**

**Remarks:**

Sets the state of the renderer's show virtual frame buffer flag.

**Parameters:**

**BOOL onOff**

TRUE is on; FALSE is off.

**Prototype:**

**virtual BOOL GetRendSaveFile()=0;**

**Remarks:**

Retrieves the state of the renderer's save file flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendSaveFile(BOOL onOff)=0;**

**Remarks:**

Sets the state of the renderer's save file flag.

**Parameters:**

**BOOL onOff**

TRUE is on; FALSE is off.

**Prototype:**

**virtual BOOL GetRendUseDevice()=0;**

**Remarks:**

Retrieves the state of the renderer's use device flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendUseDevice(BOOL onOff)=0;**

**Remarks:**

Sets the state of the renderer's use device flag.

**Parameters:**

**BOOL onOff**

TRUE is on; FALSE is off.

**Prototype:**

**virtual BOOL GetRendUseNet()=0;**

**Remarks:**

Retrieves the state of the renderer's use net flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendUseNet(BOOL onOff)=0;**

**Remarks:**

Sets the state of the renderer's use net flag.

**Parameters:****BOOL onOff**

TRUE is on; FALSE is off.

**Prototype:**

**virtual BitmapInfo& GetRendFileBI()=0;**

**Remarks:**

Retrieves the rendering file BitmapInfo. This class describes the output file. See [Class BitmapInfo](#).

**Prototype:**

**virtual BitmapInfo& GetRendDeviceBI()=0;**

**Remarks:**

Retrieves the rendering device BitmapInfo. This class describes the output device. See [Class BitmapInfo](#).

**Prototype:**

**virtual int GetRendWidth()=0;**

**Remarks:**

Retrieves the rendering output width in pixels.

**Prototype:**

**virtual void SetRendWidth(int w)=0;**

**Remarks:**

Sets the rendering output width.

**Parameters:****int w**

The width in pixels.

**Prototype:**

**virtual int GetRendHeight()=0;**

**Remarks:**

Retrieves the rendering output height in pixels.

**Prototype:**

**virtual void SetRendHeight(int h)=0;**

**Remarks:**

Sets the rendering output height.

**Parameters:**

**int h**

The height in pixels.

**Prototype:**

**virtual float GetRendAspect()=0;**

**Remarks:**

Retrieves the renderer's pixel aspect ratio setting. Note for MAX 1.2: To get the 'Image Aspect Ratio' setting use:

**float aspectRatio =  
((float) ip->GetRendWidth())/((float) ip->GetRendHeight());**

In MAX 2.0 or later **GetRendImageAspect()** may be used:

**Prototype:**

**virtual void SetRendAspect(float a)=0;**

**Remarks:**

Sets the renderer's pixel aspect ratio setting.

**Parameters:**

**float a**

The pixel aspect ratio to set.

**Prototype:**

**virtual float GetRendImageAspect()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

**Return Value:**

Returns the image aspect ratio.

**Prototype:**

**virtual float GetRendApertureWidth()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

**Return Value:**

Returns the aperture width in millimeters (mm).

**Prototype:**

**virtual void SetRendApertureWidth(float aw)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the aperture width.

**Parameters:**

**float aw**

The width to set in millimeters (mm).

**Prototype:**

**virtual BOOL GetRendFieldRender()=0;**

**Remarks:**

Retrieves the renderer's field render flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendFieldRender(BOOL onOff)=0;**

**Remarks:**

Sets the renderer's field render flag.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRenderColorCheck()=0;**

**Remarks:**

Retrieves the renderer's color check flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRenderColorCheck(BOOL onOff)=0;**

**Remarks:**

Sets the renderer's color check flag.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRenderSuperBlack()=0;**

**Remarks:**

Retrieves the renderer's super black flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRenderSuperBlack(BOOL onOff)=0;**

**Remarks:**

Sets the renderer's super black flag.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRendHidden()=0;**

**Remarks:**

Retrieves the renderer's render hidden objects flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendHidden(BOOL onOff)=0;**

**Remarks:**

Sets the renderer's render hidden objects flag.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRendForce2Side()=0;**

**Remarks:**

Retrieves the renderer's force two-sided flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendForce2Side(BOOL onOff)=0;**

**Remarks:**

Sets the renderer's force two-sided flag. TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRendAtmosphere()=0;**

**Remarks:**

Retrieves the renderer's uses atmospheric effects flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendAtmosphere(BOOL onOff)=0;**

**Remarks:**

Sets if the renderer uses atmospheric effects.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRendEffects()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

**Return Value:**

Returns TRUE if Rendering Effects will be used; otherwise FALSE.

**Prototype:**

**virtual void SetRendEffects(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets if Rendering Effects will be used.

**Parameters:**

**BOOL onOff**

TRUE to use (on); FALSE to not use (off).

**Prototype:**

**virtual BOOL GetRendDisplacement()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

**Return Value:**

Returns TRUE if rendering displacements is enabled; otherwise FALSE.

Note: Developers should use the **flags** parameter of the **View** class which is passed into **GetRenderMesh()** to determine if Displacement Mapping is being used because the values may not be the same (for instance when rendering in the Materials Editor). See [Class View](#).

**Prototype:**

**virtual void SetRendDisplacement(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets if rendering displacements are enabled.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual TSTR& GetRendPickFramesString()=0;**

**Remarks:**

Retrieves the string holding the frames the user wants to render. For example "1,3,5-12".

**Prototype:**

**virtual BOOL GetRendDitherTrue()=0;**

**Remarks:**

Retrieves the renderer's dither true color flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendDitherTrue(BOOL onOff)=0;**

**Remarks:**

Sets the renderer's dither true color flag.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRendDither256()=0;**

**Remarks:**

Retrieves the renderer's dither 256 color flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendDither256(BOOL onOff)=0;**

**Remarks:**

Sets the renderer's dither 256 color flag.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRendMultiThread()=0;**

**Remarks:**

Retrieves the renderer's multi-threaded flag.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendMultiThread(BOOL onOff)=0;**

**Remarks:**

Sets the renderer's multi-threaded flag.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRendNThSerial()=0;**

**Remarks:**

This retrieves the output file sequencing nth serial numbering setting.

**Return Value:**

Returns TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRendNThSerial(BOOL onOff)=0;**

**Remarks:**

This sets the output file sequencing nth serial numbering setting.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual int GetRendVidCorrectMethod()=0;**

**Remarks:**

Retrieves the video color check method.

**Return Value:**

One of the following values:

**0** is FLAG

**1** is SCALE\_LUMA

**2** is SCALE\_SAT

**Prototype:**

**virtual void SetRendVidCorrectMethod(int m)=0;**

**Remarks:**

Sets the video color check method.

**Parameters:**

**int m**

One of the following values:

**0** is FLAG

**1** is SCALE\_LUMA

**2** is SCALE\_SAT

**Prototype:**

**virtual int GetRendFieldOrder()=0;**

**Remarks:**

Retrieves the rendering field order.

**Return Value:**

One of the following values:

**0** is Even

**1** is Odd

**Prototype:**

**virtual void SetRendFieldOrder(int fo)=0;**

**Remarks:**

Sets the rendering field order to even or odd.

**Parameters:**

**int fo**

One of the following values:

**0** sets Even

**1** sets Odd

**Prototype:**

**virtual int GetRendNTSC\_PAL()=0;**

**Remarks:**

Retrieves the video color check NTSC or PAL setting.

**Return Value:**

One of the following values:

**0** is NTSC

**1** is PAL

**Prototype:**

**virtual void SetRendNTSC\_PAL(int np)=0;**

**Remarks:**

Sets the video color check NTSC or PAL setting.

**Parameters:**

**int np**

One of the following values:

**0** sets NTSC

**1** sets PAL

**Prototype:**

**virtual int GetRendSuperBlackThresh()=0;**

**Remarks:**

Returns the super black threshold setting.

**Prototype:**

**virtual void SetRendSuperBlackThresh(int sb)=0;**

**Remarks:**

Sets the super black threshold setting.

**Parameters:**

**int sb**

The super black threshold.

**Prototype:**

**virtual int GetRendFileNumberBase()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the File Number Base in the 'Common Parameters' rollup of the Render Scene dialog.

**Prototype:**

**virtual void SetRendFileNumberBase(int n)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the File Number Base in the 'Common Parameters' rollup of the Render Scene dialog.

**Parameters:**

**int n**

The number to set.

**Prototype:**

**virtual BOOL GetSkipRenderedFrames()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the skip existing rendered frames state is on; otherwise FALSE.

**Prototype:**

**virtual void SetSkipRenderedFrames(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the skip existing rendered frames state to on or off.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

## Right Click Menu Related Methods

### Prototype:

```
virtual RightClickMenuManager*  
GetRightClickMenuManager()=0;
```

### Remarks:

Using this pointer a developer can add to the menu that pops up when the user clicks the right mouse button. Example code is available in:

`\MAXSDK\SAMPLES\MODIFIERS\EDITSP.L.CPP`. See Also: [Class RightClickMenuManager](#).

### Return Value:

A pointer to the **RightClickMenuManager**.

## Rollup Page Related Methods

See [Command Panel -- Rollup Page Methods](#).

## Selection Sets (Named)

See Also: [Sub-Object Selection Sets \(Named\)](#)

### Prototype:

```
virtual void ClearCurNamedSelSet()=0;
```

### Remarks:

This method clears the current edit field of the named selection set drop down.

### Prototype:

```
virtual void SetCurNamedSelSet(TCHAR *setName)=0;
```

### Remarks:

This method is available in release 2.0 and later only.

Sets the edit field of the named selection set drop down to the set whose name is passed.

### Parameters:

**TCHAR \*setName**

The name of the selection set to make current.

**Prototype:**

**virtual void NamedSelSetListChanged()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Calling this method tells the system that the named sub-object selection sets have changed and that the drop down needs to be rebuilt. This will cause **BaseObject::SetupNamedSelDropDown()** to be called on the current item being edited. This is often called inside restore objects that undo changes to the selection set. This causes the system to check if the current item being edited is in sub-object selection mode, and if so, will cause **SetupNamedSelDropDown()** to be called. Note that restore objects can be invoked at any time and the user may not be in sub-object selection mode (for instance they might be in the Display panel). Restore objects however can simply call this method and the system will figure out if the drop down needs to be updated.

**Prototype:**

**virtual int GetNumNamedSelSets()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the number of named selection sets at the object level.

**Prototype:**

**virtual TCHAR \*GetNamedSelSetName(int setNum)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the name of the named selection set whose index is passed.

**Parameters:**

**int setNum**

The index of the named selection set.

**Prototype:**

**virtual int GetNamedSelSetItemCount(int setNum)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the number of items in the named selection set whose index is passed.

**Parameters:**

**int setNum**

The index of the named selection set.

**Prototype:**

**virtual INode \*GetNamedSelSetItem(int setNum,int i)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method may be used to retrieve the INode pointer of the 'i-th' item in the named selection set whose index is passed.

**Parameters:**

**int setNum**

The index of the selection set whose 'i-th' INode pointer is returned.

**int i**

The index into the selection set.

**Prototype:**

**virtual void AddNewNamedSelSet(INodeTab &nodes, TSTR &name)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Adds a new named selection set to those already available in the drop down list in the MAX toolbar.

**Parameters:**

**INodeTab &nodes**

The table of nodes making up the selection set.

**TSTR &name**

The name for the set.

**Prototype:**

**virtual void RemoveNamedSelSet(TSTR &name)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Removes the specified named selection set those already available in the drop down list in the MAX toolbar.

**Parameters:**

**TSTR &name**

The name for the set to remove.

## Scene Access

**Prototype:**

**virtual ReferenceTarget \*GetScenePointer()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer for direct access to the scene. This is primarily used for hanging AppData off the entire scene as opposed to a certain Animatable.

## Show End Result Related Methods

**Prototype:**

**virtual void EnableShowEndResult(BOOL enabled)=0;**

**Remarks:**

This method is called if a modifier wants to temporarily disable any modifiers following it. For example the edit mesh modifier does not let you edit a mesh while other modifiers later in the pipeline are affecting the result so it calls this method to disable the others temporarily.

**Parameters:**

**BOOL enabled**

TRUE is enabled; FALSE is disabled.

**Prototype:**

**virtual BOOL GetShowEndResult()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the Show End Result button is on (pressed); otherwise FALSE.

**Prototype:**

**virtual void SetShowEndResult(BOOL show)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the on/off (pressed/unpressed) state of the Show End Result button. Note that calling this method generates a redraw.

**Parameters:**

**BOOL show**

TRUE for on; FALSE for off.

## Slave / Server Mode Methods

**Prototype:**

**virtual BOOL InSlaveMode()=0;**

**Remarks:**

Returns TRUE if MAX is operating in network rendering mode and FALSE if operating in normal interactive mode. This method returns the same value as **Interface::IsNetServer()**.

**Prototype:**

**virtual BOOL IsNetServer()=0;**

**Remarks:**

Returns TRUE if MAX is operating in network rendering mode and FALSE if operating in normal interactive mode. This method returns the same value as **Interface::InSlaveMode()**.

**Prototype:**

**virtual void SetNetServer()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method is for internal use only. Calling this method will not (alone) set MAX in "Server" mode. Developers should not call this method.

## **Snap Related Methods**

**Prototype:**

**virtual int InitSnapInfo(SnapInfo \*info)=0;**

**Remarks:**

This method is available in release 2.0 and later only (previously available as a global function).

Initialized the SnapInfo structure passed with the current snap settings.

**Parameters:**

**SnapInfo \*info**

Points to the SnapInfo structure to initialize. See [Structure SnapInfo](#).

**Return Value:**

Returns nonzero if snap is on; zero if off.

**Prototype:**

**virtual float SnapAngle(float angleIn, BOOL fastSnap=TRUE, BOOL forceSnap=FALSE)=0;**

**Remarks:**

Normally, with angle snap off, interactive rotation of a node uses a rate of 1/2 degree per pixel. With angle snap on, the angles are snapped to the nearest MAX angle snap value.

This method may be used when interactive rotation is taking place with the mouse to snap the angle passed to the nearest MAX angle snap value. In this method, the input value/output value snap correspondence is accelerated as the angle grows. This prevents the user from having to move the mouse too much with larger angle values. This is why this method does not return a linear relationship between the input angle and the snapped output angle.

Note that when angle snap is off, this method just returns the input value

unaltered.

Note for R3: This method formerly was set up with a single parameter, the angle to be snapped. If the angle snap toggle was on, snapping occurred. If not, it did nothing. There were two problems with this:

1) A multiplier was applied to the angle input, giving faster interactive results in object rotations, etc. This had a negative effect if you just wanted your input angle snapped to the nearest snap angle.

2) It only snapped if the angle snap toggle was on. Sometimes, you might want to snap something to the angle snap value even if the master toggle was off.

To remedy these shortcomings, the method has had two parameters added (**fastSnap** and **forceSnap**).

**Parameters:**

**float angleIn**

Angle to snap in radians.

**BOOL fastSnap=TRUE**

This parameter is available in release 3.0 and later only.

If TRUE the snapping multiplier is used; if FALSE it is not.

**BOOL forceSnap=FALSE**

This parameter is available in release 3.0 and later only.

If TRUE snapping is used even if the master angle snap toggle is off.

**Return Value:**

Angle snapped to the nearest angle snap setting (considering acceleration if specified), in radians, to be used for interactive rotation.

**Prototype:**

**virtual float SnapPercent(float percentIn)=0;**

**Remarks:**

Given a value in the range 0.0 (0%) to 1.0 (100%) this method snaps the value using the current percentage snap.

**Parameters:**

**float percentIn**

The value to snap.

**Return Value:**

The snapped value where 0.0 = 0% and 1.0 = 100%.

**Prototype:**

```
virtual BOOL GetSnapState()=0;
```

**Remarks:**

Retrieves the snap toggle state.

**Return Value:**

TRUE if snap is on; FALSE if snap is off.

**Prototype:**

```
virtual int GetSnapMode()=0;
```

**Remarks:**

Retrieves the current snap type.

**Return Value:**

One of the following values:

```
SNAPMODE_RELATIVE  
SNAPMODE_ABSOLUTE
```

**Prototype:**

```
virtual BOOL SetSnapMode(int mode)=0;
```

**Remarks:**

Set the current snap mode. Note that setting the mode to **SNAPMODE\_ABSOLUTE** will fail if the reference coordinate system is not in set to Screen.

**Parameters:**

**int mode**

One of the following values:

```
SNAPMODE_RELATIVE  
SNAPMODE_ABSOLUTE
```

**Return Value:**

Returns TRUE if succeeded; otherwise FALSE.

## Sound Object Access

### Prototype:

```
virtual SoundObj *GetSoundObject()=0;
```

### Remarks:

Returns the current sound object. See the sample code in `\MAXSDK\SAMPLES\UTILITIES\UTILTEST.CPP`.

See Also: [Class SoundObj](#), [Class IWaveSound](#).

### Prototype:

```
virtual void SetSoundObject(SoundObj *snd)=0;
```

### Remarks:

Sets the current sound object to the one specified. See [Class SoundObj](#).

### Parameters:

```
SoundObj *snd
```

The sound object to set as current.

## Sub-Object Related Methods

### Prototype:

```
virtual void RegisterSubObjectTypes(const TCHAR **types, int  
count, int startIndex=0)=0;
```

### Remarks:

This method registers the sub-object types for a given plug-in object type. See the Advanced Topics section on [sub-object selection](#) for more details.

### Parameters:

```
const TCHAR **types
```

Array of strings listing the sub object levels. The order the strings appear in the array sets the indices used for the sub-object levels. Level 0 is always object level, the first string corresponds to level 1, and the second string corresponds to level 2, etc. In the sample code below, "**Center**" is level 1 and "**Gizmo**" is level 2.

```
int count
```

The number of strings in the array.

**int startIndex=0**

This parameter is available in release 2.0 and later only.

Specifies which string to display initially in the sub-object type combo box. This is needed because the NURBS object computes its sub-object list dynamically, and sometimes it add a new sub-object level while already in a sub-object level. The default value of zero replicates the original behavior.

**Sample Code:**

```
const TCHAR *ptype[] = { _T("Center"), _T("Gizmo") };  
ip->RegisterSubObjectTypes(ptype, 2);
```

**Prototype:**

```
virtual int GetSubObjectLevel()=0;
```

**Remarks:**

Returns the state of the sub object drop-down.

**Return Value:**

0 is object level and  $\geq 1$  refers to the levels registered by the object using **RegisterSubObjectTypes()**. The value refers to the order the item appeared in the list. 1 is the first item, 2 is the second, etc.

**Prototype:**

```
virtual void SetSubObjectLevel(int level, BOOL force =  
FALSE)=0;
```

**Remarks:**

Sets the sub-object drop down. This will cause the object being edited to receive a notification that the current sub-object level has changed (via **BaseObject::ActivateSubobjSel()**).

**Parameters:**

**int level**

The level registered by the object using **RegisterSubObjectTypes()**. 0 indicates object level. Values greater than 1 refer to the order the items appeared in the list.

**BOOL force = FALSE**

If this parameter is TRUE, this method will set the level even if the current level is the same as the level requested. This is to support objects that change sub-object levels on the fly, for instance NURBS.

**Prototype:**

**virtual int GetNumSubObjectLevels()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns the number of sub object levels that the currently edited object (or modifier) has. Like the other sub-object related methods in this class, this method will only return a valid answer if the modifier panel is displayed.

**Prototype:**

**virtual void EnableSubObjectSelection(BOOL enable)=0;**

**Remarks:**

This method enables or disables sub-object selection. Note that it will already be disabled if there are no subobject levels registered. In this case, this method has no effect.

**Parameters:**

**BOOL enable**

TRUE to enable sub-object selection; FALSE to disable.

**Prototype:**

**virtual BOOL IsSubObjectSelectionEnabled()=0;**

**Remarks:**

This method returns the state that is modified by

**EnableSubObjectSelection()**. That is, it does not actually return the state of the button, but indicates the disabled state as set by **EnableSubObjectSelection()**.

**Return Value:**

TRUE if the sub-object button has been disabled by **EnableSubObjectSelection()**; otherwise FALSE.

**Prototype:**

```
virtual void PipeSelLevelChanged()=0;
```

**Remarks:**

Plug-ins call this method to notify the system that the selection level in the pipeline has changed. The selection level flows up the pipeline so if you change the selection level you affect things later in the pipeline. For example an edit modifier that changes the sub-object level from vertex to object level must call this method after making the change to notify the system. Note that it is only called for modifiers whose sub-object levels propagate up the pipeline. For example, when entering a sub-object level within the Bend modifier, **PipeSelLevelChanged()** is not called.

This method should be called from within **ActivateSubobjSel()** to notify the system that a selection level has changed in the pipeline. Note that calling this method from within **ModifyObject()** is no good since it involves re-evaluating the pipeline, which will call **ModifyObject()**, which will call **PipeSelLevelChanged()** again, etc.

**Prototype:**

```
virtual void GetPipelineSubObjLevel(DWORDTab &levels)=0;
```

**Remarks:**

Gets the sub-object selection level at the point in the pipeline just before the current place in the history.

**Parameters:**

**DWORDTab &levels**

The sub-object level. This value depends on the object. The only level defined is level 0, which means 'object' level selection.

**Prototype:**

```
virtual int SubObHitTest(TimeValue t, int type, int crossing, int flags, IPoint2 *p, ViewExp *vpt)=0;
```

**Remarks:**

This method may be called to perform a sub-object hit test. You may access the number of hits using: **vpt->NumSubObjHits()**; See [Class ViewExp](#) for

a list of the other methods that may be used to examine the results.

**Parameters:**

**TimeValue t**

The time of the hit testing.

**int type**

The hit test type. See [List of Hit Test Types](#).

**int crossing**

Nonzero for crossing selection; 0 for normal (window).

**int flags**

The flags for hit testing. See [List of Hit Test Flags](#).

**IPoint2 \*p**

Point to check in screen coordinates.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**Return Value:**

Nonzero if the item was hit; otherwise 0.

## Sub-Object Selection Sets (Named)

**Prototype:**

```
virtual void AppendSubObjectNamedSelSet(const TCHAR  
*set)=0;
```

**Remarks:**

A modifier may call this method to add sub-object named selection sets to the named selection set drop down list in the MAX toolbar. This should be done whenever the selection level changes (in the Modifiers

**BaseObject::ActivateSubobjSel()** method). See [Class BaseObject](#) for additional methods associated with sub-object named selection sets.

**Parameters:**

**const TCHAR \*set**

The named selection set to add to the list.

**Prototype:**

**virtual void ClearSubObjectNamedSelSets()=0;**

**Remarks:**

This method clears the named sub-object selection sets from the drop down.

**Status Panel / Prompt Related Methods**

Generally prompts are set by the command mode and these are the methods used. The developer may use the **DisplayTempPrompt()** method below to temporarily display a prompt to the user independent of a command mode.

**Prototype:**

**virtual void PushPrompt( TCHAR \*s )=0;**

**Remarks:**

Pushes a prompt to display on the prompt stack.

**Parameters:**

**TCHAR \*s**

The string to display.

**Prototype:**

**virtual void PopPrompt()=0;**

**Remarks:**

Pops a displayed string off the prompt stack. The previous prompt will be restored.

**Prototype:**

**virtual void ReplacePrompt( TCHAR \*s )=0;**

**Remarks:**

Replaces the string on the top of the prompt stack.

**Parameters:**

**TCHAR \*s**

The string to display.

**Prototype:**

**virtual void DisplayTempPrompt( TCHAR \*s, int msec=1000)=0;**

**Remarks:**

Displays the string passed for the duration passed. After the time elapses, the string is popped from the stack. This may be used to put up a temporary error message for example.

**Parameters:**

**TCHAR \*s**

The string to display temporarily.

**int msec=1000**

The duration in milliseconds to display the string.

**Prototype:**

**virtual void RemoveTempPrompt()=0;**

**Remarks:**

Removes the temporary prompt immediately.

**Prototype:**

**virtual void DisableStatusXYZ()=0;**

**Remarks:**

Disables mouse tracking and display of coordinates to the X, Y, Z status boxes. Typically a plug-in would disable mouse tracking on mouse down and enable it on mouse up.

**Prototype:**

**virtual void EnableStatusXYZ()=0;**

**Remarks:**

Enables mouse tracking and display of coordinates to the X, Y, Z status boxes. Typically a plug-in would disable mouse tracking on mouse down and enable it on mouse up.

**Prototype:**

**virtual void SetStatusXYZ(Point3 xyz,int type)=0;**

**Remarks:**

Displays the point passed using the format passed in the X, Y, Z status boxes.

**Parameters:**

**Point3 xyz**

The point to be displayed.

**int type**

The format of the point:

**STATUS\_UNIVERSE**

Current system units.

**STATUS\_SCALE**

0=0%, 1=100%.

**STATUS\_ANGLE**

Degrees.

**STATUS\_OTHER**

Straight floating point value.

**Prototype:**

**virtual void SetStatusXYZ(AngAxis aa)=0;**

**Remarks:**

This method will convert the specified angle axis for status display.

**Parameters:**

**AngAxis aa**

The angle axis to convert and display.

## **Track Bar and Track View Related Methods**

**Prototype:**

**virtual ITrackViewNode \*GetTrackViewRootNode()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns a pointer to the Track View Root Node. See [Class ITrackViewNode](#).

**Prototype:**

```
virtual ITreeView* CreateTreeViewChild(ReferenceTarget* root,  
HWND hParent, DWORD style=0, ULONG id=0, int  
open=OPENTV_SPECIAL)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method will create a plain treeview window (no title, borders, etc.) as a child window of the given window. To destroy the window, delete the ITreeView pointer.

**Parameters:****ReferenceTarget\* root**

Points to the root node of the hierarchy to display in the Track View.

**HWND hParent**

The window handle of the parent for the dialog.

**DWORD style=0**

The style flags;

**TVSTYLE\_MAXIMIZEBUT**

Provide a maximize button.

**TVSTYLE\_INVIEWPORT**

Display in the viewport.

**TVSTYLE\_NAMEABLE**

The treeview is namable.

**TVSTYLE\_INMOTIONPAN**

Used in the motion panel.

**ULONG id=0**

The ID of the treeview window.

**int open=OPENTV\_SPECIAL**

One of the following values:

**OPENTV\_NEW**

Open a new treeview.

**OPENTV\_SPECIAL**

Open a special treeview.

**OPENTV\_LAST**

Open the last treeview.

**Prototype:**

**virtual ITrackBar\* GetTrackBar()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns an instance of the **ITrackBar** class. This class may be used to manipulate the track bar. See [Class ITrackBar](#).

## **Time Configuration Key Steps Settings Access**

**Prototype:**

**virtual BOOL GetKeyStepsSelOnly()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the Time Configuration / Key Steps / Selected Objects Only check box is on; otherwise FALSE.

**Prototype:**

**virtual void SetKeyStepsSelOnly(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the Time Configuration / Key Steps / Selected Objects Only check box to on or off.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetKeyStepsUseTrans()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the Time Configuration / Key Steps / Use Current Transform check box is on; otherwise FALSE.

**Prototype:**

**virtual void SetKeyStepsUseTrans(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the Time Configuration / Key Steps / Use Current Transform check box to on or off.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetKeyStepsPos()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the Time Configuration / Key Steps / Position check box is on; otherwise FALSE. This value is only meaningful if Use Current Transform is off.

**Prototype:**

**virtual void SetKeyStepsPos(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the Time Configuration / Key Steps / Position check box is to on or off.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetKeyStepsRot()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the Time Configuration / Key Steps / Rotation check box is on; otherwise FALSE. This value is only meaningful if Use Current Transform is off.

**Prototype:**

**virtual void SetKeyStepsRot(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the Time Configuration / Key Steps / Rotation check box is to on or off.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetKeyStepsScale()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the Time Configuration / Key Steps / Scale check box is on; otherwise FALSE. This value is only meaningful if Use Current Transform is off.

**Prototype:**

**virtual void SetKeyStepsScale(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the Time Configuration / Key Steps / Scale check box is to on or off.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:****virtual BOOL GetKeyStepsUseTrackBar()=0;****Remarks:**

This method is available in release 3.0 and later only.

Returns the state of the Time Configuration dialog 'Key Steps / Use TrackBar' checkbox. TRUE if checked; FALSE if unchecked.

**Prototype:****virtual void SetKeyStepsUseTrackBar(BOOL onOff)=0;****Remarks:**

This method is available in release 3.0 and later only.

Sets the state of the Time Configuration dialog 'Key Steps / Use TrackBar' checkbox.

**Parameters:****BOOL onOff**

TRUE for on; FALSE for off.

**User Interface Controls and Properties****Prototype:****virtual void SetFlyOffTime(int msec)=0;****Remarks:**

This sets the custom control flyoff time to the value passed. This is the number of milliseconds the user must hold down on a flyoff button before the flyoff is activated.

**Parameters:****int msec**

The number of milliseconds the user must hold down on the button before the flyoff is activated.

**Prototype:****virtual int GetFlyOffTime()=0;****Remarks:**

Returns the number of milliseconds the user must hold down on a flyoff button before the flyoff is activated.

**Prototype:**

**virtual BOOL GetCrossing()=0;**

**Remarks:**

Returns the state of the 'crossing' preference for hit testing.

**Return Value:**

TRUE if crossing selection is on; FALSE if off.

**Prototype:**

**virtual void SetToolButtonState(int button, BOOL state )=0;**

**Remarks:**

Sets the state of one of the transform tool buttons.

**Parameters:**

**int button**

The transform tool buttons:

**MOVE\_BUTTON**

**ROTATE\_BUTTON**

**NUSCALE\_BUTTON**

**USCALE\_BUTTON**

**SQUASH\_BUTTON**

**SELECT\_BUTTON**

**BOOL state**

TRUE indicates pressed, FALSE is not pressed.

**Prototype:**

**virtual int GetAxisConstraints()=0;**

**Remarks:**

Retrieves the state of the axis constraints flyoff.

**Return Value:**

One of the following axis constraints:

**AXIS\_XY**  
**AXIS\_ZX**  
**AXIS\_YZ**  
**AXIS\_X**  
**AXIS\_Y**  
**AXIS\_Z**

**Prototype:**

**virtual void SetAxisConstraints(int c)=0;**

**Remarks:**

Sets the state of the axis constraints flyoff.

**Parameters:**

**int c**

The axis constraint to set. You may pass one of the following:

**AXIS\_XY**  
**AXIS\_ZX**  
**AXIS\_YZ**  
**AXIS\_X**  
**AXIS\_Y**  
**AXIS\_Z**

**Prototype:**

**virtual void EnableAxisConstraints(int c, BOOL enabled)=0;**

**Remarks:**

Enables or disables the specified axis constraint.

**Parameters:**

**int c**

The axis constraint. You may pass one of the following:

**AXIS\_XY**  
**AXIS\_ZX**  
**AXIS\_YZ**  
**AXIS\_X**

**AXIS\_Y**  
**AXIS\_Z**

**BOOL enabled**

TRUE to enable; FALSE to disable.

**Prototype:**

**virtual void PushAxisConstraints(int c)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Pushes the specified axis constraint. This push/pop mechanism is used so that the appropriate axis mode can be restored after the Transform Gizmo has been used. The Gizmo itself calls this Push method in response to the HitTest with certain flags.

**Parameters:**

**int c**

The axis constraint. You may pass one of the following:

**AXIS\_XY**  
**AXIS\_ZX**  
**AXIS\_YZ**  
**AXIS\_X**  
**AXIS\_Y**  
**AXIS\_Z**

**Prototype:**

**virtual void PopAxisConstraints()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Pops the active constraint. After the Transform Gizmo pushes a constraint a selection processor pops it back again after the manipulators are deactivated.

**Prototype:**

**virtual int GetCoordCenter()=0;**

**Remarks:**

Retrieves the state of the coordinate system center.

**Return Value:**

One of the following values:

**ORIGIN\_LOCAL**

Object's pivot.

**ORIGIN\_SELECTION**

Center of selection set (or center of individual object for local or parent space).

**ORIGIN\_SYSTEM**

Center of the reference coordinate system.

**Prototype:**

**virtual void SetCoordCenter(int c)=0;**

**Remarks:**

Sets the state of the coordinate system center.

**Parameters:**

**int c**

One of the following values (from **MAXAPI.H**).

**ORIGIN\_LOCAL**

Object's pivot.

**ORIGIN\_SELECTION**

Center of selection set (or center of individual object for local or parent space).

**ORIGIN\_SYSTEM**

Center of the reference coordinate system.

**Prototype:**

**virtual void EnableCoordCenter(BOOL enabled)=0;**

**Remarks:**

Enables or disables the coordinates system center.

**Parameters:**

## **BOOL enabled**

TRUE to enable; FALSE to disable.

### **Prototype:**

```
virtual int GetRefCoordSys()=0;
```

### **Remarks:**

Retrieves the reference coordinate system setting.

### **Return Value:**

One of the following reference coordinate systems:

```
COORDS_HYBRID  
COORDS_SCREEN  
COORDS_WORLD  
COORDS_PARENT  
COORDS_LOCAL  
COORDS_OBJECT
```

### **Prototype:**

```
virtual void SetRefCoordSys(int c)=0;
```

### **Remarks:**

Sets the reference coordinate system used.

### **Parameters:**

**int c**

Reference coordinate system:

```
COORDS_HYBRID  
COORDS_SCREEN  
COORDS_WORLD  
COORDS_PARENT  
COORDS_LOCAL  
COORDS_OBJECT
```

### **Prototype:**

```
virtual void EnableRefCoordSys(BOOL enabled)=0;
```

**Remarks:**

Enables or disables the reference coordinates system.

**Parameters:****BOOL enabled**

TRUE to enable; FALSE to disable.

**Prototype:**

```
virtual BOOL GetPlaybackLoop()=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the state of the "loop" checkbox in the time configuration panel. Note that the loop control is only active when "real time" is selected.

**Return Value:**

TRUE if loop is on; FALSE if off.

**Prototype:**

```
virtual void SetPlaybackLoop(BOOL loop)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the state of the "loop" checkbox in the time configuration panel. Note that the loop control is only active when "real time" is selected.

**Parameters:****BOOL loop**

TRUE to set the loop to on; FALSE to set it off.

**Video Post Related Methods****Prototype:**

```
virtual INode *GetINodeFromRenderID(UWORD id)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the node pointer from the id in the

**BMM\_CHAN\_NODE\_RENDER\_ID** G-Buffer channel. The renderer will set the RenderID of all rendered nodes, and will set all non-rendered nodes to **0xffff**. See [List of Image Channels](#).

**Parameters:**

**UWORD id**

The id from the G Buffer channel.

**Prototype:**

**virtual int GetSelectFilter()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns your current selected select filter in the toolbar.

**Prototype:**

**virtual void SetSelectFilter(int c)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the current selected select filter in the toolbar.

**Parameters:**

**int c**

The index of the filter you wish to set.

**Prototype:**

**virtual int GetNumberSelectFilters()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the number of select filters in the drop down list.

**Prototype:**

**virtual TCHAR\* GetSelectFilterName(int index)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the name that appears in the interface for the specified filter.

**Parameters:**

**int index**

The index of the filter.

**Prototype:**

**virtual BOOL GetDisplayFilter(int index) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the state of a display filter.

**Parameters:**

**int index**

The index of the display filter that you want to check.

**Prototype:**

**virtual void SetDisplayFilter(int index, BOOL on) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the state of a display filter.

**Parameters:**

**int index**

The index of the display filter you wish to set.

**BOOL on**

TRUE for on; FALSE for off.

**Prototype:**

**virtual int GetNumberDisplayFilters() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the number of display filters in the display panel.

**Prototype:**

```
virtual TCHAR* GetDisplayFilterName(int index)=0;;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the name of the specified filter.

**Parameters:**

**int index**

The index of the filter.

**Prototype:**

```
virtual BOOL DisplayFilterIsNodeVisible(int index, int sid,  
Class_ID cid, INode *node) = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method checks the display filter at index, and sees if the node, class id, and super class id fail the filter check or not.

**Parameters:**

**int index**

The index of the filter

**int sid**

The super class id

**Class\_ID cid**

The class ID

**INode \*node**

The node to check.

**Return Value:**

TRUE if visible, otherwise FALSE.

**Viewport Access****Prototype:**

```
virtual void RedrawViews(TimeValue t, DWORD
```

```
vpFlags=REDRAW_NORMAL, ReferenceTarget
*change=NULL)=0;
```

**Remarks:**

This method may be called to cause the viewports to be redrawn.

**Parameters:**

**TimeValue t**

The time at which to redraw the viewports.

**DWORD vpFlags=REDRAW\_NORMAL**

You may specify one of the following:

**REDRAW\_BEGIN**

Call this before you redraw.

**REDRAW\_INTERACTIVE**

This allows the view quality to degrade to maintain interactivity.

**REDRAW\_END**

If during interactive redraw the state degraded, this will redraw the views in the undegraded state.

**REDRAW\_NORMAL**

This redraws the views in the undegraded state.

**ReferenceTarget \*change=NULL**

This parameter is not used - always let it default to NULL.

**Example:**

```
ip->RedrawViews(ip->GetTime(),REDRAW_BEGIN);
// More code ...
ip->RedrawViews(ip->GetTime(),REDRAW_INTERACTIVE);
// More code ...
ip->RedrawViews(ip->GetTime(),REDRAW_END);
```

**Prototype:**

```
virtual void ForceCompleteRedraw(BOOL doDisabled=TRUE)=0;
```

**Remarks:**

Calling this method will cause all the viewports to be completely redrawn.

Note: This method literally forces **everything** (every object, every screen rectangle, every view) to be marked invalid and then the whole scene is

regenerated. (The individual object pipeline caches are **not** flushed, however.)  
So this routine is guaranteed to be slow.

**Parameters:**

**BOOL doDisabled=TRUE**

This parameter is available in release 2.0 and later only.

If TRUE disabled viewports are redrawn; otherwise they are not.

**Prototype:**

**virtual void DisableSceneRedraw()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This turns the scene redraw off (disables it). Scene redraw should be disabled in any renderer's **Open()** method, and re-enabled in the renderer's **Close()** method. All calls to **DisableSceneRedraw()/Enable SceneRedraw()** should be in pairs, since an internal counter is used to actually do the redraw enable/disable action.

**Prototype:**

**virtual void EnableSceneRedraw()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This turns the scene redraw on (enables it). Scene redraw should be disabled in any renderer's **Open()** method, and re-enabled in the renderer's **Close()** method. All calls to **DisableSceneRedraw()/Enable SceneRedraw()** should be in pairs, since an internal counter is used to actually do the redraw enable/disable action.

**Prototype:**

**virtual int IsSceneRedrawDisabled()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns nonzero if the redraw is disabled; zero if enabled.

**Prototype:**

**virtual BOOL SetActiveViewport( HWND hwnd )=0;**

**Remarks:**

This allows you to specify the active viewport.

**Parameters:**

**HWND hwnd**

The handle of the window to activate.

**Return Value:**

TRUE if the viewport was not previously active; otherwise FALSE.

**Prototype:**

**virtual ViewExp \*GetActiveViewport()=0;**

**Remarks:**

Returns the **ViewExp** pointer of the active MAX viewport. Remember to release the **ViewExp** pointer with **Interface::ReleaseViewport()**.

**Prototype:**

**virtual ViewExp \*GetViewport( HWND hwnd )=0;**

**Remarks:**

This method gets a viewport interface given a window handle.

**Parameters:**

**HWND hwnd**

The window handle of the viewport.

**Prototype:**

**virtual void ReleaseViewport( ViewExp \*vpt )=0;**

**Remarks:**

When the developer is done with the viewport interface acquired via **GetViewport()** or **GetActiveViewport()** they should call this method to release it.

**Parameters:**

## **ViewExp \*vpt**

The viewport interface to release.

### **Prototype:**

**virtual int GetViewportLayout()=0;**

### **Remarks:**

This method may be called to retrieve a value that describes the configuration of the MAX viewports.

### **Return Value:**

The viewport layout configuration. The list below uses the following syntax:

# is the total number of viewports.

V = vertical split

H = horizontal split

L/R = left/right placement

T/B = top/bottom placement

One of the following values. Note: The bottom nibble (4-bits) is the total number of views. You may use the constant **VP\_NUM\_VIEWS\_MASK** to mask off the 4 bits that contains the total number of viewports.

**VP\_LAYOUT\_1**

**VP\_LAYOUT\_2V**

**VP\_LAYOUT\_2H**

**VP\_LAYOUT\_2HT**

**VP\_LAYOUT\_2HB**

**VP\_LAYOUT\_3VL**

**VP\_LAYOUT\_3VR**

**VP\_LAYOUT\_3HT**

**VP\_LAYOUT\_3HB**

**VP\_LAYOUT\_4**

**VP\_LAYOUT\_4VL**

**VP\_LAYOUT\_4VR**

**VP\_LAYOUT\_4HT**

**VP\_LAYOUT\_4HB**

**VP\_LAYOUT\_1C**

**Prototype:**

**virtual void SetViewportLayout(int layout)=0;**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the viewport configuration layout.

**Parameters:**

**int layout**

The layout to use. See the return values of **GetViewportLayout()** above.

**Prototype:**

**virtual BOOL IsViewportMaxed()=0;**

**Remarks:**

Returns TRUE if the current viewport is full screen; otherwise FALSE.

**Prototype:**

**virtual void SetViewportMax(BOOL max)=0;**

**Remarks:**

This method will maximize (set to a single full screen view) or minimize the current viewport.

**Parameters:**

**BOOL max**

If TRUE the viewport is maximized; otherwise it is minimized.

**Prototype:**

**virtual void ViewportZoomExtents(BOOL doAll, BOOL skipPersp=FALSE)=0;**

**Remarks:**

This method performs a zoom extents on the viewport(s). This fills the viewport(s) with the objects of the scene.

**Parameters:**

**BOOL doAll**

If TRUE all the viewports are zoomed to their extents; otherwise just the

current viewport is.

**BOOL skipPersp=FALSE**

If TRUE perspective viewports are not altered; otherwise these views are zoomed to their extents as well.

**Prototype:**

**virtual int IsCPEdgeOnInView()=0;**

**Remarks:**

This method returns nonzero if the construction plane is 'head on' in the current viewport. For example if the construction plane was XY and you were looking from the Front view, this method would return nonzero. This is used for example during object creation because this process doesn't work very well when the view is 'head on'.

**Return Value:**

Nonzero if the construction plane is 'head on' in the current viewport; otherwise 0.

For use with extended views: - make the extended viewport active (set on mouse click, for example) -put up the view type popup menu (put up on right-click, for example)

**Prototype:**

**virtual void MakeExtendedViewportActive(HWND hWnd)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used with Extended Viewports (see [Class ViewWindow](#)). It is called when the extended viewport needs to become active. It should be called whenever the user clicks in the non-3D window (so as to deactivate the current 3D window, and redirect commands like the Min/Max toggle to the non-3D viewport window).

**Parameters:**

**HWND hWnd**

The handle of the window which to made active.

**Prototype:**

```
virtual void PutUpViewMenu(HWND hWnd, POINT pt)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is used with Extended Viewports (see [Class ViewWindow](#)). It is called to put up the view type popup menu (for example the right-click menu). It should be called when the user right-clicks in a dead region of the non-3D window. This brings up the view selection menu so that the user can choose to replace the current window with a 3D or other non-3D window without having to go to the Views | Viewport Config dialog directly.

**Parameters:**

**HWND hWnd**

The handle of the window the menu is to appear in.

**POINT pt**

The point at which the menu is put up.

**Sample Code:**

```
case WM_RBUTTONDOWN:
```

```
    pt.x = LOWORD(l);
```

```
    pt.y = HIWORD(l);
```

```
        GetCOREInterface()->PutUpViewMenu(h, pt);
```

**Prototype:**

```
virtual BOOL RegisterViewWindow(ViewWindow *vw)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to register a window that can appear in a viewport.

**Parameters:**

**ViewWindow \*vw**

The pointer to the view window to register.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual BOOL UnRegisterViewWindow(ViewWindow \*vw)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to unregister a window that can appear in a viewport.

**Parameters:**

**ViewWindow \*vw**

The pointer to the view window to unregister.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual void ZoomToBounds(BOOL doAll, Box3 box)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to zoom the current or selected viewport to a bounding region.

**Parameters:**

**BOOL doAll**

This flag determines whether only the selected or all viewports get zoomed.

TRUE for all, FALSE for selected only.

**Box3 box**

The bounding region to zoom to.

## **Viewport Background Properties**

**Prototype:**

**virtual void SetViewportBGColor(const Point3 &color)=0;**

**Remarks:**

Sets the viewport background color to the specified color.

**Parameters:**

**const Point3 &color**

The color to set.

**Prototype:**

**virtual Point3 GetViewportBGColor()=0;**

**Remarks:**

Returns the viewport background color.

**Prototype:**

**virtual BOOL setBkgImageName(TCHAR \*name)=0;**

**Remarks:**

This method is used to specify the background image used.

**Parameters:**

**TCHAR \*name**

The name of the background image.

**Return Value:**

TRUE if the image was set; otherwise FALSE.

**Prototype:**

**virtual TCHAR \*getBkgImageName()=0;**

**Remarks:**

This method is used to retrieve the name of the background image used. The pointer returned from this method does not need to be freed.

**Prototype:**

**virtual void setBkgImageAspect(int t)=0;**

**Remarks:**

Sets the background image aspect ratio. This may match the viewport, the bitmap, or the rendering output aspect ratio.

**Parameters:**

**int t**

One of the following values:

**VIEWPORT\_BKG\_ASPECT\_VIEW**

**VIEWPORT\_BKG\_ASPECT\_BITMAP**  
**VIEWPORT\_BKG\_ASPECT\_OUTPUT**

**Prototype:**

**virtual int getBkgImageAspect()=0;**

**Remarks:**

Retrieves the background image aspect ratio. This will be the viewport, the bitmap, or the rendering output aspect ratio.

**Return Value:**

One of the following values:

**VIEWPORT\_BKG\_ASPECT\_VIEW**  
**VIEWPORT\_BKG\_ASPECT\_BITMAP**  
**VIEWPORT\_BKG\_ASPECT\_OUTPUT**

**Prototype:**

**virtual void setBkgImageAnimate(BOOL onOff)=0;**

**Remarks:**

This method sets if the background image is animated in the viewports. If TRUE the image updates to reflect the current frame. If FALSE the image remains static regardless of time.

**Parameters:**

**BOOL onOff**

TRUE to enable viewport background image animation; FALSE to disable it.

**Prototype:**

**virtual int getBkgImageAnimate()=0;**

**Remarks:**

This method determines if the background image is set to update with the current frame in the viewports.

**Prototype:**

**virtual void setBkgFrameRange(int start, int end, int step=1)=0;**

**Remarks:**

This method establishes the range of frames used for an animated background.

**Parameters:****int start**

The start frame number.

**int end**

The end frame number.

**int step=1**

The frame increment.

**Prototype:**

```
virtual int getBkgFrameRangeVal(int which)=0;
```

**Remarks:**

This method retrieves either the start or end frame number.

**Parameters:****int which**

One of the following values:

**VIEWPORT\_BKG\_START**

**VIEWPORT\_BKG\_END**

**Prototype:**

```
virtual void setBkgORTType(int which, int type)=0;
```

**Remarks:**

Sets the background Out of Range Type. This may be the start or end ORT.

**Parameters:****int which**

One of the following values:

**0** : Sets the Start Processing ORT.

**1** : Sets the End Processing ORT.

**int type**

One of the following values:

**VIEWPORT\_BKG\_BLANK**

**VIEWPORT\_BKG\_HOLD**  
**VIEWPORT\_BKG\_LOOP**

**Prototype:**

**virtual int getBkgORType(int which)=0;**

**Remarks:**

Retrieves the background Out of Range Type. This may be the start or end ORT.

**Parameters:**

**int which**

One of the following values:

**0** : Gets the Start Processing ORT.

**1** : Gets the End Processing ORT.

**Return Value:**

One of the following values:

**VIEWPORT\_BKG\_BLANK**

**VIEWPORT\_BKG\_HOLD**

**VIEWPORT\_BKG\_LOOP**

**Prototype:**

**virtual void setBkgStartTime(TimeValue t)=0;**

**Remarks:**

This sets the "Start at" parameter from the Views / Background Image... dialog.

**Parameters:**

**TimeValue t**

The time to start.

**Prototype:**

**virtual TimeValue getBkgStartTime()=0;**

**Remarks:**

This returns the "Start at" parameter from the Views / Background Image...

dialog.

**Prototype:**

**virtual void setBkgSyncFrame(int f)=0;**

**Remarks:**

Sets the background "Sync Start to Frame" setting.

**Parameters:**

**int f**

The frame number.

**Prototype:**

**virtual int getBkgSyncFrame()=0;**

**Remarks:**

Returns the background "Sync Start to Frame" setting.

**Prototype:**

**virtual int getBkgFrameNum(TimeValue t)=0;**

**Remarks:**

This method will convert the TimeValue passed to a frame number based on the background image settings (ORTs, start/end times, sync frame, etc.).

**Parameters:**

**TimeValue t**

The time to convert.

**Return Value:**

The frame number corresponding to the time passed.

## **Window Handle of MAX**

**Prototype:**

**virtual HWND GetMAXHwnd()=0;**

**Remarks:**

Returns the window handle of MAX.

## Windows Messages

### Prototype:

**virtual void TranslateAndDispatchMAXMessage(MSG &msg)=0;**

### Remarks:

If a plug-in needs to do a **PeekMessage()** and wants to actually remove the message from the queue, it can use this method to have the message translated and dispatched.

### Parameters:

**MSG &msg**

The message from **PeekMessage()**.

### Prototype:

**virtual BOOL CheckMAXMessages()=0;**

### Remarks:

This will go into a **PeekMessage()** loop until there are no more messages left. This is a way a plug-in can relieve control to the system.

There may be certain circumstances where a plug-in wants to give control back to MAX. For example a plug-in may put up a progress bar with a cancel button during a lengthy operation. However the cancel button would not receive any messages if the user was clicking on it because no messages are being dispatched.

This method will relieve control and let any messages that are in the queue get processed. If there are no messages it will return right away. This provides a way for a plug-in to yield control.

Note: A developer must be prepared to handle a lot of different conditions if this is done. For example the user could click on the delete key and delete the object that was being processed. **EndEditParams()** could be called on the plug-in. So in **EndEditParams()** there must be some logic to signal the other lengthy process that **EndEditParams()** was called.

### Return Value:

If this method returns FALSE then the user is attempting to quit MAX and the caller should return.

**Prototype:**

**virtual void RescaleWorldUnits(float f, BOOL selected)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to rescale the world units of the entire scene, or optionally the current selection set.

**Parameters:**

**float f**

The scale factor to apply to the scene.

**BOOL selected**

TRUE to scale selected objects only; otherwise the entire scene is scaled.

## Transform Gizmo Related Methods

**Prototype:**

**virtual BOOL GetUseTransformGizmo()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the state of the Transform Tools / Gizmo toggle.

**Return Value:**

TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetUseTransformGizmo(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method enables or disables the use of Transform Gizmos.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetTransformGizmoRestoreAxis(BOOL bOnOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method sets whether the TransformGizmo should restore the axis constraint when released, or if the axis constraint is permanently changed. The value is saved in the 3DSMAX.INI file for later sessions.

**Parameters:**

**BOOL bOnOff**

Enable or disable the restoration of the axis constraint.

**Prototype:**

**virtual BOOL GetTransformGizmoRestoreAxis()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Indicates if the TransformGizmo will restore the axis constraint when released. Returns TRUE if it will; FALSE if it won't.

**Prototype:**

**virtual BOOL GetConstantAxisRestriction()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the state of the Transform Tools / Constant Axis toggle.

**Return Value:**

TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetConstantAxisRestriction(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the state of the Transform Tools / Constant Axis toggle.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

```
virtual int HitTestTransformGizmo(IPoint2 *p, ViewExp *vpt, int axisFlags) = 0;
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to hittest gizmos for sub-objects.

**Parameters:**

**IPoint2 \*p**

Point to check in screen coordinates.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**int axisFlags**

One or more of the following values:

**HIT\_TRANSFORMGIZMO**

This flag is passed in on a **MOUSE\_FREEMOVE** message so that the axis is hit tested and it highlights if it is hit, but it doesn't actually switch the transform mode.

**HIT\_SWITCH\_GIZMO**

In case of a **MOUSE\_POINT**, this flag is used, and if the axis is hit, the 'hit' transform mode will be pushed on the transform mode stack.

**Return Value:**

Nonzero if the item was hit; otherwise 0.

## **Add / Delete Class Methods**

**Prototype:**

```
virtual int AddClass(ClassDesc *cd)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to dynamically add a plug-in class. This method will

update the control panel in the Create or Modify branches dynamically.

**Parameters:**

**ClassDesc \*cd**

Points to the Class Descriptor to add. See [Class ClassDesc](#).

**Return Value:**

Returns -1 if the superclass was unknown, 0 if the class already exists, or 1 if the class was added successfully.

**Prototype:**

**virtual int DeleteClass(ClassDesc \*cd)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to dynamically delete a plug-in class. This method will update the control panel in the Create or Modify branches dynamically.

**Parameters:**

**ClassDesc \*cd**

Points to the Class Descriptor to add. See [Class ClassDesc](#).

**Return Value:**

Returns -1 if the superclass was unknown, 0 if the class does not exist, or 1 if the class was deleted successfully.

## **Property Set Access**

The following methods provide developer access to the property set data stored by MAX. A MAX user can enter this data via the File Properties dialog. There are three tabs to this dialog which correspond to the options which may be specified for the **PropertySet** parameter used in the methods below. The **PROPSPEC** and **PROPVARIANT** structures used below are part of the Windows API. Developers can find sample code using these methods in **\MAXSDK\SAMPLES\UTILITIES\PROPERTYTEST\PROPERTYT**

**Prototype:**

**virtual int GetNumProperties(int PropertySet)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of properties of the specified property set. See the note at the start of this group of methods above for info on property sets.

**Parameters:**

**int PropertySet**

See [List of PropertySet Options](#).

**Prototype:**

```
virtual int FindProperty(int PropertySet, const PROPSPEC*  
propspec)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Return the index of the specified property or -1 if it is not found. See the note at the start of this group of methods above for info on property sets.

**Parameters:**

**int PropertySet**

See [List of PropertySet Options](#).

**const PROPSPEC\* propspec**

Points to a PROPSPEC structure of the property to find. The Windows API PROPSPEC structure is used by many of the methods of IPropertyStorage to specify a property either by its property identifier or the associated string name. See the Windows API for details on this structure.

**Return Value:**

The zero based index of the specified property or -1 if not found.

**Prototype:**

```
virtual const PROPVARIANT* GetPropertyVariant(int  
PropertySet, int idx)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Return the value of the property at this index, in PROPVARIANT form. See the note at the start of this group of methods above for info on property sets.

**Parameters:****int PropertySet**See [List of PropertySet Options](#).**int idx**

The zero based index of the property variant to get.

**Return Value:**

Points to a **PROPVARIANT** structure. This Windows API structure is used in most of the methods of IPropertyStorage to define the type tag and the value of a property in a property set. See the Windows API for details on this structure.

**Prototype:**

```
virtual const PROPSPEC* GetPropertySpec(int PropertySet, int idx)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Return the name of the property at this index, in PROPSPEC form. See the note at the start of this group of methods above for info on property sets.

**Parameters:****int PropertySet**See [List of PropertySet Options](#).**int idx**

The zero based index of the property name to get.

**Return Value:**

Points to a PROPSPEC structure. The Windows API PROPSPEC structure is used by many of the methods of IPropertyStorage to specify a property either by its property identifier or the associated string name. See the Windows API for details on this structure.

**Prototype:**

```
virtual void AddProperty(int PropertySet, const PROPSPEC* propspec, const PROPVARIANT* propvar)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

This method adds a property to the specified property set. See the sample code in

`\MAXSDK\SAMPLES\UTILITIES\PROPERTYTEST\PROPERTY`

See the note at the start of this group of methods above for info on property sets.

**Parameters:**

**int PropertySet**

See [List of PropertySet Options](#).

**const PROPSPEC\* propspec**

Points to a PROPSPEC structure.

**const PROPVARIANT\* propvar**

Points to a PROPVARIANT structure.

**Prototype:**

**virtual void DeleteProperty(int PropertySet, const PROPSPEC\* propspec)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes the specified property. The property will be removed and the memory freed. See the note at the start of this group of methods above for info on property sets.

**Parameters:**

**int PropertySet**

See [List of PropertySet Options](#).

**const PROPSPEC\* propspec**

Points to a PROPSPEC structure to delete.

## **XRef Methods**

**Prototype:**

**virtual void SetIncludeXRefsInHierarchy(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method allows a plug-in to specify whether scene XRef objects are hidden from the hierarchy when it is traversed. Normally this parameter is set to FALSE except during rendering. If a plug-in wants access to XRef scene objects then it should set this to TRUE and traverse the scene and then set it back to FALSE when it's done.

Most of the time the XRef trees (whose root node is a child of the client scene's root node) are skipped when traversing the hierarchy. When this option is turned on, all root nodes will include child XRef scene root nodes in any traversal related functions such as **NumberOfChildren()** and **GetChildNode(i)**.

This option is turned on automatically before rendering and turned off after so that scene XRefs appear in the production renderer. **Note: This option should not be left on if it is turned on since it would cause scene XRef objects to be accessible to the user in the client scene.**

Note that plug-ins can also access XRef objects using the [Class INode XRef methods](#).

**Parameters:**

**BOOL onOff**

TRUE to include XRefs in the hierarchy; FALSE to not include them.

**Prototype:**

**virtual BOOL GetIncludeXRefsInHierarchy()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if XRefs are included in the traversal of the scene hierarchy; otherwise FALSE. See the method above for details.

**Prototype:**

**virtual BOOL IsXRefAutoUpdateSuspended()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the automatic updating of XRefs is suspended; otherwise

FALSE. When an XRef file is changed and that causes an XRef object to update, the old XRef object gets deleted from memory which can cause problems for some plug-ins. For example, the Dynamics system would have a problem if an update occurred while a solution was solving. This method is used to disable the automatic updating to prevent the problem.

**Prototype:**

**virtual void SetXRefAutoUpdateSuspended(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets if the automatic updating of XRefs is suspended or not. See the note in **IsXRefAutoUpdateSuspended()** for details.

**Parameters:**

**BOOL onOff**

TRUE to suspend; FALSE to restore automatic updating.

**Prototype:**

**virtual BOOL IsSceneXRefNode(INode \*node)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the specified node is part of a scene XRef or FALSE if the node is a regular modifiable node in the current scene.

**Parameters:**

**INode \*node**

The node to check.

## **Licensing Methods**

**Prototype:**

**virtual bool IsTrialLicense()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the application is running under a trial license, as opposed to

a full, authorized license; otherwise FALSE.

**Prototype:**

**virtual bool IsNetworkLicense()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

As of R4 this method will always return false as R4 does not have network licencing.

**Prototype:**

**virtual bool IsEmergencyLicense()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

As of R4 this method is no longer supported and always returns false.

Returns TRUE if the application is running under an emergency license, while the full license is exported; otherwise FALSE.

**Prototype:**

**virtual int GetProductVersion()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns the product version which is one of the following values:

**PRODUCT\_VERSION\_DEVEL** -- A debug build, or licensed in-house.

**PRODUCT\_VERSION\_TRIAL** -- A trial license.

**PRODUCT\_VERSION\_ORDINARY** -- A commercial license.

**PRODUCT\_VERSION\_NFR** -- Not for resale.

**PRODUCT\_VERSION\_EDU** -- Educational or student license.

**Prototype:**

**virtual int GetLicenseBehavior()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns one of the following values which indicates the liscence behaviour:

**LICENSE\_BEHAVIOR\_PERMANENT** -- A permanent license, or hardware lock.

**LICENSE\_BEHAVIOR\_EXTENDABLE** -- A term license which can be extended.

**LICENSE\_BEHAVIOR\_NONEXTENDABLE** -- A term license which cannot be extended.

**Prototype:**

**virtual int GetLicenseDaysLeft()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns an integer indicating the number of full days left in the term of the license. A value of 0 means that today is the last day of validity. For permanent licenses, a fixed value is returned indicating greater than 10 years are left.

**Prototype:**

**virtual bool IsFeatureLicensed(int subNum)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method is not currently supported and always returns false. In the future it will be used for returning true or false as the license subgroup designated by the argument is or is not enabled.

**Return Value:**

TRUE if licensed; FALSE if not licensed.

**Deferred Loading Related Methods****Prototype:**

**virtual void EnableDeferredPluginLoading(bool onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

In the Preferences dialog / General Tab / Plug-In Loading section there is a Checkbox labelled 'Load Plug-Ins when Used'. This method sets the state of this toggle. See the Advanced Topics section [Deferred Loading of Plug-Ins](#).

**Parameters:****bool onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual bool DeferredPluginLoadingEnabled()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

In the Preferences dialog / General Tab / Plug-In Loading section there is a Checkbox labelled 'Load Plug-Ins when Used'. This method returns the state of this toggle.. See the Advanced Topics section [Deferred Loading of Plug-Ins](#).

**Return Value:**

TRUE if on; FALSE if off.

**Undo / Redo Related Methods****Prototype:**

**virtual void FlushUndoBuffer()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This function will flush the undo buffer. See the Advanced Topics section [Undo / Redo](#).

## Class List by Category

See Also: [Plug-In Types Overview](#).

This section provides a list of all the main classes in the SDK organized by category. Hyperlinks are provided to the reference section of each class. You may also use the Search function in the on-line help toolbar. When using search note that all the classes in the SDK are named beginning with **Class**, so they are grouped together and appear in alphabetical order. To find a particular method or function by name, choose Search, select the Find tab, then enter the name in the 'Type the word(s) you want to find' field.

[Overview of the Principal Classes](#)

[Main Plug-In Classes](#)

[Interface Classes](#)

[Geometry / Bitmap Classes](#)

[User Interface Classes](#)

[Miscellaneous Utility Classes](#)

## Sparks Addin for MS VC++ 6 IDE

**See Also:** [Sparks Developer Program](#), [Sparks Developer Knowledgebase](#), [Sparks Addin for MS VC++ 7 IDE](#)

## Overview

New for 3ds max 5.1 is the Sparks Addin for use with the MS VC++ 6 IDE. Its functionality allows 3ds max developers to search the sparks knowledgebase for the text that's selected in the IDE.

**The Sparks Addin is available to the public.**

## Installation

1. Open VC++ 6, and go to Tools: Customize: Add-ins and Macro Files.
2. Click on the 'browse' button and locate SparksAddin.dll in the maxsdk help directory. Click 'open'.
3. Check the checkbox next to its entry in the 'Add-ins and macro files:' list box.
4. The toolbar should now be visible in your environment.

## Usage

1. With the addin toolbar enabled, select a string of text in the IDE.
2. Click a button on the toolbar to launch the sparks knowledgebase with the selected string.
  - a. (3ds max icon): Searches the entire knowledgebase.
  - b. (letter icon): Searches all webboard support threads, legacy support threads, developer techdocs, and solutions.
  - c. (class icon): Searches the latest 3ds max SDK documentation only. Very useful for class/method lookup.
3. Once inside the sparks knowledgebase, you can alter your search as well as use 'and/or' keywords to refine your search.

## **Local Address**

The addin can be found in the maxsdk help directory of the 5.1 install. Please read the SparksAddinReadme therein for usage and installation instructions.

## Internet Address

[http://sparks.discreet.com/downloads/downloadshome.cfm?  
f=2&wf\\_id=87](http://sparks.discreet.com/downloads/downloadshome.cfm?f=2&wf_id=87)

## Sparks Addin for MS VC++ 7 IDE

See Also: [Sparks Developer Program](#), [Sparks Developer Knowledgebase](#)

## Overview

New for 3ds max 6 is the Sparks Addin for use with the MS VC++ 7 IDE. Its functionality allows 3ds max developers to search the sparks knowledgebase for the text that's selected in the IDE.

**The Sparks Addin is available to the public.**

## **Installation**

- 1. Follow the installation instructions in the included installer.**

## Usage

1. With the addin toolbar enabled, select a string of text in the IDE.
2. Click the 'Search 3ds max SDK Knowledgebase' button to launch the search. Below are descriptions for items in the Collection dropdown
  - a. Entire Knowledgebase:** Searches the entire knowledgebase.
  - b. Support Threads:** Searches all webboard support threads, legacy support threads, developer techdocs, and solutions.
  - c. SDK Documentation:** Searches the latest 3ds max SDK documentation only. Very useful for class/method lookup.
3. Once inside the sparks knowledgebase, you can alter your search as well as use 'and/or' keywords to refine your search.

**Internet Address**

[http://sparks.discreet.com/downloads/downloadshome.cfm?f=2&wf\\_id=105](http://sparks.discreet.com/downloads/downloadshome.cfm?f=2&wf_id=105)

## Class GeomObject

See Also: [Class Object](#), [Class Mesh](#).

class GeomObject : public Object

### Description:

This is the base class for the creation of Geometric Object plug-ins. This class represents an object that has geometry and is renderable.

### Methods:

#### Prototype:

**virtual int IsInstanceDependent()**

#### Remarks:

Implemented by the Plug-In.

If an object creates different meshes depending on the particular instance (view-dependent) it should return nonzero; otherwise 0.

#### Default Implementation:

**{ return 0; }**

#### Prototype:

**virtual Mesh\* GetRenderMesh(TimeValue t, INode \*inode, View& view, BOOL& needDelete);**

#### Remarks:

Implemented by the Plug-In.

This method should be implemented by all renderable GeomObjects. It provides a mesh representation of the object for use by the renderer. Primitives that already have a mesh cached can just return a pointer to it (and set **needDelete** to FALSE).

In release 3.0 and later, any implementations of this method which take a long time should periodically call **View::CheckForRenderAbort()** to see if the user has canceled the render. If canceled, the function can either return NULL, or return a non null pointer with the appropriate value for **needDelete**. (If **needDelete** is TRUE a non-null mesh will be deleted.)

#### Parameters:

**TimeValue t**

The time to get the mesh.

**INode \*inode**

The node in the scene.

**View& view**

If the renderer calls this method it will pass the view information here. See [Class View](#).

**BOOL& needDelete**

Set to TRUE if the renderer should delete the mesh, FALSE otherwise.

**Return Value:**

A pointer to the mesh object.

**Prototype:**

```
virtual PatchMesh* GetRenderPatchMesh(TimeValue, INode  
*inode, View& view, BOOL& needDelete);
```

**Remarks:**

Implemented by the Plug-In.

This method provides a patch mesh representation of the object for use by the renderer. If this method returns NULL, then **GetRenderMesh()** will be called.

**Parameters:****TimeValue t**

The time to get the patch mesh.

**INode \*inode**

The node in the scene.

**View& view**

If the renderer calls this method it will pass the view information here. See [Class View](#).

**BOOL& needDelete**

Set to TRUE if the renderer should delete the patch mesh, FALSE otherwise.

**Return Value:**

A pointer to the patch mesh. See [Class PatchMesh](#).

**Prototype:**

```
virtual int NumberOfRenderMeshes();
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

Objects may supply multiple render meshes ( e.g. particle systems). If this method returns a positive number, then **GetMultipleRenderMesh** and **GetMultipleRenderMeshTM** will be called for each mesh, instead of calling **GetRenderMesh**.

**Return Value:**

The number of render meshes, or 0 to indicate that multiple meshes aren't supported.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual Mesh* GetMultipleRenderMesh(TimeValue t, INode  
*inode, View& view, BOOL& needDelete, int meshNumber);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

For multiple render meshes, this method must be implemented. set **needDelete** to TRUE if the render should delete the mesh, FALSE otherwise.

**Parameters:**

**TimeValue t**

The time at which to obtain the mesh.

**INode \*inode**

The pointer to the node.

**View& view**

A reference to the view.

**BOOL& needDelete**

TRUE if the mesh needs to be deleted, otherwise FALSE.

**int meshNumber**

Specifies which of the multiple meshes is being asked for.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void GetMultipleRenderMeshTM(TimeValue t, INode  
*inode, View& view, int meshNumber, Matrix3& meshTM,  
Interval& meshTMValid);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

For multiple render meshes, this method must be implemented.

**Parameters:**

**TimeValue t**

The time at which to obtain the mesh.

**INode \*inode**

The pointer to the node.

**View& view**

A reference to the view.

**int meshNumber**

Specifies which of the multiple meshes is being asked for.

**Matrix3& meshTM**

Should be returned with the transform defining the offset of the particular mesh in object space.

**Interval& meshTMValid**

Should contain the validity interval of **meshTM**.

**Default Implementation:**

```
{ return; }
```

**Prototype:**

```
virtual BOOL CanDoDisplacementMapping();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if this object can do displacement mapping; otherwise FALSE.

**Default Implementation:**

```
{ return 0; }
```

## Class SimpleObject

See Also: [Class GeomObject](#), [Class IParamBlock](#), [Class Mesh](#).

```
class SimpleObject : public GeomObject
```

### Description:

This is a base class for creating procedural objects. This class implements many of the methods required to create a procedural object. The only limitation for a procedural object using **SimpleObject** as a base class is that it must represent itself with a mesh.

### Data Members:

Note: Methods of the base class refer to these data members. For example the base class implementations of the bounding box methods use the **mesh** data member. Therefore the plug-in derived from SimpleObject must use these same data members.

public:

#### **IParamBlock \*pblock;**

The parameter block for managing the object's parameters.

#### **Mesh mesh;**

The mesh object that is built by **BuildMesh()**.

#### **Interval ivalid;**

The validity interval for the **mesh**. This interval is used to determine how **BuildMesh()** is called. If this interval is not set **BuildMesh()** will be called over and over as the system won't know when the mesh is valid or not. Make sure you set this interval to accurately reflect the validity interval for the mesh.

#### **BOOL suspendSnap;**

If TRUE, this causes no snapping to occur. This is commonly used to prevent an object from snapping to itself when it is creating. For example, in the mouse proc used to create an object, the following code is often used when snapping mouse points:

```
ob->suspendSnap = TRUE;
```

```
p0 = vpt->SnapPoint(m,m,NULL,SNAP_IN_PLANE);
```

This disables snapping temporarily to keep the object from snapping to itself.

Procedural Object plug-ins which subclass off SimpleObject must implement

these methods. The default implementations are noted.

**Methods:**

**Prototype:**

**virtual void BuildMesh(TimeValue t)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to build the mesh representation of the object using its parameter settings at the time passed. The plug-in should use the data member **mesh** to store the built mesh.

**Parameters:**

**TimeValue t**

The time at which to build the mesh.

**Prototype:**

**virtual ParamDimension \*GetParameterDim(int pbIndex)**

**Remarks:**

Implemented by the Plug-In.

This method returns the parameter dimension of the parameter whose index is passed.

**Parameters:**

**int pbIndex**

The index of the parameter to return the dimension of.

**Return Value:**

Pointer to a ParamDimension.

**Example:**

**return stdNormalizedDim;**

**Default Implementation:**

The default implementation returns **defaultDim**.

See Also: [ParamDimension](#)

**Prototype:**

```
virtual TSTR GetParameterName(int pbIndex)
```

**Remarks:**

Implemented by the Plug-In.

This method returns the name of the parameter whose index is passed.

**Parameters:**

```
int pbIndex
```

The index of the parameter to return the name of.

**Return Value:**

The name of the parameter.

**Default Implementation:**

```
The default implementation returns TSTR(_T("Parameter"))
```

**Prototype:**

```
virtual void InvalidateUI()
```

**Remarks:**

Implemented by the Plug-In.

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

**Example:**

If the plug-in uses a parameter map for handling its UI, it may call a method of the parameter map to handle this: **pmapParam->Invalidate();**

If the plug-in does not use parameter maps, it should call the **SetValue()** method on each of its controls that display a value, for example the spinner controls. This will cause to the control to update the value displayed. The code below shows how this may be done for a spinner control. Note that ip and pblock are assumed to be initialized interface and parameter block pointers

```
(IObjParam *ip, IParamBlock *pblock).
```

```
float newval;
```

```
Interval valid=FOREVER;
```

```
TimeValue t=ip->GetTime();
```

```
// Get the value from the parameter block at the current time.  
pblock->GetValue( PB_ANGLE, t, newval, valid );  
// Set the value. Note that the notify argument is passed as  
FALSE.  
// This ensures no messages are sent when the value changes.  
angleSpin->SetValue( newval, FALSE );
```

**Prototype:**

```
virtual BOOL OKtoDisplay(TimeValue t)
```

**Remarks:**

Implemented by the Plug-In.

This method returns a BOOL to indicate if it is okay to draw the object at the time passed. Normally it is always OK to draw the object, so the default implementation returns TRUE. However for certain objects it might be a degenerate case to draw the object at a certain time (perhaps the size went to zero for example), so these objects could return FALSE.

**Parameters:**

**TimeValue t**

The time at which the object would be displayed.

**Default Implementation:**

```
{ return TRUE; }
```

**Return Value:**

TRUE if the object may be displayed; otherwise FALSE.

## Class ParticleObject

See Also: [Class GeomObject](#), [Class ForceField](#), [Class CollisionObject](#), [Class ShadeContext](#).

```
class ParticleObject : public GeomObject
```

### Description:

This is the base class for creating particle system plug-ins.

Many particle systems may be derived from class **SimpleParticle** instead of this class. See [Class SimpleParticle](#) for more details.

Note: This class is derived from GeomObject and still has **GEOMOBJECT\_CLASS\_ID** as its super class. To determine if an object is a ParticleObject, call:

**Animatable::GetInterface()** with the ID **I\_PARTICLEOBJ** or use the macro:

**GetParticleInterface(anim)** where **anim** is the object in question. This will return a **ParticleObject\*** or **NULL**. See [Class Animatable](#) .

Note: See the method **Animatable::GetProperty()** for details on choosing the method used to evaluate the particle system during motion blur rendering. See [Class Animatable](#) .

### Methods:

#### Prototype:

```
virtual void ApplyForceField(ForceField *ff)=0;
```

#### Remarks:

Implemented by the Plug-In.

This method is called to add the force field object passed to the list of force field objects operating on this particle system.

#### Parameters:

**ForceField \*ff**

Points to an instance of a ForceField object.

#### Sample Code:

```
void SimpleParticle::ApplyForceField(ForceField *ff) {  
    fields.Append(1,&ff);  
}
```

```
}
```

**Prototype:**

```
virtual BOOL ApplyCollisionObject(CollisionObject *co)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called to add the collision object passed to the list of collision objects operating on this particle system.

**Parameters:**

**CollisionObject \*co**

Points to an instance of a collision object.

**Return Value:**

If a particle does not support this method it should return FALSE; otherwise return TRUE.

**Sample Code:**

```
BOOL SimpleParticle::ApplyCollisionObject(CollisionObject *co)
{
    cobjs.Append(1,&co);
    return TRUE;
}
```

**Prototype:**

```
int IsDeformable();
```

**Remarks:**

Implemented by the System.

This method returns TRUE to indicate it is deformable. A particle object is deformable, but does not let itself be deformed using the usual **GetPoint()** / **SetPoint()** methods. Instead a space warp must apply a force field to deform the particle system.

**Prototype:**

```
BOOL CanCacheObject();
```

**Remarks:**

Implemented by the System.

This method returns FALSE to indicate the object cannot be cached. Particle objects don't perform a shallow copy and therefore cannot be cached.

**Prototype:**

```
virtual BOOL NormalAlignVector(TimeValue t,Point3 &pt, Point3 &norm);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

This method is inherited from [Class Object](#). This is a default implementation provided for particle systems.

**Parameters:**

**TimeValue t**

The time to compute the normal align vector.

**Point3 &pt**

The point of intersection.

**Point3 &norm**

The normal at the point of intersection.

**Return Value:**

TRUE if this method is implemented to return the normal align vector; otherwise FALSE.

**Default Implementation:**

```
{pt=Point3(0,0,0);norm=Point3(0,0,-1);return TRUE;}
```

**Prototype:**

```
virtual Point3 ParticlePosition(TimeValue t,int i);
```

**Remarks:**

Implemented by the Plug-In.

This method is available in release 2.0 and later only.

Returns the position of the specified particle in world space at the time passed.

The Particle Age texture map and the Particle Motion Blur texture map use this method.

**Parameters:**

**TimeValue t**

The time to return the particle position.

**int i**

The index of the particle.

Note: When a texture map calls these methods, the particle index **i** is passed to the texmap in the data member **ShadeContext::mtlNum**. The particle systems encode the index of the particle associated with the face of the particle mesh being shaded into the **mtlNum**. For instance, once the particle system generates a mesh to be rendered, every face of the mesh corresponds to a particle. This isn't a one-to-one correspondance because there are more faces than particles (if the particles are represented as tetrahedrons there are four faces per particle). When a texture map or material that is shading a mesh generated by a particle system wants to know which particle the face is associated with it gets this info out of the **ShadeContext::mtlNum**.

For example, here is a fragment of the code from the Particle Age texture map where it evaluates the color of the point being shaded:

```
AColor PartAgeTex::EvalColor(ShadeContext& sc)  
{  
  ...  
  // Evaluate...  
  Object *ob = sc.GetEvalObject();  
  if (ob && ob->IsParticleSystem()) {  
    ParticleObject *obj = (ParticleObject*)ob;  
    TimeValue t = sc.CurTime();  
    TimeValue age = obj->ParticleAge(t,sc.mtlNum);  
    TimeValue life = obj->ParticleLife(t,sc.mtlNum);  
    ...etc.  
  }
```

**Default Implementation:**

```
{return Point3(0,0,0);}
```

**Prototype:**

```
virtual Point3 ParticleVelocity(TimeValue t,int i);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Returns the velocity of the specified particle at the time passed (in 3ds max units per tick). This is specified as a vector. The Particle Age texture map and the Particle Motion Blur texture map use this method.

**Parameters:**

**TimeValue t**

The time to return the particle velocity.

**int i**

The index of the particle.

**Default Implementation:**

```
{return Point3(0,0,0);};
```

**Prototype:**

```
virtual float ParticleSize(TimeValue t,int i);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Returns the world space size of the specified particle in at the time passed.

The Particle Age texture map and the Particle Motion Blur texture map use this method.

**Parameters:**

**TimeValue t**

The time to return the particle size.

**int i**

The index of the particle.

**Default Implementation:**

```
{return 0.0f;};
```

**Prototype:**

```
virtual int ParticleCenter(TimeValue t,int i);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Returns a value indicating where the particle geometry (mesh) lies in relation to the particle position.

This is used by Particle Motion Blur for example. It gets the point in world space of the point it is shading, the size of the particle from **ParticleSize()**, and the position of the mesh from **ParticleCenter()**. Given this information, it can know where the point is, and it makes the head and the tail more transparent.

**Parameters:**

**TimeValue t**

The time to return the particle center.

**int i**

The index of the particle.

**Return Value:**

One of the following:

**PARTCENTER\_HEAD**

The particle geometry lies behind the particle position.

**PARTCENTER\_CENTER**

The particle geometry is centered around particle position.

**PARTCENTER\_TAIL**

The particle geometry lies in front of the particle position.

**Default Implementation:**

```
{return PARTCENTER_CENTER;}
```

**Prototype:**

```
virtual TimeValue ParticleAge(TimeValue t, int i);
```

**Remarks:**

Implemented by the Plug-In.

This method is available in release 2.0 and later only.

Returns the age of the specified particle -- the length of time it has been 'alive'.

The Particle Age texture map and the Particle Motion Blur texture map use this method.

**Parameters:**

**TimeValue t**

Specifies the time to compute the particle age.

**int i**

The index of the particle.

**Default Implementation:**

```
{return -1;}
```

**Prototype:**

```
virtual TimeValue ParticleLife(TimeValue t, int i);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Returns the life of the particle -- the length of time the particle will be 'alive'.

The Particle Age texture map and the Particle Motion Blur texture map use this method.

**Parameters:**

**TimeValue t**

Specifies the time to compute the particle life span.

**int i**

The index of the particle.

**Default Implementation:**

```
{return -1;}
```

**Prototype:**

```
virtual BOOL HasConstantTopology();
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

If a particle system has a fixed number of particles of fixed topology, then it can return TRUE for this method, and the renderer will then compute the image motion blur velocities based on the vertex motions, giving motion blur for rotating particles etc. If the particle system is topology-varying it should return FALSE.

**Default Implementation:**

```
{return FALSE;}
```

## Class SimpleParticle

See Also: [Class ParticleObject](#), [Marker Types](#), [Class Mesh](#), [Class ParticleSys](#), [Class Interval](#), [Template Class Tab](#), [Class ParamDimension](#).

```
class SimpleParticle : public ParticleObject
```

### Description:

This class provides a base class from which you may derive Particle System plug-ins. This class may be used by particle systems that fit within its form. The form is primarily dictated by the data members maintain by the class. The class maintains an instance of class ParticleSys that describes the particles. It also has a table of force fields and collision objects. The emitter for the particles is represented by a mesh. There is also a parameter block pointer available.

Particle system plug-ins that don't fit this form may derive from a base class without any constraints. See [Class ParticleObject](#) for more details.

### Data Members:

public:

#### **IParamBlock \*pblock;**

The parameter block pointer.

#### **ParticleSys parts;**

This is a description of the particles themselves (their count, position, velocities, ...).

#### **TimeValue tvalid;**

A particle system derived from SimpleParticle is valid at a particular time only (it does not have a validity interval). It is assumed to be always changing. This data member holds the time at which it is valid (when **tvalid** is TRUE).

#### **BOOL valid;**

This flag indicates if the particle system is valid. If TRUE, **tvalid** should contain the time it is valid for.

#### **Tab<ForceField\*> fields;**

The table of force fields affecting the particles.

#### **Tab<CollisionObject\*> cobjs;**

The table of collision objects affecting the particles.

#### **Mesh mesh;**

The mesh object that represents the emitter.

**Interval mvalid;**

The validity interval for the emitter mesh. If the mesh is invalid

**BuildEmitter()** will be called.

**static SimpleParticle \*editOb;**

The SimpleParticle object that is being edited between **BeginEditParams()** and **EndEditParams()**.

**static IObjParam \*ip;**

Storage for the interface pointer passed into **BeginEditParams()**. This pointer is only valid between **BeginEditParams()** and **EndEditParams()**.

**Methods:**

**Prototype:**

**SimpleParticle();**

**Remarks:**

Constructor. The **pblock** is initialized to NULL, the **mvalid** interval is set to empty, and **valid** is set to FALSE.

**Prototype:**

**virtual void UpdateParticles(TimeValue t, INode \*node)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called so the particle system can update its state to reflect the current time passed. This may involve generating new particle that are born, eliminating old particles that have expired, computing the impact of collisions or force field effects, and modify the positions and velocities of the particles.

**Parameters:**

**TimeValue t**

The particles should be updated to reflect this time.

**INode \*node**

This is the emitter node. Particles system are world space objects so they are not instanced. This means that the particle system can depend on the node's world space position.

**Sample Code:**

For example code see `\MAXSDK\SAMPLES\OBJECTS\RAIN.CPP`.

**Prototype:**

```
void SetParticlePosition(TimeValue t, int i, Point3 pos);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the position of the specified particle at the specified time.

**Parameters:****TimeValue t**

The time at which to set the particle position.

**int i**

The zero based index of the particle to set.

**Point3 pos**

The position to set.

**Prototype:**

```
void SetParticleVelocity(TimeValue t, int i, Point3 vel);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the velocity of the specified particle at the specified time (in 3ds max units per tick).

**Parameters:****TimeValue t**

The time at which to set the particle velocity.

**int i**

The zero based index of the particle to set.

**Point3 vel**

The velocity to set.

**Prototype:**

```
void SetParticleAge(TimeValue t, int i, TimeValue age);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the age of the specified particle at the specified time.

**Parameters:****TimeValue t**

The time at which to set the particle age.

**int i**

The zero based index of the particle to set.

**TimeValue age**

The age to set.

**Prototype:**

**virtual void BuildEmitter(TimeValue t, Mesh& amesh)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to allow the plug-in to provide a representation of its emitter in the 3D viewports.

**Parameters:****TimeValue t**

Specifies the time to build the emitter.

**Mesh& amesh**

Store the built mesh representation here.

**Prototype:**

**virtual Interval GetValidity(TimeValue t)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to retrieve the validity time of the particle system emitter.

**Parameters:****TimeValue t**

The time to compute the validity interval.

**Return Value:**

The validity interval of the particle system emitter at the specified time.

**Prototype:**

```
virtual MarkerType GetMarkerType();
```

**Remarks:**

Implemented by the Plug-In.

Returns one of the defined marker types to use when displaying particles.

**Return Value:**

One of the following values:

See [Marker Types](#).

**Default Implementation:**

```
{return POINT_MRKR;}
```

**Prototype:**

```
virtual BOOL OKtoDisplay(TimeValue t);
```

**Remarks:**

Implemented by the Plug-In.

This method is called to determine if the particle emitter is okay to display at the specified time. If at certain times it is not okay to display this method should return FALSE. This might occur if a size goes to 0. Normally however it is always okay to display so the default implementation returns TRUE.

**Parameters:**

**TimeValue t**

The time to display the emitter.

**Return Value:**

TRUE if it is okay to display, FALSE otherwise.

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
virtual BOOL EmitterVisible();
```

**Remarks:**

Implemented by the Plug-In.

This method is called to determine if the particle emitter is visible in the viewports. If the plug-in provides a UI control to toggle the emitter on and off, this method should return the state of this control.

**Return Value:**

TRUE if the emitter is visible; otherwise FALSE.

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
virtual void InvalidateUI();
```

**Remarks:**

Implemented by the Plug-In.

It is important the user interface controls display values that reflect the current time. This method is called if the user interface parameters needs to be updated because the user moved to a new time.

**Example:**

If the plug-in uses a parameter map for handling its UI, it may call a method of the parameter map to handle this: **pmapParam->Invalidate();**

If the plug-in does not use parameter maps, it should call the **SetValue()** method on each of its controls that display a value, for example the spinner controls. This will cause to the control to update the value displayed. The code below shows how this may be done for a spinner control. Note that ip and pblock are assumed to be initialized interface and parameter block pointers (**IObjParam \*ip, IParamBlock \*pblock**).

```
float newval;
```

```
Interval valid=FOREVER;
```

```
TimeValue t=ip->GetTime();
```

```
// Get the value from the parameter block at the current time.
```

```
pblock->GetValue( PB_ANGLE, t, newval, valid );
```

```
// Set the value. Note that the notify argument is passed as  
FALSE.
```

```
// This ensures no messages are sent when the value changes.  
angleSpin->SetValue( newval, FALSE );
```

**Prototype:**

```
virtual ParamDimension *GetParameterDim(int pbIndex);
```

**Remarks:**

Implemented by the Plug-In.

This method returns the parameter dimension of the parameter whose index is passed.

**Parameters:**

**int pbIndex**

The index of the parameter to return the dimension of.

**Return Value:**

Pointer to a ParamDimension.

**Example:**

```
return stdNormalizedDim;
```

**Default Implementation:**

The default implementation returns **defaultDim**.

See Also: [ParamDimension](#)

**Prototype:**

```
virtual TSTR GetParameterName(int pbIndex);
```

**Remarks:**

Implemented by the Plug-In.

This method returns the name of the parameter whose index is passed.

**Parameters:**

**int pbIndex**

The index of the parameter to return the name of.

**Return Value:**

The name of the parameter.

**Default Implementation:**

The default implementation returns **TSTR(\_T("Parameter"))**

## Class PatchObject

See Also: [Class GeomObject](#), [Class IPatchOps](#), [Class IPatchSelect](#), [Class IPatchSelectData](#), [Class ISubMtlAPI](#), [Class AttachMatDlgUser](#), [Class PatchMesh](#), [Class Patch](#), [Class Mesh](#), [Class TessApprox](#), [Working with Patches](#).

```
class PatchObject : public GeomObject, IPatchOps, IPatchSelect,  
IPatchSelectData, ISubMtlAPI, AttachMatDlgUser
```

### Description:

This class is the base class for the creation of Patch objects. This class stores an instance of a **PatchMesh** that holds all the Patches that make up this patch object. This class also maintains a **Mesh** cache. All methods of this class are implemented by the system.

### Data Members:

public:

#### **PatchMesh patch;**

The patch mesh for this patch object.

#### **Mesh mesh;**

The Mesh cache.

#### **BOOL meshValid;**

Indicates if the mesh cache is valid.

#### **BOOL showMesh;**

Indicates if the mesh is shown in the viewports

#### **GenericNamedSelSetList vselSet;**

This data member is available in release 3.0 and later only.

Vertex level named selection sets.

#### **GenericNamedSelSetList eselSet;**

This data member is available in release 3.0 and later only.

Edge level named selection sets.

#### **GenericNamedSelSetList pselSet;**

This data member is available in release 3.0 and later only.

Patch level named selection sets.

#### **int patchSelSubType;**

This data member is available in release 4.0 and later only.

The sub-object selection level, defined by;

### **PO\_PATCH**

Patch sub-object level. When SetSubobjectLevel(PO\_PATCH) is called, both the PatchMesh selection level and patchSelSubType are set to PO\_PATCH.

### **PO\_ELEMENT**

Element sub-object level. When SetSubobjectLevel(PO\_ELEMENT) is called, the PatchMesh selection level is set to PO\_PATCH and patchSelSubType is set to PO\_ELEMENT.

### **Methods:**

#### **Prototype:**

**PatchObject();**

#### **Remarks:**

Constructor.

#### **Prototype:**

**~PatchObject();**

#### **Remarks:**

Destructor.

#### **Prototype:**

**void UpdatePatchMesh(TimeValue t);**

#### **Remarks:**

This method is available in release 2.0 and later only.

This should be implemented by classes derived from PatchObject whose patches change over time.

#### **Parameters:**

**TimeValue t**

The time to update the patch mesh.

#### **Default Implementation:**

**{}**

**Prototype:**

**void PrepareMesh(TimeValue t);**

**Remarks:**

This method checks to see if the mesh cache is up to date, and if not, it generates it.

**Parameters:**

**TimeValue t**

This parameter is available in release 2.0 and later only.

The mesh cache should be generated to reflect this time.

**Prototype:**

**BOOL ShowLattice();**

**Remarks:**

Returns TRUE if the patch lattice is displayed; otherwise FALSE.

**Prototype:**

**BOOL ShowVerts();**

**Remarks:**

Returns TRUE if the patch vertices are shown; otherwise FALSE.

**Prototype:**

**void SetShowLattice(BOOL sw);**

**Remarks:**

Sets the state of the lattice display switch.

**Parameters:**

**BOOL sw**

TRUE to turn on the lattice display; FALSE to turn it off.

**Prototype:**

**void SetShowVerts(BOOL sw);**

**Remarks:**

Sets the state of the vertex display switch

**Parameters:**

**BOOL sw**

TRUE to turn on the vertex display; FALSE to turn it off.

**Prototype:**

**void SetMeshSteps(int steps);**

**Remarks:**

Sets the number of mesh steps (viewport).

**Parameters:**

**int steps**

The number of steps to set.

**Prototype:**

**int GetMeshSteps();**

**Remarks:**

Returns the number of mesh steps (viewport).

**Prototype:**

**void SetMeshStepsRender(int steps);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the Surface Render Steps setting.

**Parameters:**

**int steps**

The value to set.

**Prototype:**

**int GetMeshStepsRender();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the Surface Render Steps setting.

**Prototype:**

**void SetShowInterior(BOOL si);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the 'Show Interior Edges' value.

**Parameters:**

**BOOL si**

TRUE for on; FALSE for off.

**Prototype:**

**BOOL GetShowInterior();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the 'Show Interior Edge' setting; TRUE if on; FALSE if off.

**Prototype:**

**void SetAdaptive(BOOL sw);**

**Remarks:**

Sets the state of the adaptive switch.

**Parameters:**

**BOOL sw**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**BOOL GetAdaptive();**

**Remarks:**

Returns the state of the adaptive switch. TRUE is on; FALSE is off.

**Prototype:**

**void SetViewTess(TessApprox tess);**

**Remarks:**

Sets the tessellation approximation object used for viewport rendering.

**Parameters:**

**TessApprox tess**

The tessellation approximation object to be used for viewport rendering.

**Prototype:**

**TessApprox GetViewTess();**

**Remarks:**

Returns the tessellation approximation object used for rendering in the viewports.

**Prototype:**

**void SetProdTess(TessApprox tess);**

**Remarks:**

Sets the tessellation approximation object used for production rendering.

**Parameters:**

**TessApprox tess**

The tessellation approximation object to be used for production rendering.

**Prototype:**

**TessApprox GetProdTess();**

**Remarks:**

Returns the tessellation approximation object used for production rendering.

**Prototype:**

**void SetDispTess(TessApprox tess);**

**Remarks:**

Sets the tessellation approximation object used for display in the viewports.

**Parameters:**

## **TessApprox tess**

The tessellation approximation object to be used for the viewports.

### **Prototype:**

**TessApprox GetDispTess();**

### **Remarks:**

Returns the tessellation approximation object used for display in the viewports.

### **Prototype:**

**BOOL GetViewTessNormals();**

### **Remarks:**

Returns TRUE if normals are used from the viewport tesselator; otherwise FALSE.

### **Prototype:**

**void SetViewTessNormals(BOOL use);**

### **Remarks:**

Sets if normals are used from the viewport tesselator.

### **Parameters:**

**BOOL use**

TRUE to use normals; FALSE to not use them.

### **Prototype:**

**BOOL GetProdTessNormals();**

### **Remarks:**

Returns TRUE if normals are used from the production renderer tesselator; otherwise FALSE.

### **Prototype:**

**void SetProdTessNormals(BOOL use);**

### **Remarks:**

Sets if normals are used from the production renderer tesselator.

**Parameters:**

**BOOL use**

TRUE to use normals; FALSE to not use them.

**Prototype:**

**BOOL GetViewTessWeld();**

**Remarks:**

Returns TRUE if the viewport mesh is welded after tessellation; otherwise FALSE.

**Prototype:**

**void SetViewTessWeld(BOOL weld);**

**Remarks:**

Sets if the viewport mesh is welded after tessellation; otherwise FALSE.

**Parameters:**

**BOOL weld**

TRUE to weld; FALSE to not weld.

**Prototype:**

**BOOL GetProdTessWeld();**

**Remarks:**

Returns TRUE if the production renderer mesh is welded after tessellation; otherwise FALSE.

**Prototype:**

**void SetProdTessWeld(BOOL weld);**

**Remarks:**

Sets if the production renderer mesh is welded after tessellation; otherwise FALSE.

**Parameters:**

**BOOL weld**

TRUE to weld; FALSE to not weld.

**Prototype:**

**void InvalidateMesh();**

**Remarks:**

Invalidates the mesh cache.

**Prototype:**

**void SetFlag(DWORD fl, BOOL val=TRUE);**

**Remarks:**

This method is available in release 4.0 and later only.

This method sets or clears the status of the Show End Result flag.

**Parameters:**

**DWORD fl**

The flag you wish to set or clear. Currently the only flag defined is the Show End Result flag **EP\_DISP\_RESULT**.

**BOOL val**

Specifies if the given flag should be set or cleared.

**Prototype:**

**void ClearFlag (DWORD fl);**

**Remarks:**

This method is available in release 4.0 and later only.

This method clears the status of the Show End Result flag.

**Parameters:**

**DWORD fl**

The flag you wish to set or clear. Currently the only flag defined is the Show End Result flag **EP\_DISP\_RESULT**.

**Prototype:**

**bool GetFlag(DWORD fl);**

**Remarks:**

This method is available in release 4.0 and later only.

This methods allows you to obtain the status of the Show End Result flag.

**Parameters:**

**DWORD fl**

The flag you wish to set or clear. Currently the only flag defined is the Show End Result flag **EP\_DISP\_RESULT**.

**Prototype:**

**void ShowEndResultChanged(BOOL showEndResult);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is called by the system then the status of the Show End Result function changes (ie. the Show End Results button has been toggled on or off).

Note that setting the state of the Show End Result is done through the

**Interface::SetShowEndResult()** method.

**Parameters:**

**BOOL showEndResult**

This flag specifies the Show End Result status, which is TRUE if on; FALSE if off.

**Prototype:**

**int Display(TimeValue t, INode\* inode, ViewExp \*vpt, int flags, ModContext \*mc);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used to display the gizmo version of the patch mesh.

**Parameters:**

**TimeValue t**

The time to display the object.

**INode\* inode**

The node to display.

**ViewExp\* vpt**

An interface pointer that may be used to call methods associated with the

viewports.

**int flags**

The display flags. See the [List of Display Flags](#) for more information.

**ModContext\* mc**

A pointer to the modifiers ModContext.

**Prototype:**

**void GetWorldBoundingBox (TimeValue t, INode \* inode, ViewExp\* vp, Box3& box, ModContext \*mc);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the world space bounding box for the gizmo version of the patch mesh.

**Parameters:**

**TimeValue t**

The time to compute the bounding box.

**INode\* inode**

The node to calculate the bounding box for.

**ViewExp\* vpt**

An interface pointer that may be used to call methods associated with the viewports.

**Box3& box**

The bounding box which was computed.

**ModContext\* mc**

A pointer to the modifiers ModContext.

**Prototype:**

**int GetSubobjectType();**

**Remarks:**

This method is available in release 4.0 and later only.

This method goes hand-in-hand with **GetSubobjectLevel()**, except that this method returns the type of geometry that is actually being acted upon.

**Return Value:**

The sub-object type, either **PO\_PATCH** or **PO\_ELEMENT**.

**Prototype:**

**Color GetVertColor(int mp=0, bool \*differs=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the common color for all selected vertices. If no vertices are selected then white (1,1,1) will be returned, however, if multiple vertices with different colors are selected, then black (0,0,0) will be returned.

**Parameters:**

**int mp=0**

The map channel.

**bool \*differs=NULL**

This parameter is returned to indicate if there were any differences.

**Prototype:**

**void SetVertColor(Color clr, int mp=0);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will set all selected vertices to the specified color.

**Parameters:**

**Color clr**

The color you wish to apply to all the selected vertices.

**int mp=0**

The map channel.

**Prototype:**

**Color GetPatchColor(int mp=0, bool \*differs=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the common color for all selected patches. If no patches are selected then white (1,1,1) will be returned, however, if different vertex colors are present in the selected patches, then black (0,0,0) will be returned.

**Parameters:**

**int mp=0**

The map channel.

**bool \*differs=NULL**

This parameter is returned to indicate if there were any differences.

**Prototype:**

**void SetPatchColor(Color clr, int mp=0);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will set all selected patches to the specified color.

**Parameters:**

**Color clr**

The color you wish to apply to all the selected patches.

**int mp=0**

The map channel.

**Prototype:**

**void SelectVertByColor(VertColor clr, int deltaR, int deltaG, int deltaB, BOOL add, BOOL sub, int mp=0);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will select all vertices which fall into a specified color range.

**Parameters:**

**VertColor clr**

The starting color of the vertices you wish to select by color.

**int deltaR**

The difference range for the red color component.

**int deltaG**

The difference range for the green color component.

**int deltaB**

The difference range for the blue color component.

**BOOL add**

This flag adds vertices to the selection that fall into the color range.

**BOOL sub**

This flag subtracts vertices from the selection that fall into the color range.

**int mp=0**

The map channel.

**Prototype:**

```
void SelectOpenEdges();
```

**Remarks:**

This method is available in release 4.0 and later only.

This method examines the patch mesh and selects any edges used by only one single patch.

**Prototype:**

```
void DoBevel(TimeValue t);
```

**Remarks:**

When called with the *Animate* state active and on a nonzero *TimeValue*, this method will prepare the controllers for the geometry that is being created. The program can then fill in the animated vertex values later.

**Parameters:****TimeValue t**

This parameter is available in release 4.0 and later only.

The time at which to prepare and execute the bevel operation.

**Prototype:**

```
void DoExtrude(TimeValue t);
```

**Remarks:**

When called with the Animate state active and on a nonzero TimeValue, this method will prepare the controllers for the geometry that is being created. The program can then fill in the animated vertex values later.

**Parameters:**

**TimeValue t**

This parameter is available in release 4.0 and later only.

The time at which to prepare and execute the extrude operation.

**Prototype:**

**void DoCreateShape();**

**Remarks:**

This method is available in release 4.0 and later only.

This method will create a bezier spline shape from the selected edges of the patch mesh. Each edge will become a separate spline in the output shape. The user will be prompted to enter a name for the new editable spline object that will be created.

**Prototype:**

**void DoEdgeWeld();**

**Remarks:**

This method is available in release 4.0 and later only.

This method will perform the edge weld function on the patch object. Note that this does not take into account any threshold but welds edges only if they use the same two vertices as endpoints. When two or more edges are welded, the locations of the edge vectors are averaged to create the new edge.

**Prototype:**

**void DoFlipNormals(int patchIndex = -1);**

**Remarks:**

This method is available in release 4.0 and later only.

This method flips the normals of a specified patch or all selected patches. This method will save undo information and displays a prompt if **patchIndex < 0** while there are no patches selected.

**Parameters:****int patchIndex**

The index of the patch for which to flip the normal. If this parameter is  $< 0$ , the normals of all selected patches will be flipped (if there are any selected).

Any vertices set to **PVERT\_COPLANAR** that lie on the boundary between flipped and unflipped patches will have their type set to

**PVERT\_CORNER**. This is because attempting to compute normals of neighboring patches with opposite normals in order to get a proper plane often results in invalid normals being generated. Making the vertex a corner type prevents the problem. Any vertices not on the boundary between flipped and unflipped patches are left as is.

**Prototype:**

**void DoUnifyNormals();**

**Remarks:**

This method is available in release 4.0 and later only.

This method examines the selected patch set and attempts to make them all face the same direction. Preferred direction is arbitrary; the first selected patch encountered in each contiguous group determines the direction all patches in that group will attain.

Any vertices set to **PVERT\_COPLANAR** that lie on the boundary between flipped and unflipped patches will have their type set to

**PVERT\_CORNER**. This is because attempting to compute normals of neighboring patches with opposite normals in order to get a proper plane often results in invalid normals being generated. Making the vertex a corner type prevents the problem. Any vertices not on the boundary between flipped and unflipped patches are left as is.

**Prototype:**

**void DoBreak(BOOL interactive = TRUE);**

**Remarks:**

This method is available in release 4.0 and later only.

In vertex mode, this method examines selected vertices, and if any of the vertices that are part of the selection set are used by more than one patch,

those vertices (and any attached vectors) are duplicated into separate geometry for each patch using it.

In edge mode, this method examines the selected edges and any vectors on the selected edges that are used by more than one patch are duplicated into separate geometry for each patch using them. Any vertices used by more than one selected edge are duplicated as well for patches on opposite sides of the edge.

Note: If the vertices and vectors involved have controllers attached, they are removed by this operation.

**Parameters:**

**BOOL interactive**

If this parameter is set to TRUE it will cause the method to display the appropriate prompts, create an undo object, and notifies the dependents.

**Prototype:**

**void DoDeleteSelected(BOOL interactive = TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

This method will delete the selected patches, exactly like the DeleteMesh modifier does.

**Parameters:**

**BOOL interactive**

This parameter is available in release 4.0 and later only.

If this parameter is set to TRUE it will cause the method to display the appropriate prompts, create an undo object, and notifies the dependents.

**Prototype:**

**void ChangeMappingTypeLinear(BOOL linear);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will change the mapping type of the selected patches to linear or curved.

**Parameters:**

## **BOOL linear**

If TRUE the mapping type will be changed to linear. FALSE will change the mapping type to curved.

### **Prototype:**

```
virtual void GetUIParam(patchUIParam uiCode, int &ret);
```

### **Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit patch parameters from the command panel. Currently not in use.

### **Parameters:**

**patchUIParam uiCode**

This enum is currently empty.

**int &ret**

The returned value.

### **Default Implementation:**

```
{ }
```

### **Prototype:**

```
virtual void SetUIParam(patchUIParam uiCode, int val);
```

### **Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit patch parameters from the command panel. Currently not in use.

### **Parameters:**

**patchUIParam uiCode**

This enum is currently empty.

**int val**

The value to set.

### **Default Implementation:**

```
{ }
```

**Prototype:**

**virtual void GetUIParam(patchUIParam uiCode, float &ret);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit patch parameters from the command panel. Currently not in use.

**Parameters:**

**patchUIParam uiCode**

This enum is currently empty.

**float &ret**

The returned value.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual void SetUIParam(patchUIParam uiCode, float val);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit patch parameters from the command panel. Currently not in use.

**Parameters:**

**patchUIParam uiCode**

This enum is currently empty.

**float val**

The value to set.

**Default Implementation:**

```
{ }
```

**Prototype:**

**bool Editing();**

**Remarks:**

This method is available in release 4.0 and later only.

This method will return TRUE if the SplineShape object or Edit Spline modifier is active in the command panel.

**Default Implementation:**

```
{ return (ip && (editObj==this)) ? TRUE : FALSE; }
```

## Class HelperObject:

See Also: [Class Object](#), [Class Animatable](#).

class HelperObject : public Object

### Description:

This is used as a base class to create helper object plug-ins. It simply provides implementations for a few of the methods of Animatable and Object.

### Methods:

#### Prototype:

**SClass\_ID SuperClassID();**

#### Remarks:

Implemented by the System.

Returns the super class ID of this plug-in type: **HELPER\_CLASS\_ID**

#### Prototype:

**int IsRenderable();**

#### Remarks:

Implemented by the System.

Returns 0 to indicate this object type may not be rendered.

#### Prototype:

**virtual void InitNodeName(TSTR& s);**

#### Remarks:

Implemented by the System.

Sets the default node name to "**Helper**".

#### Prototype:

**virtual int UsesWireColor();**

#### Remarks:

Implemented by the System.

Returns TRUE to indicate the object color is used for display.

## Class ConstObject

See Also: [Class HelperObject](#), [Class INode](#), [Class Object](#), [Class ViewExp](#), [Class Matrix3](#).

```
class ConstObject : public HelperObject
```

### Description:

This is a base class used to create construction grid objects. It implements a few of the methods of Animatable and Object and provides a few for working with construction grids.

### Methods:

#### Prototype:

```
int IsConstObject();
```

#### Remarks:

Implemented by the System.

Returns 1 to indicate this object is a construction grid object.

#### Prototype:

```
virtual void GetConstructionTM( TimeValue t, INode* inode,  
    ViewExp *vp, Matrix3 &tm ) = 0;
```

#### Remarks:

Implemented by the Plug-In.

This method returns the construction grid transformation matrix. This is the world space orientation and position of the construction plane.

#### Parameters:

##### **TimeValue t**

The time to retrieve the matrix.

##### **INode\* inode**

The node in the scene corresponding to the construction grid object.

##### **ViewExp \*vp**

The viewport the TM is being returned for. Certain construction grid objects might have a different plane for different viewports.

##### **Matrix3 &tm**

The transform matrix for this view is returned here.

**Prototype:**

**virtual Point3 GetSnaps(TimeValue t) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is specific to construction grids. The system calls this method to retrieve the snap dimension of the grid. In the 3ds max user interface for the construction grid helper object there is a spinner for 'Spacing'. This is the spacing for the grid. When **GetSnaps()** is called the Point3 returned will have this value in all three axes. This value is used, for example, when you create a box or other primitive and are setting the height dimension.

Note: In release 3.0 and later this method was changed to return a Point3 (not a Point3&).

**Parameters:**

**TimeValue t**

The time to retrieve the snap values.

## Class SimpleSpline

See Also: [Class ShapeObject](#).

```
class SimpleSpline : public ShapeObject
```

### Description:

Defines a simple spline object class to make spline primitives easier to create. This class provides default implementations for most of the **ShapeObject** methods. The plug-in derived from **SimpleSpline** must only implement a handful of methods to create a shape plug-in.

SimpleSpline plug-ins use a Super Class ID of **SHAPE\_CLASS\_ID**.

### Data Members:

public:

**IParamBlock \*ipblock;**

Interpolation parameter block (handled by **SimpleSpline**).

**IParamBlock \*pblock;**

User's parameter block. See [Class IParamBlock](#).

**static IParamMap \*ipmapParam;**

The parameter map. See [Class IParamMap](#).

**static int dlgSteps;**

The dialog steps settings.

**static BOOL dlgOptimize;**

The dialog Optimize toggle.

**static BOOL dlgAdaptive;**

The dialog Adaptive toggle.

**BezierShape shape;**

The Spline cache.

**Interval ivalid;**

The validity interval for the spline. See [Class Interval](#).

**BOOL suspendSnap;**

Flag to suspend snapping used during creation.

**static SimpleSpline \*editOb;**

This is the spline being edited in the command panel.

**Methods:**

**Prototype:**

**SimpleSpline();**

**Remarks:**

Constructor. The validity interval is set to empty, and the pblocks are set to NULL.

**Prototype:**

**~SimpleSpline();**

**Remarks:**

Destructor.

Clients of SimpleSpline need to implement these methods:

**Prototype:**

**virtual TCHAR \*GetObjectName() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the name that will appear in the history browser.

**Prototype:**

**virtual void InitNodeName(TSTR& s) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method retrieves the default name of the node when it is created.

**Parameters:**

**TSTR& s**

The name is stored here.

**Prototype:**

**virtual Class\_ID ClassID() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the unique Class\_ID of the plug-in. See [Class Class\\_ID](#) for more details.

**Prototype:**

**virtual void GetClassName(TSTR& s) = 0;**

**Remarks:**

Implemented by the Plug-In.

Retrieves the name of the plug-in class. This is used internally for debugging purposes.

**Parameters:**

**TSTR& s**

The name is stored here.

**Prototype:**

**virtual void BuildShape(TimeValue t,BezierShape& ashape) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to build the shape at the specified time and store the results in **ashape**.

**Parameters:**

**TimeValue t**

The time to build the shape.

**BezierShape& ashape**

The created shape is store here.

**Prototype:**

**virtual RefTargetHandle Clone(RemapDir& remap = NoRemap()) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to have the plug-in clone itself. The plug-in should clone

all its references as well.

**Parameters:**

**RemapDir &remap = NoRemap()**

This class is used for remapping references during a Clone. See [Class RemapDir](#).

**Return Value:**

A pointer to the cloned item.

**Prototype:**

**virtual void EndEditParams(IObjParam \*ip, ULONG flags, Animatable \*next) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called when the user is finished editing object's parameters. The system passes a flag into the **EndEditParams()** method to indicate if the rollup page should be removed. If this flag is TRUE, the plug-in must unregister the rollup page, and delete it from the panel.

**Parameters:**

**IObjParam \*ip**

This is an interface pointer passed in. The developer may use the interface pointer to call methods such as **DeleteRollupPage()**.

**ULONG flags**

The following flag may be set:

**END\_EDIT\_REMOVEUI**

If TRUE, the item's user interface should be removed.

**Animatable \*next**

This parameter may be used in the motion and hierarchy branches of the command panel. This pointer allows a plug-in to look at the ClassID of the next item that was being edited, and if it is the same as this item, to not replace the entire UI in the command panel. Note that for items that are edited in the modifier branch this field can be ignored.

**Prototype:**

**virtual CreateMouseButton\* GetCreateMouseButton() = 0;**

**Remarks:**

Implemented by the Plug-In.

This method allows the system to retrieve a callback object used in creating the shape in the 3D viewports. This method returns a pointer to an instance of a class derived from **CreateMouseButton**. This class has a method **proc()** which is where the developer defines the user/mouse interaction used during the shape creation phase.

**Return Value:**

A pointer to an instance of a class derived from [CreateMouseButton](#).

**Prototype:**

**virtual BOOL ValidForDisplay(TimeValue t) = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if it is okay to display the shape at the time passed; otherwise FALSE. Certain shapes may not want to be displayed at a certain time, for example if their size goes to zero at some point.

**Parameters:**

**TimeValue t**

The time to check.

**Prototype:**

**virtual void InvalidateUI();**

**Remarks:**

Implemented by the Plug-In.

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

If the plug-in uses a parameter map for handling its UI, it may call a method of the parameter map to handle this: **ipmapParam->Invalidate();**

If the plug-in does not use parameter maps, it should call the **SetValue()** method on each of its controls that display a value, for example the spinner

controls. This will cause the control to update the value displayed. The code below shows how this may be done for a spinner control. Note that **ip** and **pblock** are assumed to be initialized interface and parameter block pointers (**IObjParam \*ip, IParamBlock \*pblock**).

```
float newval;  
Interval valid=FOREVER;  
TimeValue t=ip->GetTime();  
// Get the value from the parameter block at the current time.  
pblock->GetValue( PB_ANGLE, t, newval, valid );  
// Set the value. Note that the notify argument is passed as  
FALSE.  
// This ensures no messages are sent when the value changes.  
angleSpin->SetValue( newval, FALSE );
```

**Prototype:**

```
virtual ParamDimension *GetParameterDim(int pbIndex)
```

**Remarks:**

Implemented by the Plug-In.

This method returns the parameter dimension of the parameter whose index is passed.

**Parameters:**

```
int pbIndex
```

The index of the parameter to return the dimension of.

**Return Value:**

Pointer to a ParamDimension. See [Class ParamDimension](#).

**Default Implementation:**

```
{return defaultDim;}
```

**Prototype:**

```
virtual TSTR GetParameterName(int pbIndex)
```

**Remarks:**

Implemented by the Plug-In.

Returns the name of the parameter whose index is passed.

**Parameters:**

**int pbIndex**

The index into the parameter block of the parameter to return the name of.

**Default Implementation:**

```
{return TSTR(_T("Parameter"));} 
```

**Prototype:**

```
virtual BOOL DisplayVertTicksDuringCreation();
```

**Remarks:**

Returns TRUE if the Simple Spline should display vertex ticks during its creation; otherwise FALSE.

**Default Implementation:**

```
{ return TRUE; }
```

## Class SimpleShape

See Also: [Class ShapeObject](#), [Class IParamBlock](#), [Class PolyShape](#), [Class Interval](#), [Working with Shapes and Splines](#).

```
class SimpleShape : public ShapeObject
```

### Description:

This class defines a simple shape object to make procedural shape primitives easier to create. For example, the 3ds max Helix plug-in is derived from this class. There are a set of mandatory and optional methods to implement.

Revised for 3ds max 2.0 SimpleShape-based objects have a new 'General' rollup, which contains renderable shape options: Renderable checkbox, Thickness spinner, and a Mapping coords checkbox. These are supported automatically. To support the new features of the renderable splines, the derived class of SimpleShape needs to work with a few new methods -- see **SimpleShapeClone()** and **ReadyGeneralParameters()** below for details.

### Data Members:

public:

**IParamBlock \*pblock;**

The parameter block for managing the shape's parameters.

**static IObjParam \*ip;**

This data member is available in release 2.0 and later only.

This is the interface pointer stored by the class.

**static HWND hGenParams;**

This data member is available in release 2.0 and later only.

The window handle to the 'General' rollup.

**static BOOL dlgRenderable;**

This data member is available in release 2.0 and later only.

The 'Renderable' flag in the 'General' rollup.

**static float dlgThickness;**

This data member is available in release 2.0 and later only.

The 'Thickness' setting in the 'General' rollup.

**static BOOL dlgGenUVs;**

This data member is available in release 2.0 and later only.  
The 'Generate Mapping Coords' flag in the 'General' rollup.

**PolyShape shape;**

The shape cache.

**Interval ivalid;**

The validity interval for the shape cache.

**BOOL suspendSnap;**

A flag to suspend snapping used during the creation process.

**static SimpleShape \*editOb;**

The shape that is currently being edited in the command panel.

**static ISpinnerControl \*thickSpin;**

Points to the spinner control used for the thickness parameter.

**Methods:**

**Prototype:**

**SimpleShape();**

**Remarks:**

Constructor.

**Prototype:**

**~SimpleShape();**

**Remarks:**

Destructor.

Clients of SimpleShape need to implement these methods:

**Prototype:**

**virtual void BuildShape(TimeValue t, PolyShape& ashape) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to build the shape at the specified time and store the result into the PolyShape passed.

**Parameters:**

**TimeValue t**

The time to build the shape.

**PolyShape& ashape**

The built shape is stored here.

**Prototype:**

**virtual BOOL ValidForDisplay(TimeValue t) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method indicates if the shape may be displayed at the time passed. At certain times, for certain shapes, the shape may not be in a displayable form. For example, the size of the shape may go to zero at a certain point and would be inappropriate to display.

**Parameters:****TimeValue t**

The time to check.

**Return Value:**

TRUE if the shape may be displayed at the specified time; otherwise FALSE.

**Prototype:**

**virtual void InvalidateUI()**

**Remarks:**

Implemented by the Plug-In.

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

If the plug-in uses a parameter map for handling its UI, it may call a method of the parameter map to handle this: **ipmapParam->Invalidate();**

If the plug-in does not use parameter maps, it should call the **SetValue()** method on each of its controls that display a value, for example the spinner controls. This will cause to the control to update the value displayed. The code below shows how this may be done for a spinner control. Note that **ip** and **pblock** are assumed to be initialized interface and parameter block pointers

```
(IObjParam *ip, IParamBlock *pblock).  
float newval;  
Interval valid=FOREVER;  
TimeValue t=ip->GetTime();  
// Get the value from the parameter block at the current time.  
pblock->GetValue( PB_ANGLE, t, newval, valid );  
// Set the value. Note that the notify argument is passed as  
FALSE.  
// This ensures no messages are sent when the value changes.  
angleSpin->SetValue( newval, FALSE );
```

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual ParamDimension *GetParameterDim(int pbIndex)
```

**Remarks:**

Implemented by the Plug-In.

This method returns the parameter dimension of the parameter whose index is passed.

**Parameters:**

```
int pbIndex
```

The index of the parameter to return the dimension of.

**Return Value:**

Pointer to a ParamDimension. See [Class ParamDimension](#).

**Default Implementation:**

```
{return defaultDim;}
```

**Prototype:**

```
virtual TSTR GetParameterName(int pbIndex)
```

**Remarks:**

Implemented by the Plug-In.

Returns the name of the parameter whose index is passed.

**Parameters:****int pbIndex**

The index into the parameter block of the parameter to return the name of.

**Default Implementation:**

```
{return TSTR(_T("Parameter"));} 
```

**For procedural shapes the following methods must be implemented:**

**Prototype:**

```
virtual Point3 InterpCurve3D(TimeValue t, int curve, float param,  
int ptype = PARAM_SIMPLE) = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method returns a point interpolated on the entire curve.

**Parameters:****TimeValue t**

The time to evaluate.

**int curve**

The index of the curve to evaluate.

**float param**

The 'distance' along the curve where 0 is the start and 1 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The interpolated point on the curve.

**Prototype:**

```
virtual Point3 TangentCurve3D(TimeValue t, int curve, float  
param, int ptype = PARAM_SIMPLE) = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method returns a tangent vector interpolated on the entire curve.

**Parameters:**

**TimeValue t**

The time at which to evaluate the curve.

**int curve**

The index of the curve to evaluate.

**float param**

The 'distance' along the curve where 0.0 is the start and 1.0 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The tangent vector.

**Prototype:**

**virtual float LengthOfCurve(TimeValue t, int curve) = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the length of the specified curve.

**Parameters:**

**TimeValue t**

The time at which to compute the length.

**int curve**

The index of the curve.

**The following methods are optional.**

You should strongly consider implementing these, because the default implementations just do the bare-minimum job. It is often best to break up a curve into several smaller pieces than to have one single long curve. For example, the user may use a step setting of 10, and will expect to see a reasonable approximation of the shape using such a step setting. A curve that is

too long will not be accurately represented by such a steps setting as it will be too course. Chopping your curve up into manageable pieces will make things look better.

**Prototype:**

**virtual int NumberOfPieces(TimeValue t, int curve)**

**Remarks:**

Implemented by the Plug-In.

Returns the number of sub-curves in a curve.

**Parameters:**

**TimeValue t**

The time at which to check.

**int curve**

The index of the curve.

**Default Implementation:**

```
{ return 1; }
```

**Prototype:**

**virtual Point3 InterpPiece3D(TimeValue t, int curve, int piece,  
float param, int ptype=PARAM\_SIMPLE)**

**Remarks:**

Implemented by the Plug-In.

This method returns the interpolated point along the specified sub-curve.

**Parameters:**

**TimeValue t**

The time to evaluate the sub-curve.

**int curve**

The curve to evaluate.

**int piece**

The sub-curve (segment) to evaluate.

**float param**

The position along the sub-curve to return where 0.0 is the start and 1.0 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The point in world space.

**Default Implementation:**

```
{ return InterpCurve3D(t, curve, param); }
```

**Prototype:**

```
virtual Point3 TangentPiece3D(TimeValue t, int curve,  
    int piece, float param, int ptype=PARAM_SIMPLE)
```

**Remarks:**

Implemented by the Plug-In.

Returns the tangent vector on a sub-curve at the specified 'distance' along the curve.

**Parameters:**

**TimeValue t**

The time to evaluate the sub-curve.

**int curve**

The curve to evaluate.

**int piece**

The sub-curve (segment) to evaluate.

**float param**

The position along the sub-curve to return where 0.0 is the start and 1.0 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The tangent vector.

**Default Implementation:**

```
{ return TangentCurve3D(t, curve, param, ptype); }
```

**Prototype:****void ReadyGeneralParameters();****Remarks:**

This method is available in release 2.0 and later only.

To support the new features of the renderable splines, in the derived class's constructor, call ReadyGeneralParameters(). This will set up the general parameters in the base class to the proper defaults. Failure to make this call will cause SimpleShape-based objects to be created with default general parameters rather than those of the previously-created object.

**Prototype:****void SimpleShapeClone(SimpleShape \*sshpSource);****Remarks:**

This method is available in release 2.0 and later only.

To support the new features of the renderable splines, the derived class of SimpleShape needs to, in the Clone method, call this method. This will insure that the base class parameters are copied to the cloned object. Failure to make this call will cause cloned SimpleShape-based objects to revert to the default rendering parameters.

**Parameters:****SimpleShape \*sshpSource**

The source shape for the clone.

**Prototype:****virtual MtlID GetMatID(TimeValue t, int curve, int piece);****Remarks:**

This method is available in release 3.0 and later only.

Returns the material ID of the specified segment of the specified curve or the shape.

**Parameters:****TimeValue t**

The time at which to return the material ID

**int curve**

The zero based index of the curve.

**int piece**

The zero based index of the segment of the curve.

## Class LightObject

See Also: [Class Object](#), [Class ObjLightDesc](#), [Class Interval](#), [Class Texmap](#).

```
class LightObject : public Object
```

### Description:

This is the base class from which plug-in lights may be derived.

### Methods:

#### Prototype:

```
virtual RefResult EvalLightState(TimeValue time, Interval&  
valid, LightState *ls)=0;
```

#### Remarks:

Implemented by the Plug-In.

This method is called to update the passed **LightState** and validity interval of the light.

#### Parameters:

##### **TimeValue time**

The time to evaluate the light state.

##### **Interval& valid**

The validity interval of the light about the specified time. This interval should be updated to reflect the validity interval of the light.

##### **LightState \*ls**

A pointer to the **LightState** structure which describes the properties of the light. This function updates the data in the structure to reflect the properties of the light at the specified time. See [Structure LightState](#).

#### Return Value:

**REF\_SUCCEED** if the **LightState** was updated; otherwise **REF\_FAIL**.

#### Prototype:

```
virtual ObjLightDesc *CreateLightDesc(INode *n, BOOL  
forceShadowBuffer  
)
```

**Remarks:**

Implemented by the Plug-In.

When the renderer goes to render the scene it asks all of the lights to create an **ObjectLightDesc** object. This is the method that is called to return this object.

**Parameters:**

**INode \*n**

The node pointer of the light.

**BOOL forceShadowBuffer**

Forces the creation of a shadow buffer.

**Return Value:**

An instance of **ObjectLightDesc**. See [Class ObjectLightDesc](#).

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual void SetUseLight(int onOff)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets if the light is on or off.

**Parameters:**

**int onOff**

TRUE for on; FALSE for off.

**Prototype:**

```
virtual BOOL GetUseLight()=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the light is on; otherwise FALSE.

**Prototype:**

**virtual void SetHotspot(TimeValue time, float f)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets the hotspot to the specified angle at the specified time.

**Parameters:**

**TimeValue time**

The time to set the hotspot angle.

**float f**

The angle to set in degrees.

**Prototype:**

**virtual float GetHotspot(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Implemented by the Plug-In.

Retrieves the hotspot angle.

**Parameters:**

**TimeValue t**

The time to retrieve the angle.

**Interval& valid = Interval(0,0)**

The validity interval that this method will update to reflect the hotspot setting.

**Return Value:**

The hotspot angle (in degrees).

**Prototype:**

**virtual void SetFalloffsize(TimeValue time, float f)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets the falloff setting of the light.

**Parameters:**

**TimeValue time**

The time to set the falloff.

**float f**

The falloff angle in degrees.

**Prototype:**

```
virtual float GetFallsizes(TimeValue t, Interval& valid = Interval(0,0))=0;
```

**Remarks:**

Implemented by the Plug-In.

Returns the falloff angle of the light in radians.

**Parameters:**

**TimeValue t**

The time to retrieve the falloff angle.

**Interval& valid = Interval(0,0)**

The validity interval that this method will update to reflect the falloff setting.

**Return Value:**

The falloff angle of the light in degrees.

**Prototype:**

```
virtual void SetAtten(TimeValue time, int which, float f)=0;
```

**Remarks:**

Implemented by the Plug-In.

Sets the specified attenuation range distance at the time passed.

**Parameters:**

**TimeValue time**

The time to set the attenuation distance.

**int which**

Indicates which distance to set. One of the following values:

**LIGHT\_ATTEN\_START** - The start range radius.

**LIGHT\_ATTEN\_END** - The end range radius.

**float f**

The distance to set.

**Prototype:**

**virtual float GetAtten(TimeValue t, int which, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the specified attenuation range distance at the time passed.

**Parameters:**

**TimeValue t**

The time to retrieve the attenuation distance.

**int which**

Indicates which distance to retrieve. One of the following values:

**LIGHT\_ATTEN\_START** - The start range radius.

**LIGHT\_ATTEN\_END** - The end range radius.

**Interval& valid = Interval(0,0)**

The validity interval that this method will update to reflect the attenuation setting.

**Return Value:**

The specified attenuation range distance.

**Prototype:**

**virtual void SetTDist(TimeValue time, float f)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets the light's target distance.

**Parameters:**

**TimeValue time**

The time to set the distance.

**float f**

The distance to set.

**Prototype:**

**virtual float GetTDist(TimeValue t, Interval& valid =**

**Interval(0,0)=0;**

**Remarks:**

Implemented by the Plug-In.  
Retrieves the light's target distance.

**Parameters:**

**TimeValue t**

The time to retrieve the distance.

**Interval& valid = Interval(0,0)**

The validity interval that this method will update to reflect the target distance setting.

**Return Value:**

The light's target distance.

**Prototype:**

**virtual void SetConeDisplay(int s, int notify=TRUE)=0;**

**Remarks:**

Implemented by the Plug-In.  
Sets the light's cone display flag. This controls if the cone is depicted graphically in the viewports.

**Parameters:**

**int s**

Indicates if the cone display should be on or off. If nonzero, the cone should be displayed; otherwise it should be turned off.

**int notify=TRUE**

If notify is TRUE the plug-in should call **NotifyDependents()** to notify its dependents.

**Prototype:**

**virtual BOOL GetConeDisplay()=0;**

**Remarks:**

Implemented by the Plug-In.  
Retrieves the light's cone display setting. This indicates if the cone is depicted

graphically in the viewports.

**Return Value:**

TRUE to indicate the cone is displayed; FALSE to indicate it is turned off.

**Prototype:**

**virtual int GetShadowMethod();**

**Remarks:**

Implemented by the Plug-In.

Returns the type of shadows used by the light.

**Return Value:**

One of the following values:

**LIGHTSHADOW\_NONE**

**LIGHTSHADOW\_MAPPED**

**LIGHTSHADOW\_RAYTRACED**

**Default Implementation:**

```
{return LIGHTSHADOW_NONE;}
```

**Prototype:**

**virtual void SetRGBColor(TimeValue t, Point3& rgb);**

**Remarks:**

Implemented by the Plug-In.

Sets the color of the light at the specified time.

**Parameters:**

**TimeValue t**

The time to set the color.

**Point3& rgb**

The color to set.

**Prototype:**

**virtual Point3 GetRGBColor(TimeValue t, Interval &valid = Interval(0,0));**

**Remarks:**

Implemented by the Plug-In.

Returns the color of the light at the specified time and updates the validity interval to reflect this parameters validity interval.

**Parameters:****TimeValue t**

The time to retrieve the value.

**Interval &valid = Interval(0,0)**

The validity interval to intersect with this parameters interval.

**Return Value:**

The color of the light at the specified time.

**Default Implementation:**

```
{return Point3(0,0,0);}
```

**Prototype:**

```
virtual void SetIntensity(TimeValue time, float f);
```

**Remarks:**

Implemented by the Plug-In.

Sets the intensity of the light to the value passed.

**Parameters:****TimeValue time**

The time to set the value.

**float f**

The value to set.

**Prototype:**

```
virtual float GetIntensity(TimeValue t, Interval& valid =  
Interval(0,0));
```

**Remarks:**

Implemented by the Plug-In.

Retrieves the intensity of the light at the specified time and updates the validity interval passed to reflect the validity interval of this parameter.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Interval &valid = Interval(0,0)**

The validity interval to intersect with this parameters interval.

**Return Value:**

The intensity of the light at the specified time

**Default Implementation:**

```
{return 0.0f;}
```

**Prototype:**

```
virtual void SetAspect(TimeValue t, float f);
```

**Remarks:**

Implemented by the Plug-In.

Sets the aspect ratio of the light at the specified time.

**Parameters:**

**TimeValue t**

The time to set the value.

**float f**

The value to set.

**Prototype:**

```
virtual float GetAspect(TimeValue t, Interval& valid =  
Interval(0,0));
```

**Remarks:**

Implemented by the Plug-In.

Retrieves the aspect ratio of the light at the specified time and updates the validity interval passed to reflect the validity interval of this parameter.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Interval &valid = Interval(0,0)**

The validity interval to intersect with this parameters interval.

**Return Value:**

The aspect ratio of the light at the specified time

**Default Implementation:**

```
{return 0.0f;}
```

**Prototype:**

```
virtual void SetUseAtten(int s);
```

**Remarks:**

Implemented by the Plug-In.

Sets the flag to indicate if the light is attenuated.

**Parameters:**

**int s**

Nonzero to indicate the light is attenuated; otherwise 0.

**Prototype:**

```
virtual BOOL GetUseAtten();
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE to indicate the light is attenuated; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void SetAttenDisplay(int s);
```

**Remarks:**

Implemented by the Plug-In.

Sets the flag to indicate if the light attenuation ranges are displayed.

**Parameters:**

**int s**

Nonzero to indicate the light attenuation ranges are displayed; otherwise 0.

**Prototype:**

**virtual BOOL GetAttenDisplay();**

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the light attenuation ranges are displayed; otherwise FALSE.

**Default Implementation:**

**{return FALSE;}**

**Prototype:**

**virtual void Enable(int enab);**

**Remarks:**

Implemented by the Plug-In.

Sets the light to enabled or disables (on or off).

**Parameters:**

**int enab**

Nonzero to set the light to on; zero to turn the light off.

**Prototype:**

**virtual void SetMapBias(TimeValue t, float f);**

**Remarks:**

Implemented by the Plug-In.

Sets the map bias setting at the time passed.

**Parameters:**

**TimeValue t**

The time to set the value.

**float f**

The map bias value to set. The 3ds max lights use a range of 0.0 to 100.0.

**Prototype:**

**virtual float GetMapBias(TimeValue t, Interval& valid = Interval(0,0));**

**Remarks:**

Implemented by the Plug-In.

Returns the map bias setting at the time passed and updates the validity interval to reflect the validity interval of this parameter.

**Parameters:****TimeValue t**

The time to retrieve the value.

**Interval& valid = Interval(0,0)**

The validity interval to update to reflect this parameters validity interval.

**Return Value:**

The map bias setting at the time passed.

**Default Implementation:**

```
{return 0.0f;}
```

**Prototype:**

```
virtual void SetMapRange(TimeValue t, float f);
```

**Remarks:**

Implemented by the Plug-In.

Sets the map sample range setting to the value passed at the time passed.

**Parameters:****TimeValue t**

The time to set the value.

**float f**

The value to set. The 3ds max lights use a range of 0.0 to 20.0.

**Prototype:**

```
virtual float GetMapRange(TimeValue t, Interval& valid = Interval(0,0));
```

**Remarks:**

Implemented by the Plug-In.

Retrieves the lights map sample range setting at the specified time and updates the validity interval to reflect the validity interval of this parameter.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Interval& valid = Interval(0,0)**

The validity interval to update to reflect this parameters validity interval.

**Return Value:**

The lights map sample range setting.

**Default Implementation:**

```
{return 0.0f;}
```

**Prototype:**

```
virtual void SetMapSize(TimeValue t, int f);
```

**Remarks:**

Implemented by the Plug-In.

Sets the lights map size parameter to the value passed at the time passed.

**Parameters:**

**TimeValue t**

The time to set the value.

**int f**

The value to set.

**Prototype:**

```
virtual int GetMapSize(TimeValue t, Interval& valid =  
Interval(0,0))
```

**Remarks:**

Implemented by the Plug-In.

Returns the lights map size parameter at the specified time and updates the validity interval passed to reflect the validity interval of this parameter.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Interval& valid = Interval(0,0)**

The validity interval to update to reflect this parameters validity interval.

**Return Value:**

The lights map size parameter.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void SetRayBias(TimeValue t, float f);
```

**Remarks:**

Implemented by the Plug-In.

Sets the raytrace bias setting to the value passed at the specified time.

**Parameters:**

**TimeValue t**

The time to set the value.

**float f**

The value to set.

**Prototype:**

```
virtual float GetRayBias(TimeValue t, Interval& valid =  
Interval(0,0));
```

**Remarks:**

Implemented by the Plug-In.

Returns the lights raytrace bias setting at the specified time and updates the validity interval passed to reflect the validity interval of this parameter.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Interval& valid = Interval(0,0)**

The validity interval to update to reflect this parameters validity interval.

**Return Value:**

The lights raytrace bias setting at the specified time.

**Default Implementation:**

```
{return 0.0f;}
```

**Prototype:**

```
virtual int GetUseGlobal()
```

**Remarks:**

Implemented by the Plug-In.

Returns the Use Global Settings flag setting.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void SetUseGlobal(int a);
```

**Remarks:**

Implemented by the Plug-In.

Sets the lights Use Global Settings flag.

**Parameters:**

**int a**

Nonzero indicates the light uses the global settings; zero indicates the light uses its own settings.

**Prototype:**

```
virtual int GetShadow()
```

**Remarks:**

Implemented by the Plug-In.

Returns the lights Cast Shadows flag.

**Return Value:**

Nonzero indicates the light casts shadows; otherwise 0.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

**virtual void SetShadow(int a);**

**Remarks:**

Implemented by the Plug-In.

Sets the lights Cast Shadows flag.

**Parameters:**

**int a**

Nonzero indicates the light casts shadows; zero indicates the light does not cast shadows.

**Prototype:**

**virtual int GetShadowType()**

**Remarks:**

Implemented by the Plug-In.

Retrieves the type of shadows used by the light - mapped or raytraced.

**Return Value:**

One of the following values:

**-1:** if the Shadow Generator is NULL. (R3 only).

**0:** if the light uses Shadow Maps.

**1:** if the light uses Raytraced Shadows.

**0xffff:** for any other Shadow Generators. (R3 only).

**Default Implementation:**

**{return 0;}**

**Prototype:**

**virtual void SetShadowType(int a);**

**Remarks:**

Implemented by the Plug-In.

Sets the type of shadows used by the light - mapped or raytraced.

**Parameters:**

**int a**

One of the following values:

**0:** This value plugs in a Shadow Map Generator  
**1:** This value plugs in a Raytraced Shadow Generator.  
Any other value is a NOOP.

**Prototype:**

**virtual void SetShadowGenerator(ShadowType \*s);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the shadow generator used by the light.

**Parameters:**

**ShadowType \*s**

The shadow plug-in to use. See [Class ShadowType](#).

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual ShadowType \*GetShadowGenerator();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns a pointer to the shadow generator plug-in in use by the light. See [Class ShadowType](#).

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

**virtual int GetAbsMapBias()**

**Remarks:**

Implemented by the Plug-In.  
Returns the lights Absolute Map Bias setting.

**Return Value:**

Nonzero indicates Absolute Map Bias is on; zero indicates it is off.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void SetAbsMapBias(int a);
```

**Remarks:**

Implemented by the Plug-In.

Sets the lights Absolute Map Bias setting.

**Parameters:**

**int a**

Nonzero indicates Absolute Map Bias is on; zero indicates it is off.

**Prototype:**

```
void SetAtmosShadows(TimeValue t, int onOff);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the state of the atmospheric shadowing shadows on / off toggle for the light.

**Parameters:**

**TimeValue t**

The time at which to set the state.

**int onOff**

Nonzero for on; zero for off.

**Prototype:**

```
int GetAtmosShadows(TimeValue t);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the state of the atmospheric shadowing shadows on / off toggle for the light.

**Parameters:**

**TimeValue t**

The time at which to retrieve the state.

**Prototype:**

**void SetAtmosOpacity(TimeValue t, float f);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the atmospheric shadowing opacity for the light to the specified value at the time specified.

**Parameters:**

**TimeValue t**

The time at which to set the value.

**float f**

The value to set.

**Prototype:**

**float GetAtmosOpacity(TimeValue t, Interval& valid=FOREVER);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the atmospheric shadowing opacity for the light at the specified time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the value.

**Interval& valid=FOREVER**

The validity interval which is updated to reflect the validity of the opacity setting.

**Prototype:**

**void SetAtmosColAmt(TimeValue t, float f);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the atmospheric shadowing color amount to the specified value at the

time passed.

**Parameters:**

**TimeValue t**

The time at which to set the value.

**float f**

The value to set.

**Prototype:**

**float GetAtmosColAmt(TimeValue t, Interval& valid=FOREVER);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the atmospheric shadowing color amount setting at the specified time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the value.

**Interval& valid=FOREVER**

The validity interval which is updated to reflect the validity of the color amount setting.

**Prototype:**

**virtual int GetOvershoot()**

**Remarks:**

Implemented by the Plug-In.

Returns the lights Overshoot on / off setting. Nonzero indicates overshoot is on; otherwise 0.

**Default Implementation:**

**{return 0;}**

**Prototype:**

**virtual void SetOvershoot(int a);**

**Remarks:**

Implemented by the Plug-In.

Sets the lights Overshoot on / off setting.

**Parameters:**

**int a**

Nonzero indicates overshoot is on; otherwise 0.

**Prototype:**

**virtual int GetProjector()**

**Remarks:**

Implemented by the Plug-In.

Returns the lights Projector on / off setting. Nonzero indicates this light projects an image; otherwise 0.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

**virtual void SetProjector(int a);**

**Remarks:**

Implemented by the Plug-In.

Sets the lights projector on / off setting.

**Parameters:**

**int a**

Nonzero indicates this light projects an image; otherwise 0.

**Prototype:**

**virtual ExclList\* GetExclList();**

**Remarks:**

Implemented by the Plug-In.

Returns the list of names of items included or excluded by this light. See [Class NameTab](#).

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual BOOL Include();
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the light's name list is of items to be included by the light.

Returns FALSE if the list is of items to exclude from the light.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual Texmap* GetProjMap()
```

**Remarks:**

Implemented by the Plug-In.

Returns the map used by a projector light.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual void SetProjMap(Texmap* pmap);
```

**Remarks:**

Implemented by the Plug-In.

Sets the image(s) used by the projector light.

**Parameters:**

**Texmap\* pmap**

The map to use.

**Prototype:**

```
virtual void UpdateTargDistance(TimeValue t, INode* inode);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Updates the display of the light's target distance in the light's rollup page.

**Parameters:**

**TimeValue t**

The time to retrieve the distance.

**INode\* inode**

The light node.

**Default Implementation:**

```
{
```

## Class GenLight

See Also: [Class LightObject](#), [Class NameTab](#), [Class Control](#), [Class Interval](#), [Class Point3](#), [Class ObjLightDesc](#).

```
class GenLight : public LightObject
```

### Description:

This class describes a generic light object. It is used as a base class for creating plug-in lights. Methods of this class are used to get and set properties of the light. All methods of this class are implemented by the plug-in.

### Methods:

#### Prototype:

```
virtual GenLight *NewLight(int type)=0;
```

#### Remarks:

Creates a new light object of the specified type.

#### Parameters:

**int type**

One of the following values:

**OMNI\_LIGHT** -- Omnidirectional light.

**TSPOT\_LIGHT** -- Targeted spot light.

**DIR\_LIGHT** -- Directional light.

**FSPOT\_LIGHT** -- Free spot light.

**TDIR\_LIGHT** -- Targeted directional light.

#### Return Value:

A pointer to a new instance of the specified light type.

#### Prototype:

```
virtual int Type()=0;
```

#### Remarks:

This method is available in release 2.0 and later only.

Returns the type of light this is.

#### Return Value:

One of the following values:

**OMNI\_LIGHT** -- Omnidirectional light.

**TSPOT\_LIGHT** -- Targeted spot light.

**DIR\_LIGHT** -- Directional light.

**FSPOT\_LIGHT** -- Free spot light.

**TDIR\_LIGHT** -- Targeted directional light.

**Prototype:**

**virtual BOOL IsSpot()=0;**

**Remarks:**

Returns TRUE if the light is a spotlight; otherwise FALSE.

**Prototype:**

**virtual BOOL IsDir()=0;**

**Remarks:**

Returns TRUE if the light is directional; otherwise FALSE.

**Prototype:**

**virtual void SetUseLight(int onOff)=0;**

**Remarks:**

Sets the light on or off.

**Parameters:**

**int onOff**

Nonzero sets the light on; zero sets it off.

**Prototype:**

**virtual BOOL GetUseLight(void)=0;**

**Remarks:**

Returns TRUE if the light is on; otherwise FALSE.

**Prototype:**

**virtual void SetSpotShape(int s)=0;**

**Remarks:**

Sets the shape used for a spotlight, either rectangular or circular.

**Parameters:**

**int s**

One of the following values:

**RECT\_LIGHT**

**CIRCLE\_LIGHT**

**Prototype:**

**virtual int GetSpotShape(void)=0;**

**Remarks:**

Retrieves the shape used for a spotlight.

**Return Value:**

One of the following values:

**RECT\_LIGHT**

**CIRCLE\_LIGHT**

**Prototype:**

**virtual void SetHotspot(TimeValue time, float f)=0;**

**Remarks:**

Sets the hotspot to the specified angle at the specified time.

**Parameters:**

**TimeValue time**

The time to set the value.

**float f**

The angle in degrees.

**Prototype:**

**virtual float GetHotspot(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Returns the hotspot angle in degrees at the specified time and updates the interval to reflect the validity of the hotspot controller.

**Parameters:**

**TimeValue t**

The time to get the angle.

**Interval& valid = Interval(0,0)**

The interval which is updated.

**Prototype:**

**virtual void SetFalloffsize(TimeValue time, float f)=0;**

**Remarks:**

Sets the falloff angle at the specified time.

**Parameters:**

**TimeValue time**

The time to set the angle.

**float f**

The angle to set in degrees.

**Prototype:**

**virtual float GetFalloffsize(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Returns the falloff angle in degrees at the specified time and updates the interval passed to reflect the validity of the falloff controller.

**Parameters:**

**TimeValue t**

The time to return the value.

**Interval& valid = Interval(0,0)**

The interval which is updated.

**Prototype:**

**virtual void SetAtten(TimeValue time, int which, float f)=0;**

**Remarks:**

Sets the specified attenuation range distance at the time passed.

**Parameters:****TimeValue time**

The time to set the attenuation distance.

**int which**

Indicates which distance to set. One of the following values:

**ATTEN1\_START**

The near start range.

**ATTEN1\_END**

The near end range.

**ATTEN\_START**

The far start range.

**ATTEN\_END**

The far end range.

**float f**

The distance to set.

**Prototype:**

```
virtual float GetAtten(TimeValue t, int which, Interval& valid =  
Interval(0,0))=0;
```

**Remarks:**

Returns the specified attenuation distance at the time passed and updates the interval to reflect the validity of the attenuation controller.

**Parameters:****TimeValue t**

The time to get the attenuation distance.

**int which**

Indicates which distance to get. One of the following values:

**ATTEN1\_START**

The near start range.

**ATTEN1\_END**

The near end range.

**ATTEN\_START**

The far start range.

**ATTEN\_END**

The far end range.

**Interval& valid = Interval(0,0)**

The interval which is updated.

**Prototype:**

```
virtual void SetTDist(TimeValue time, float f)=0;
```

**Remarks:**

Sets the light's target distance.

**Parameters:****TimeValue time**

The time to set the distance.

**float f**

The distance to set.

**Prototype:**

```
virtual float GetTDist(TimeValue t, Interval& valid =  
Interval(0,0))=0;
```

**Remarks:**

Returns the light's target distance at the specified time and updates the interval passed to reflect the validity of the target distance.

**Parameters:****TimeValue t**

The time to retrieve the distance.

**Interval& valid = Interval(0,0)**

The interval to update.

**Prototype:**

```
virtual ObjLightDesc *CreateLightDesc(INode *n,BOOL  
forceShadowBuffer=false)=0;
```

**Remarks:**

When the renderer goes to render the scene it asks all of the lights to create an ObjLighDesc object. This is the method that is called to return a pointer to this object.

**Parameters:****INode \*n**

The node pointer of the light.

**BOOL forceShadowBuffer**

Forces the creation of a shadow buffer.

**Prototype:****virtual void SetRGBColor(TimeValue t, Point3& rgb)=0;****Remarks:**

Sets the color of the light at the specified time.

**Parameters:****TimeValue t**

The time to set the color.

**Point3& rgb**

The color to set.

**Prototype:****virtual Point3 GetRGBColor(TimeValue t, Interval &valid = Interval(0,0))=0;****Remarks:**

Returns the color of the light at the specified time and updates the validity interval to reflect this parameters validity interval.

**Parameters:****TimeValue t**

The time to get the color.

**Interval &valid = Interval(0,0)**

The interval which is updated.

**Prototype:**

```
virtual void SetHSVColor(TimeValue t, Point3& hsv)=0;
```

**Remarks:**

Sets the HSV color of the light at the specified time.

**Parameters:****TimeValue t**

The time to set the color.

**Point3& hsv**

The color.

**Sample Code:**

The following sample shows how the RGB value can be converted to HSV.

```
{  
  int h, s, v;  
  Point3 rgbf = GetRGBColor(t, valid);  
  DWORD rgb = RGB((int)(rgbf[0]*255.0f),  
    (int)(rgbf[1]*255.0f), (int)(rgbf[2]*255.0f));  
  RGBtoHSV (rgb, &h, &s, &v);  
  return Point3(h/255.0f, s/255.0f, v/255.0f);  
}
```

**Prototype:**

```
virtual Point3 GetHSVColor(TimeValue t, Interval &valid =  
  Interval(0,0))=0;
```

**Remarks:**

Retrieves the HSV color of the light at the specified time and updates the validity interval to reflect the color parameter.

**Parameters:****TimeValue t**

The time to retrieve the color.

**Interval &valid = Interval(0,0)**

The interval to update.

**Return Value:**

The color of the light (as a Point3).

**Prototype:**

**virtual void SetIntensity(TimeValue time, float f)=0;**

**Remarks:**

Sets the intensity (multiplier value) of the light at the specified time.

**Parameters:**

**TimeValue time**

The time to set the intensity.

**float f**

The value to set.

**Prototype:**

**virtual float GetIntensity(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Returns the intensity (multiplier value) of the light at the specified time and updates the interval passed to reflect the validity of the controller.

**Parameters:**

**TimeValue t**

The time to get the value.

**Interval& valid = Interval(0,0)**

The interval is updated.

**Prototype:**

**virtual void SetContrast(TimeValue time, float f)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the light's contrast setting.

**Parameters:**

**TimeValue time**

The time to set the contrast value.

**float f**

The new contrast value in the range of 0.0 to 100.0.

**Prototype:**

**virtual float GetContrast(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the light's contrast setting in the range 0.0 to 100.0.

**Parameters:**

**TimeValue t**

The time to get the light's contrast setting.

**Interval& valid = Interval(0,0)**

This interval is updated to reflect the interval of the light's contrast setting.

**Prototype:**

**virtual void SetAspect(TimeValue t, float f)=0;**

**Remarks:**

Set the aspect property to the specified value.

**Parameters:**

**TimeValue t**

The time at which to set the aspect ratio.

**float f**

Specifies the aspect ratio setting.

**Prototype:**

**virtual float GetAspect(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Returns the aspect property (for rectangular lights) at the specified time and updates the interval passed to reflect the validity of the aspect controller.

**Parameters:**

**TimeValue t**

The time to get the value.

**Interval& valid = Interval(0,0)**

The interval to update.

**Prototype:**

```
virtual void SetConeDisplay(int s, int notify=TRUE)=0;
```

**Remarks:**

Sets the spotlight cone display to on or off. This controls if the cone is depicted graphically in the viewports.

**Parameters:****int s**

Indicates if the cone display should be on or off. Nonzero indicates the cone should be displayed; otherwise it will be turned off.

**int notify=TRUE**

If notify is TRUE the plug-in should call **NotifyDependents()** to notify its dependents.

**Prototype:**

```
virtual BOOL GetConeDisplay()=0;
```

**Remarks:**

Returns the cone display property. TRUE if the spotlight cone is on; FALSE if off.

**Prototype:**

```
virtual void SetUseAtten(int s)=0;
```

**Remarks:**

Sets the far attenuation state to on or off.

**Parameters:****int s**

Nonzero for on; zero for off.

**Prototype:**

**virtual BOOL GetUseAtten()=0;**

**Remarks:**

Returns nonzero if far attenuation is on; zero if off.

**Prototype:**

**virtual void SetAttenDisplay(int s)=0;**

**Remarks:**

Establishes if the light far attenuation range is displayed in the viewports.

**Parameters:**

**int s**

Nonzero for on; zero for off.

**Prototype:**

**virtual BOOL GetAttenDisplay()=0;**

**Remarks:**

Returns TRUE if the far attenuation range is displayed; otherwise FALSE.

**Prototype:**

**virtual void SetUseAttenNear(int s)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets if the light uses near attenuation.

**Parameters:**

**int s**

Nonzero to use near attenuation; otherwise zero.

**Prototype:**

**virtual BOOL GetUseAttenNear(void)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the light has near attenuation on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetAttenNearDisplay(void)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the light near attenuation range is displayed in the viewports; otherwise FALSE.

**Prototype:**

**virtual void SetAttenNearDisplay(int s)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Establishes if the light near attenuation range is displayed in the viewports.

**Parameters:**

**int s**

TRUE to turn on the display; otherwise FALSE.

**Prototype:**

**virtual void Enable(int enab)=0;**

**Remarks:**

Sets the light to enabled or disables (on or off).

**Parameters:**

**int enab**

Nonzero for on; zero for off.

**Prototype:**

**virtual void SetMapBias(TimeValue t, float f)=0;**

**Remarks:**

Sets the map bias value at the time passed.

**Parameters:**

**TimeValue t**

The time to set the value.

**float f**

The value to set.

**Prototype:**

```
virtual float GetMapBias(TimeValue t, Interval& valid =  
Interval(0,0))=0;
```

**Remarks:**

Returns the map bias setting at the specified time and updates the interval passed to reflect the validity of the map bias.

**Parameters:****TimeValue t**

The time to get the value.

**Interval& valid = Interval(0,0)**

The interval to update.

**Prototype:**

```
virtual void SetMapRange(TimeValue t, float f)=0;
```

**Remarks:**

Sets the map range value at the time passed.

**Parameters:****TimeValue t**

The time to set the value.

**float f**

The value to set.

**Prototype:**

```
virtual float GetMapRange(TimeValue t, Interval& valid =  
Interval(0,0))=0;
```

**Remarks:**

Returns the map range setting at the specified time and updates the interval

passed to reflect the validity of the map range.

**Parameters:**

**TimeValue t**

The time to get the value.

**Interval& valid = Interval(0,0)**

The interval to update.

**Prototype:**

**virtual void SetMapSize(TimeValue t, int f)=0;**

**Remarks:**

Sets the map size value at the time passed.

**Parameters:**

**TimeValue t**

The time to set the value.

**int f**

The value to set.

**Prototype:**

**virtual int GetMapSize(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Returns the map size setting at the specified time and updates the interval passed to reflect the validity of the map size.

**Parameters:**

**TimeValue t**

The time to get the value.

**Interval& valid = Interval(0,0)**

The interval to update.

**Prototype:**

**virtual void SetRayBias(TimeValue t, float f)=0;**

**Remarks:**

Sets the map raytrace bias value at the time passed.

**Parameters:**

**TimeValue t**

The time to set the value.

**float f**

The value to set.

**Prototype:**

**virtual float GetRayBias(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Returns the raytrace bias setting at the specified time and updates the interval passed to reflect the validity of the bias.

**Parameters:**

**TimeValue t**

The time to get the value.

**Interval& valid = Interval(0,0)**

The interval to update.

**Prototype:**

**virtual int GetUseGlobal()=0;**

**Remarks:**

Returns TRUE if the use global setting is on; otherwise FALSE.

**Prototype:**

**virtual void SetUseGlobal(int a)=0;**

**Remarks:**

Set the use global setting to on or off.

**Parameters:**

**int a**

TRUE for on; FALSE for off.

**Prototype:**

```
virtual int GetShadow();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the lights Cast Shadows flag. Nonzero indicates the light casts shadows; otherwise 0.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void SetShadow(int a);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the lights Cast Shadows flag.

**Parameters:**

**int a**

Nonzero indicates the light casts shadows; zero indicates the light does not cast shadows.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual int GetShadowType();
```

**Remarks:**

This method is available in release 3.0 and later only.

Retrieves the type of shadows used by the light - mapped or raytraced.

**Return Value:**

One of the following values:

**-1:** if the Shadow Generator is NULL. (R3 only).

**0:** if the light uses Shadow Maps.

**1:** if the light uses Raytraced Shadows.

**0xffff**: for any other Shadow Generators. (R3 only).

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void SetShadowType(int a);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the type of shadows used by the light - mapped or raytraced.

**Parameters:**

**int a**

The shadow type. One of the following values:

**0**: This value plugs in a Shadow Map Generator.

**1**: This value plugs in a Raytraced Shadow Generator.

Any other value is a NO-OP.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void SetShadowGenerator(ShadowType *s);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the shadow generator used by the light.

**Parameters:**

**ShadowType \*s**

The shadow plug-in to use. See [Class ShadowType](#).

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual ShadowType *GetShadowGenerator();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the shadow generator plug-in in use by the light. See [Class ShadowType](#).

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void SetAtmosShadows(TimeValue t, int onOff);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the atmospheric shadow flag to on or off at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the value.

**int onOff**

TRUE for on; FALSE for off.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int GetAtmosShadows(TimeValue t);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the atmospheric shadow setting at the specified time.

**Parameters:**

**TimeValue t**

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

**virtual void SetAtmosOpacity(TimeValue t, float f);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the atmospheric opacity value at the time passed.

**Parameters:**

**TimeValue t**

The time to set the value.

**float f**

The value to set.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual float GetAtmosOpacity(TimeValue t, Interval& valid=FOREVER);**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the atmospheric opacity value at the specified time and updates the validity interval to reflect the validity of the opacity controller.

**Parameters:**

**TimeValue t**

The time to get the value.

**Interval& valid=FOREVER**

The interval to update.

**Default Implementation:**

```
{ return 0.0f; }
```

**Prototype:**

**virtual void SetAtmosColAmt(TimeValue t, float f);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the atmospheric shadow color amount at the specified time.

**Parameters:**

**TimeValue t**

The time to set the value.

**float f**

The value to set.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual float GetAtmosColAmt(TimeValue t, Interval& valid=FOREVER);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the atmospheric shadow color amount at the specified time and updates the interval passed to reflect the validity of the amount.

**Parameters:**

**TimeValue t**

The time to get.

**Interval& valid=FOREVER**

The interval to update.

**Default Implementation:**

```
{ return 0.0f; }
```

**Prototype:**

```
virtual int GetOvershoot()=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the overshoot setting. Nonzero is on; zero is off.

**Prototype:**

**virtual void SetOvershoot(int a)=0;**

**Remarks:**

This method is available in release 2.0 and later only.  
Sets the overshoot setting.

**Parameters:**

**int a**

Nonzero for on; zero for off.

**Prototype:**

**virtual ExclList& GetExclusionList()=0;**

**Remarks:**

Returns the exclusion list for the light.

**Prototype:**

**virtual void SetExclusionList(ExclList &list)=0;**

**Remarks:**

Sets the exclusion list for the light.

**Parameters:**

**ExclList &list**

The exclusion list.

**Prototype:**

**virtual BOOL SetHotSpotControl(Control \*c)=0;**

**Remarks:**

Sets the controller for the hot spot parameter.

**Parameters:**

**Control \*c**

The controller to set.

**Return Value:**

TRUE if the controller was set; otherwise FALSE.

**Prototype:**

**virtual BOOL SetFalloffControl(Control \*c)=0;**

**Remarks:**

Sets the controller for the falloff parameter.

**Parameters:**

**Control \*c**

The controller to set.

**Return Value:**

TRUE if the controller was set; otherwise FALSE.

**Prototype:**

**virtual BOOL SetColorControl(Control \*c)=0;**

**Remarks:**

Sets the controller for the color parameter.

**Parameters:**

**Control \*c**

The controller to set.

**Return Value:**

TRUE if the controller was set; otherwise FALSE.

**Prototype:**

**virtual Control\* GetHotSpotControl()=0;**

**Remarks:**

Returns the controller for the hot spot parameter.

**Prototype:**

**virtual Control\* GetFalloffControl()=0;**

**Remarks:**

Returns the controller for the falloff parameter.

**Prototype:**

**virtual Control\* GetColorControl()=0;**

**Remarks:**

Returns the controller for the color parameter.

**Prototype:**

**virtual void SetAffectDiffuse(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Establishes if the light affects the diffuse color of objects.

**Parameters:**

**BOOL onOff**

TRUE to have the light affect the diffuse color; otherwise FALSE.

**Prototype:**

**virtual BOOL GetAffectDiffuse()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the light affects the diffuse color of objects; otherwise FALSE.

**Prototype:**

**virtual void SetAffectSpecular(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Establishes if the light affects the specular color of objects.

**Parameters:**

**BOOL onOff**

TRUE to have the light affect the specular color; otherwise FALSE.

**Prototype:**

**virtual BOOL GetAffectSpecular()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the light affects the specular color of objects; otherwise FALSE.

**Prototype:**

**virtual void SetDecayType(BOOL onOff);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the decay state of the light.

**Parameters:**

**BOOL onOff**

This boolean works as an integer where **0** is None, **1** is Inverse and **2** is Inverse Square.

**Default Implementation:**

```
{}
```

**Prototype:**

**virtual BOOL GetDecayType();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the decay state of the light.

**Return Value:**

This boolean works as an integer where **0** is None, **1** is Inverse and **2** is Inverse Square.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

**virtual void SetDecayRadius(TimeValue time, float f);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the decay radius (i.e. falloff) of the light.

**Parameters:**

**TimeValue time**

The time at which to set the radius.

**float f**

The radius to set.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual float GetDecayRadius(TimeValue t, Interval& valid =  
Interval(0,0));
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the decay radius of the light and updates the validity interval to reflect the validity of the radius controller.

**Parameters:**

**TimeValue t**

The time at which to return the radius.

**Interval& valid = Interval(0,0)**

The validity interval which is updated.

**Default Implementation:**

```
{ return 0.0f;}
```

**Prototype:**

```
virtual void SetDiffuseSoft(TimeValue time, float f);
```

**Remarks:**

This method is available in release 2.0 and later only.

Sets the state of the 'Soften Diffuse Edge' parameter.

**Parameters:**

**TimeValue time**

The time at which to set the value.

**float f**

The value to set in the range of 0.0 to 100.0.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual float GetDiffuseSoft(TimeValue t, Interval& valid =  
Interval(0,0));
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the state of the 'Soften Diffuse Edge' parameter.

**Parameters:**

**TimeValue t**

The time at which to return the value.

**Interval& valid = Interval(0,0)**

The validity interval that is updated to reflect the state of this parameter.

**Default Implementation:**

```
{ return 0.0f; }
```

**Prototype:**

```
virtual void SetShadColor(TimeValue t, Point3& rgb);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the shadow color to the specified value at the time passed.

**Parameters:**

**TimeValue t**

The time at which to set the shadow color.

**Point3& rgb**

The color to set.

**Default Implementation:**

```
}
```

**Prototype:**

```
virtual Point3 GetShadColor(TimeValue t, Interval &valid =  
Interval(0,0));
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the shadow color at the time passed and updates the validity interval passed to reflect the validity of the shadow color controller.

**Parameters:**

**TimeValue t**

The time at which to return the shadow color.

**Interval &valid = Interval(0,0)**

The validity interval which is updated.

**Default Implementation:**

```
{ return Point3(0,0,0); }
```

**Prototype:**

```
virtual BOOL GetLightAffectsShadow();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the Light Affects Shadow Color flag is set; otherwise FALSE.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual void SetLightAffectsShadow(BOOL b);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the state of the Light Affects Shadow Color flag to the value passed.

**Parameters:**

**BOOL b**

TRUE for set; FALSE for off.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void SetShadMult(TimeValue t, float m);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the shadow color multiplier (density) to the value passed at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the value.

**float m**

The value to set.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual float GetShadMult(TimeValue t, Interval &valid =  
Interval(0,0));
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the shadow color multiplier (density) at the specified time and updates the interval passed to reflect the validity of the multiplier controller.

**Parameters:**

**TimeValue t**

The time at which to return the value.

**Interval &valid = Interval(0,0)**

The interval which is updated.

**Default Implementation:**

```
{ return 1.0f; }
```

**Prototype:**

```
virtual Texmap* GetProjMap();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the texmap used as the projector image or NULL if not set.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void SetProjMap(Texmap* pmap);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the texmap to use as the light's projector image.

**Parameters:**

**Texmap\* pmap**

Points to the texmap to set or NULL to clear it.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual Texmap* GetShadowProjMap();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the texmap used as the shadow projector or NULL if not set.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void SetShadowProjMap(Texture* pmap);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the texmap to use as the light's shadow projector.

**Parameters:**

```
Texture* pmap
```

Points to the texmap to set or NULL to clear it.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual void SetAmbientOnly(BOOL onOff);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the ambient only flag to on or off.

**Parameters:**

```
BOOL onOff
```

TRUE for on; FALSE for off.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual BOOL GetAmbientOnly();
```

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the state of the ambient only flag. TRUE is on; FALSE is off.

**Default Implementation:**

```
{ return FALSE; }
```

# Class CameraObject

See Also: [Class Object](#).

class CameraObject : public Object

## Description:

This is a base class from which camera plug-ins may be derived. Methods of this class are used to get and set properties of the camera. All methods of this class are implemented by the plug-in.

## Methods:

### Prototype:

```
virtual RefResult EvalCameraState(TimeValue time,  
Interval& valid, CameraState* cs)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called to update the CameraState and validity interval at the specified time.

### Parameters:

#### **TimeValue time**

Specifies the time to evaluate the camera.

#### **Interval& valid**

The plug-in computes the validity interval of the camera at the specified time and stores the result here.

#### **CameraState\* cs**

The camera state to update. See [Structure CameraState](#). Note: The view vector and 'up' vector for the camera are stored with the matrix transform for the node. Cameras can be multiple-instanced so it must work this way. To get at this matrix use the following method from [Class INode](#):

```
virtual Matrix3 GetObjTMAfterWSM(TimeValue time,  
Interval* valid=NULL)=0;
```

The scaling of this matrix may be removed by normalizing each of the rows.

### Return Value:

**REF\_SUCCEEDED** if the camera state was updated successfully; otherwise **REF\_FAIL**.

**Prototype:**

**virtual void SetOrtho(BOOL b)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets whether the camera is on ortho mode or not.

**Parameters:**

**BOOL b**

Pass TRUE for ortho and FALSE for not ortho.

**Prototype:**

**virtual BOOL IsOrtho()=0;**

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the camera is in ortho mode and FALSE if it is not.

**Prototype:**

**virtual void SetFOV(TimeValue time, float f)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets the field-of-view of the camera at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the field-of-view.

**float f**

The value to set in radians.

**Prototype:**

**virtual float GetFOV(TimeValue t, Interval& valid = Interval(0,0))=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the field-of-view setting of the camera at the specified time and adjusts the validity interval of the camera at this time to reflect the field-of-view parameter.

**Parameters:****TimeValue t**

The time to retrieve the field-of-view setting.

**Interval& valid = Interval(0,0)**

The validity interval to set.

**Return Value:**

The field-of-view of the camera in radians.

**Prototype:**

```
virtual void SetTDist(TimeValue time, float f)=0;
```

**Remarks:**

Implemented by the Plug-In.

Sets the target distance setting (for free cameras) at the specified time.

**Parameters:****TimeValue t**

The time at which to set the target distance.

**float f**

The value to set.

**Prototype:**

```
virtual float GetTDist(TimeValue t, Interval& valid =  
Interval(0,0))=0;
```

**Remarks:**

Implemented by the Plug-In.

Returns the target distance setting of the camera at the specified time and adjusts the validity interval of the camera to reflect the target distance parameter.

**Parameters:**

**TimeValue t**

The time to retrieve the target distance setting.

**Interval & valid = Interval(0,0)**

This validity interval is intersected with the validity interval of the target distance parameter.

**Return Value:**

The target distance of the camera.

**Prototype:**

**virtual int GetManualClip()=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the manual clip flag. This indicates the camera will perform clipping at its hither and yon distances.

**Return Value:**

Nonzero if manual clipping is enabled; otherwise 0.

**Prototype:**

**virtual void SetManualClip(int onOff)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets the manual clip flag. This indicates the camera will perform clipping at its hither and yon distances.

**Parameters:**

**int onOff**

The state of the manual clipping flag to set. Nonzero indicates clipping will be performed.

**Prototype:**

**virtual float GetClipDist(TimeValue t, int which,  
Interval &valid=Interval(0,0))=0;**

**Remarks:**

Implemented by the Plug-In.

Retrieves the clipping distance of the specified plane at the specified time and modifies the validity interval to reflect the setting of the clipping distance parameter.

**Parameters:****TimeValue t**

The time to retrieve the clipping distance.

**int which**

Indicates which distance to return. One of the following values:

**CAM\_HITHER\_CLIP** - The hither distance

**CAM\_YON\_CLIP** - The yon distance.

**Interval &valid=Interval(0,0)**

The validity interval that this method will update to reflect the clipping distance interval.

**Return Value:**

The clipping distance.

**Prototype:**

```
virtual void SetClipDist(TimeValue t, int which, float val)=0;
```

**Remarks:**

Implemented by the Plug-In.

Sets the clipping distance of the specified plane at the specified time.

**Parameters:****TimeValue t**

The time to set the clipping distance.

**int which**

Indicates which distance to set. One of the following values:

**CAM\_HITHER\_CLIP** - The hither distance

**CAM\_YON\_CLIP** - The yon distance.

**float val**

The distance to set.

**Prototype:**

**virtual void SetEnvRange(TimeValue time, int which, float f)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets the environment range distance at the specified time.

**Parameters:****TimeValue time**

The time to set the environment range.

**int which**

Indicates which distance to set. One of the following values:

**ENV\_NEAR\_RANGE** - The near distance.

**ENV\_FAR\_RANGE** - The far distance.

**float f**

The distance to set.

**Prototype:**

**virtual float GetEnvRange(TimeValue t, int which,  
Interval& valid = Interval(0,0))=0;**

**Remarks:**

Implemented by the Plug-In.

Retrieves the environment range distance at the specified time and intersects the specified validity interval with the interval of the environment range parameter.

**Parameters:****TimeValue time**

The time to retrieve the environment range.

**int which**

Indicate which distance to set. One of the following values:

**ENV\_NEAR\_RANGE** - The near distance.

**ENV\_FAR\_RANGE** - The far distance.

**Interval& valid = Interval(0,0)**

The validity interval that this method will update to reflect the environment

range setting.

**Return Value:**

The environment range distance at the specified time.

**Prototype:**

**virtual void SetEnvDisplay(BOOL b, int notify=TRUE)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets the environment range display flag. This indicates if the camera will display its range settings.

**Parameters:**

**BOOL b**

The flag state to set.

**int notify=TRUE**

If notify is TRUE, dependents of this message are sent the

**REFMSG\_CHANGE** message, via:

**NotifyDependents(FOREVER, PART\_OBJ,  
REFMSG\_CHANGE);**

Otherwise no notification is sent.

**Prototype:**

**virtual BOOL GetEnvDisplay()=0;**

**Remarks:**

Implemented by the Plug-In.

Retrieves the environment range display setting.

**Return Value:**

TRUE if ranges are displayed; otherwise FALSE.

**Prototype:**

**virtual void RenderApertureChanged(TimeValue t)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called on all cameras when the render aperture width has changed.

**Parameters:**

**TimeValue t**

The time of the change.

**Prototype:**

**virtual void UpdateTargDistance(TimeValue t, INode \*inode);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called on all target cameras when the target distance has changed. For instance, a distance shown in the user interface may be updated in this method.

**Parameters:**

**TimeValue t**

The time of the change.

**INode \*inode**

The camera node.

**Default Implementation:**

**{**

**Prototype:**

**virtual void SetIMultiPassCameraEffect(IMultiPassCameraEffect \*pIMultiPassCameraEffect);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the plug-in.

The **IMultiPassCameraEffect** should be checked to see if compatible with the camera before being assigned.

**Parameters:**

**IMultiPassCameraEffect \*pIMultiPassCameraEffect**

The IMultiPassCameraEffect to assign.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void SetMultiPassEffectEnabled(TimeValue t, BOOL  
enabled);
```

**Remarks:**

This method is available in release 4.0 and later only.  
Enables or disables the multi-pass effect.

**Parameters:**

**TimeValue t**

The time at which to enable the effect.

**BOOL enabled**

TRUE for enabled; FALSE for disabled.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BOOL GetMultiPassEffectEnabled(TimeValue t, Interval&  
valid);
```

**Remarks:**

This method is available in release 4.0 and later only.  
Returns the enabled or disabled state of the multi-pass effect setting for the camera.

**Parameters:**

**TimeValue t**

The time at which to get the setting.

**Interval& valid**

The validity interval for the setting.

**Return Value:**

TRUE for enabled; FALSE for disabled.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual IMultiPassCameraEffect *GetIMultiPassCameraEffect();
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the current multi-pass camera effect. See [Class IMultiPassCameraEffect](#).

**Default Implementation:**

```
{ return NULL; }
```

## Class GenCamera

See Also: [Class CameraObject](#), [Class Interval](#), [Class Control](#).

```
class GenCamera : public CameraObject
```

### Description:

This class describes a generic camera object. It is used as a base class for creating plug-in cameras. Methods of this class are used to get and set properties of the camera. All methods of this class are implemented by the plug-in.

### Methods:

#### Prototype:

```
virtual GenCamera *NewCamera(int type)=0;
```

#### Remarks:

Creates a new generic camera object.

#### Parameters:

**int type**

Nonzero if the camera has a target; otherwise 0.

#### Return Value:

A pointer to a new instance of the specified light type.

#### Prototype:

```
virtual void SetConeState(int s)=0;
```

#### Remarks:

Sets if the camera cone is displayed in the viewports.

#### Parameters:

**int s**

Nonzero to display the camera cone; otherwise 0.

#### Prototype:

```
virtual int GetConeState()=0;
```

#### Remarks:

Returns TRUE if the camera cone is displayed in the viewports; otherwise

FALSE.

**Prototype:**

**virtual void SetHorzLineState(int s)=0;**

**Remarks:**

Sets if the camera has a horizon line displayed.

**Parameters:**

**int s**

Nonzero to display the horizon line; otherwise 0.

**Prototype:**

**virtual int GetHorzLineState()=0;**

**Remarks:**

Returns TRUE if the camera has a horizon line displayed; otherwise FALSE.

**Prototype:**

**virtual void Enable(int enab)=0;**

**Remarks:**

Enables or disables the camera. If enabled the camera may be displayed, hit tested, etc.

**Parameters:**

**int enab**

Nonzero to enable; zero to disable.

**Prototype:**

**virtual BOOL SetFOVControl(Control \*c)=0;**

**Remarks:**

Sets the controller for the field-of-view parameter.

**Parameters:**

**Control \*c**

Points to the controller to set.

**Return Value:**

Returns TRUE if set; otherwise FALSE.

**Prototype:**

```
virtual Control *GetFOVControl()=0;
```

**Remarks:**

Returns the controller for the field-of-view parameter.

**Prototype:**

```
virtual void SetFOVType(int ft)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the Field-Of-View type of the camera.

**Parameters:****int ft**

One of the following values:

**FOV\_W**

Width-related FOV

**FOV\_H**

Height-related FOV

**FOV\_D**

Diagonal-related FOV

**Prototype:**

```
virtual int GetFOVType()=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the Field-Of-View type of the camera. One of the following values:

**FOV\_W**

Width-related FOV

**FOV\_H**

Height-related FOV

## **FOV\_D**

Diagonal-related FOV

### **Prototype:**

```
virtual int Type()=0;
```

### **Remarks:**

This method is available in release 3.0 and later only.

Returns one of the following values to indicate the camera type:

**FREE\_CAMERA** (No Target)

**TARGETED\_CAMERA** (Target / Look At Controller)

**PARALLEL\_CAMERA** (Orthographic Camera)

### **Prototype:**

```
virtual void SetType(int tp)=0;
```

### **Remarks:**

This method is available in release 3.0 and later only.

Sets the type of camera.

### **Parameters:**

**int tp**

One of the following types:

**FREE\_CAMERA** (No Target)

**TARGETED\_CAMERA** (Target / Look At Controller)

**PARALLEL\_CAMERA** (Orthographic Camera)

## Class Modifier

See Also: [Class BaseObject](#), [Class ObjectState](#), [Class ModContext](#), [Class ModContextEnumProc](#), [Class Interval](#), [Class ISave](#), [Class ILoad](#), [Class Class\\_ID](#).

```
class Modifier : public BaseObject
```

### Description:

This is the class from which you may derive Object Space and Space Warp (World Space) Modifier plug-ins. This class is subclassed off of **BaseObject** so the modifier can put up a graphical representation in the viewport to use as a gizmo.

### Method Groups:

The hyperlinks below jump to the start of groups of related methods within the class:

[Modifier Name Access](#)

[InputType, ChannelsUsed, ChannelsChanged.](#)

[Object Modification Methods](#)

[Topology Dependence](#)

[Loading and Saving Methods](#)

[Notification of Change](#)

[Modifier Stack Enable/Disable Methods](#)

[ModContext Enumeration](#)

[Validity Intervals](#)

### Methods:

#### Modifier Name Access

#### Prototype:

```
virtual TSTR GetName();
```

#### Remarks:

Implemented by the System.

Returns the name of the modifier.

**Prototype:**

**virtual void SetName(TSTR n);**

**Remarks:**

Implemented by the System.

Sets the name of the modifier to the name passed.

**Parameters:**

**TSTR n**

Specifies the name to set.

**InputType, ChannelsUsed, ChannelsChanged.****Prototype:**

**virtual Class\_ID InputType()=0;**

**Remarks:**

Implemented by the Plug-In.

This is the type of object that the modifier knows how to modify. Simple modifiers that just modify points of an object can operate on generic 'Deformable' objects. Deformable objects are any type of object that has points. A modifier could also work on a particular type of object such as a TriObject or PatchObject.

**Return Value:**

The Class\_ID of the item. You can request any Class\_ID for your input type. For example, **Class\_ID(OMNI\_LIGHT\_CLASS\_ID, 0)**. See [List of Class IDs](#).

**Prototype:**

**virtual ChannelMask ChannelsUsed()=0;**

**Remarks:**

Implemented by the Plug-In.

These are channels that the modifier needs in order to perform its modification. This should at least include the channels specified in

**ChannelsChanged()** but may include more.

Note that **ChannelsUsed()** is called many times but the channels returned

should not change on the fly.

**Return Value:**

The channels required. See [List of Channel Bits](#).

**Sample Code:**

```
{ return GEOM_CHANNEL|TOPO_CHANNEL; }
```

**Prototype:**

```
virtual ChannelMask ChannelsChanged()=0;
```

**Remarks:**

Implemented by the Plug-In.

These are the channels that the modifier actually modifies. Note that **ChannelsChanged()** is called many times but the channels returned should not change on the fly.

**Return Value:**

The channels that are changed. See [List of Channel Bits](#).

**Prototype:**

```
ChannelMask TotalChannelsUsed();
```

**Remarks:**

Returns the same value as **ChannelsUsed()** above except **GFX\_DATA\_CHANNEL** will be ORed in if the **TOPO\_CHANNEL** or the **TEXMAP\_CHANNEL** are being used.

**Prototype:**

```
ChannelMask TotalChannelsChanged();
```

**Remarks:**

Returns the same value as **ChannelsChanged()** above except **GFX\_DATA\_CHANNEL** will be ORed in if the **TOPO\_CHANNEL**, the **TEXMAP\_CHANNEL**, or the **VERTCOLOR\_CHANNEL** are being changed.

**Prototype:**

**virtual bool ChangesSelType();**

**Remarks:**

This method is available in release 4.0 and later only.

If a modifier want to make it possible to switch dynamically between changing the selection type that flows up the stack, or leaving it like it is, it can overwrite this. The default implementation indicates that it changes the selection type, if the **SUBSEL\_TYPE\_CHANNEL** is part of **ChannelsChanged()**. Note that **ChannelsChanged()** can not dynamically changed for various reasons.

**Default Implementation:**

```
{ return ChannelsChanged() & SUBSEL_TYPE_CHANNEL ? true  
: false; }
```

## Object Modification

**Prototype:**

**virtual void ModifyObject(TimeValue t, ModContext &mc,  
ObjectState\* os, INode \*node)=0;**

**Remarks:**

Implemented by the Plug-In.

This is the method that actually modifies the input object. This method is responsible for altering the object and then updating the validity interval of the object to reflect the validity of the modifier.

**Parameters:**

**TimeValue t**

The time at which the modification is being done.

**ModContext &mc**

A reference to the ModContext.

**ObjectState\* os**

The object state flowing through the pipeline. This contains a pointer to the object to modify.

**INode \*node**

The node the modifier is applied to. This parameter is always NULL for

Object Space Modifiers and non-NULL for World Space Modifiers (Space Warps). This is because a given WSM is only applied to a single node at a time whereas an OSM may be applied to several nodes. This may be used for example by particle system space warps to get the transformation matrix of the node at various times.

See Also: Advanced Topics section on [Object Modification](#).

## Validity Intervals

### Prototype:

**virtual Interval LocalValidity(TimeValue t);**

### Remarks:

Implemented by the Plug-In.

This method returns the validity interval of a modifier. It is simply the combination of the validity of all the modifier's parameters. It's used to determine when to cache in the pipeline, but is not directly responsible for determining when **ModifyObject()** is called. **ModifyObject()** is called when the pipeline needs to be evaluated either because someone sent a **REFMSG\_CHANGE** message or the validity of the object does not include the current time.

If a modifier is not animated it's OK to simply return **FOREVER** from this method. In the case where the modifier changes because a user changes a non-animated control in the user interface (for instance a check box), you can cause reevaluation by notifying your dependents of the change, i.e.:

**NotifyDependents(FOREVER, PART\_ALL,  
REFMSG\_CHANGE);**

### Parameters:

**TimeValue t**

The time to calculate the Interval.

See Also: Advanced Topics on [Intervals](#).

## Topology Dependence

### Prototype:

## **virtual BOOL DependOnTopology(ModContext &mc)**

### **Remarks:**

Implemented by the Plug-In.

Modifiers that place a dependency on topology should return TRUE for this method. An example would be a modifier that stores a selection set base on vertex indices. This modifier depends on the indices being intact for it to operate correctly.

### **Parameters:**

#### **ModContext &mc**

Reference to the ModContext.

### **Default Implementation:**

```
{ returns FALSE; }
```

### **Return Value:**

TRUE if the modifier depends on topology; otherwise FALSE.

## **Modifier Stack Access**

### **Prototype:**

```
void DisableMod()
```

### **Remarks:**

Implemented by the System.

This disables the modifier in the history browser (modifier stack).

### **Prototype:**

```
void EnableMod()
```

### **Remarks:**

Implemented by the System.

This enables the modifier in the history browser (modifier stack).

### **Prototype:**

```
int IsEnabled()
```

### **Remarks:**

Implemented by the System.

This returns the status (enabled or disabled) of the modifier in the history browser.

**Return Value:**

Nonzero if enabled; otherwise 0.

**Prototype:**

**void DisableModInRender()**

**Remarks:**

Implemented by the System.

This turns off the modifier in the renderer

**Prototype:**

**void EnableModInRender()**

**Remarks:**

Implemented by the System.

This turns on the modifier in the renderer

**Prototype:**

**int IsEnabledInRender()**

**Remarks:**

Implemented by the System.

This returns the status (enabled or disabled) of the modifier in the renderer.

**Return Value:**

Nonzero if enabled; otherwise 0.

**Prototype:**

**void DisableModInViews();**

**Remarks:**

Implemented by the System.

Disables the modifier in the viewports (it remains active in the renderer unless **DisableMod()** above is used).

**Prototype:**

```
void EnableModInViews();
```

**Remarks:**

Implemented by the System.

Enables the modifier in the viewports.

**Prototype:**

```
int IsEnabledInViews();
```

**Remarks:**

Implemented by the System.

Returns nonzero if the modifier is enabled in the viewports; otherwise zero.

**Prototype:**

```
void DisableModApps()
```

**Remarks:**

This method is used internally.

**Prototype:**

```
void EnableModApps()
```

**Remarks:**

This method is used internally.

**Notification of Input Changed****Prototype:**

```
virtual void NotifyInputChanged(Interval changeInt, PartID  
partID, RefMessage message, ModContext *mc)
```

## Remarks:

Implemented by the Plug-In.

This method is called when an item in the modifier stack before this modifier sends a **REFMSG\_CHANGE** message via **NotifyDependents()**.

Consider the following example: Assume the modifier stack contains a Sphere Object, then a Bend, then a Edit Mesh. The Edit Mesh modifier does not have a reference to the Bend or the Sphere because it does not officially depend on these items. However it does depend on them in a certain sense, because it modifies the data that these items produce. So, if they change it may affect the modifier. A modifier may build a cache based on its input object. The modifier needs a way to know when to discard this cache because the input object has changed. Whenever one of the items before this modifier in the stack sends out a **REFMSG\_CHANGE** message via **NotifyDependents()** to indicate it has changed this method is called. The modifier may respond in a way appropriate to it, for example by discarding its cache of the input object.

It is not legal, to issue a **NotifyDependent()**'s in the **NotifyInputChanged()** method of a modifier, without checking for reentrancy. Imagine, you have an instanced modifier applied to the same object in the stack. Sending a refmsg from the **NotifyInputChanged** method will casue an endles loop. Simply putting a guard in, that checks for reentrancy should get rid of the problem.

## Parameters:

### **Interval changeInt**

This is the interval from the message. It is reserved for future use - now it will always be FOREVER.

### **PartID partID**

This is the partID from the message.

### **RefMessage message**

This is the message sent.

### **ModContext \*mc**

The ModContext for the pipeline that changed. If a modifier is applied to multiple objects, then there are ModApps in each pipeline that it is applied to. These ModApps are pointing to the same modifier. Consider the following example: Say you apply a Bend modifier to a Sphere, a Cylinder and a Box object. There are three ModApps but only one Bend modifier. Then you go to

the Sphere and adjust its Radius. This will cause **NotifyInputChanged()** to be called on the Bend because the Bend's input changed. However only one of its inputs changed - only the Sphere changed and not the Cylinder or the Box. Therefore **NotifyInputChanged()** will be called once, and the ModContext passed in will be for the Sphere's changed pipeline. It is possible that all three objects could change at the same time. If an instanced float controller was assigned to the radius, width, and height - one parameter for each object - then the controller was adjusted in the function curve editor, all three items would change. In this case **NotifyInputChanged()** would be called three times on the Bend. Once for each pipeline, once with each ModContext.

## Loading and Saving Methods

### Prototype:

```
IOResult Save(ISave *isave);
```

### Remarks:

Implemented by the System.

This method handles saving the modifier name. This method should be called by the derived class BEFORE it saves any chunks. See the sample code below.

### Parameters:

```
ISave *isave
```

You may use this pointer to call methods of ISave to write data.

### Return Value:

One of the following values: **IO\_OK, IO\_ERROR.**

### Sample Code:

```
IOResult DispMod::Save(ISave *isave)  
{  
// First save the modifier name by  
// calling the base class version.  
Modifier::Save(isave);  
// Then save this modifiers data.  
isave->BeginChunk(BMIO_CHUNK);  
bi.Save(isave);  
isave->EndChunk();
```

```
return IO_OK;
}
```

**Prototype:**

**IOResult Load(ILoad \*iLoad);**

This method handles loading the modifier name. It should be called by the derived class BEFORE it loads any chunks.

**Remarks:**

Implemented by the System.

**Parameters:**

**ILoad \*iLoad**

You may use this pointer to call methods of ILoad to read data.

**Return Value:**

One of the following values: **IO\_OK, IO\_ERROR.**

**Prototype:**

**virtual IOResult LoadLocalData(ILoad \*iLoad, LocalModData \*\*pLd)**

**Remarks:**

Implemented by the Plug-In.

When a 3ds max file is being loaded, this method is called so that the modifier can load the LocalModData structure that is hung off each ModContext. If the modifier doesn't store any data in the ModContext it can ignore this method.

**Parameters:**

**ILoad \*iLoad**

You may use this pointer to call methods of ILoad to read data.

**LocalModData \*\*pLd**

A pointer to a pointer in the ModContext. The modifier must set this pointer to point at a new LocalModData derived class.

**Return Value:**

One of the following values: **IO\_OK, IO\_ERROR.**

**Prototype:**

**virtual IOResult SaveLocalData(ISave \*isave, LocalModData \*ld)**

**Remarks:**

Implemented by the Plug-In.

When a 3ds max file is being saved, this method is called so that the modifier can save the localData structure that is hung off each ModContext. If the modifier doesn't store any data in the ModContext it can ignore this method.

**Parameters:**

**ISave \*isave**

You may use this pointer to call methods of ISave to write data.

**LocalModData \*ld**

Pointer to the LocalModData for the modifier.

**Return Value:**

One of the following values: **IO\_OK**, **IO\_ERROR**.

## **ModContext Enumeration**

**Prototype:**

**void EnumModContexts(ModContextEnumProc \*proc);**

**Remarks:**

Implemented by the System.

This method will call the callback object proc method once for each application of the modifier.

**Parameters:**

**ModContextEnumProc \*proc**

The callback object whose proc method is called.

See Also: [Class ModContextEnumProc](#).

**Prototype:**

**void GetIDerivedObject(ModContext \*mc, IDerivedObject  
\*&derObj, int &modIndex);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will retrieve the IDerivedObject and index of this modifier for a given modifier context.

**Parameters:**

**ModContext \*mc**

Points to the ModContext for the modifier.

**IDerivedObject \*&derObj**

A pointer to the IDerivedObject is returned here.

**int &modIndex**

The zero based index of the modifier in the derived object is returned here.

**Prototype:**

**virtual void CopyAdditionalChannels(Object \*fromObj, Object \*toObj);**

**Remarks:**

This method is available in release 4.0 and later only.

In case the modifier changes the object type (basically the os->obj pointer in ModifyObject) \*and\* changes the ExtensionChannel, it has to overwrite this method and copy only the channels that it doesn't modify/added already to the new object.

**Parameters:**

**Object \*fromObj**

**Object \*toObj**

**Default Implementation:**

**{ toObj->CopyAdditionalChannels(fromObj);}**

**Prototype:**

**virtual int NeedUseSubselButton();**

**Remarks:**

This method is no longer used.

## Class SimpleMod

See Also: [Class Modifier](#).

class SimpleMod : public Modifier

### Description:

The **SimpleMod** class supplies most of the methods needed to implement an object space modifier.

To be a 'Simple' modifier, the following assumptions are made:

The modifier only modifies the geometry channel.

The modifier uses an instance of a class derived from Deformer to do the modifying.

The modifier's gizmo is represented as a 3D box that has had the modifier applied to it.

This class maintains a pointer to a parameter block. If the client of **SimpleMod** uses a single parameter block then **SimpleMod** can manage all the methods associated with SubAnims and References for the client.

If the client of **SimpleMod** maintains several parameter blocks then the client must implement the methods **NumSubs()**, **SubAnim(i)**, **SubAnimName(i)**, **NumRefs()**, **GetReference(i)** and **SetReference(i)** and call the **SimpleMod** methods when 'i' refers to the parameters maintained by **SimpleMod**.

When clients of **SimpleMod** are cloning themselves, they should call this method on the clone to copy **SimpleMod**'s data.

```
void SimpleModClone(SimpleMod *smodSource);
```

Clients of **SimpleMod** probably want to override these. If they do they should call these from within their implementation of these methods.

```
void BeginEditParams(IObjParam *ip, ULONG flags,Animatable *prev);
```

```
void EndEditParams(IObjParam *ip, ULONG flags,Animatable *next);
```

### Data Members:

public:

```
Control *tmControl;
```

Points to the transform controller for the Gizmo.

**Control \*posControl;**

Points to the position controller for the Center.

**IParamBlock \*pblock;**

Pointer to a parameter block. Clients of SimpleMod should use the following value as the reference index of this parameter block. **#define**

**SIMPMOD\_PBLOCKREF 2**

**static IObjParam \*ip;**

Storage for the interface pointer.

**static MoveModBoxCMode \*moveMode;**

Storage for the move modifier box command mode.

**static RotateModBoxCMode \*rotMode;**

Storage for the rotate modifier box command mode.

**static UScaleModBoxCMode \*uscaleMode;**

Storage for the uniform scale modifier box command mode.

**static NUScaleModBoxCMode \*nuscaleMode;**

Storage for the non-uniform scale modifier box command mode.

**static SquashModBoxCMode \*squashMode;**

Storage for the squash modifier box command mode.

**static SimpleMod \*editMod;**

Storage for the instance of SimpleMod that is being edited in the command panel.

**Methods:**

**Prototype:**

**virtual Deformer& GetDeformer(TimeValue t,ModContext &mc,Matrix3& mat,Matrix3& invmat)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is used to retrieve the callback object that will handle the deformation.

**Parameters:**

**TimeValue t**

Specifies the time the modification is being performed.

### **ModContext &mc**

A reference to the ModContext.

### **Matrix3& mat**

A reference to a matrix that describes the space the modification is supposed to happen in. This is computed from the ModContext matrix and the controllers controlling the gizmo and center of the modifier. The plug-in developers job is simply to transform each point to be deformed by this matrix before it performs its own deformation to the point. After the modifier applies its own deformation to the point, the developer transforms the point by the inverse of this matrix (passed below).

### **Matrix3& invmat**

This is the inverse of the matrix above. See the comment above for how this is used.

### **Return Value:**

A C++ reference to the deformer callback object.

### **Prototype:**

**virtual void InvalidateUI()**

### **Remarks:**

Implemented by the Plug-In.

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

### **Example:**

If the plug-in uses a parameter map for handling its UI, it may call a method of the parameter map to handle this: **pmapParam->Invalidate();**

If the plug-in does not use parameter maps, it should call the **SetValue()** method on each of its controls that display a value, for example the spinner controls. This will cause to the control to update the value displayed. The code below shows how this may be done for a spinner control. Note that ip and pblock are assumed to be initialized interface and parameter block pointers

**float newval;**

```
Interval valid=FOREVER;  
TimeValue t=ip->GetTime();  
// Get the value from the parameter block at the current time.  
pblock->GetValue( PB_ANGLE, t, newval, valid );  
// Set the value. Note that the notify argument is passed as  
FALSE.  
// This ensures no messages are sent when the value changes.  
angleSpin->SetValue( newval, FALSE );
```

**Prototype:**

```
virtual Interval GetValidity(TimeValue t)
```

**Remarks:**

Implemented by the Plug-In.

The SimpleMod class calls this method to retrieve the validity interval of the modifier. The modifier provides this interval by starting an interval at FOREVER and intersecting it with each of its parameters validity intervals.

**Parameters:**

**TimeValue t**

The time to compute the validity interval.

**Default Implementation:**

```
{return FOREVER;}
```

**Return Value:**

The validity interval of the modifier.

See Also: The Advanced Topics section on [Intervals](#).

**Prototype:**

```
virtual ParamDimension *GetParameterDim(int pbIndex)
```

**Remarks:**

Implemented by the Plug-In.

Returns the dimension of the parameter whose index is passed. See Class [ParamDimension](#).

**Parameters:****int pbIndex**

The index of the parameter.

**Default Implementation:****{return defaultDim;}****Return Value:**

A pointer to the dimension of the parameter.

**Prototype:****virtual TSTR GetParameterName(int pbIndex)****Remarks:**

Implemented by the Plug-In.

Returns the name of the parameter whose index is passed.

**Parameters:****int pbIndex**

Index of the parameter.

**Default Implementation:****{return TSTR(\_T("Parameter"))};}****Return Value:**

The name of the parameter.

**Prototype:****virtual BOOL GetModLimits(TimeValue t,float &zmin, float &zmax, int &axis)****Remarks:**

Implemented by the Plug-In.

If the effect can be limited (like the way bend/taper/twist/etc. can be limited) then it should specify the min and max limits and the axis that it is limited along. SimpleMod will then display the limits as part of the Gizmo. If it does not support limits then it should return FALSE or simply not implement this method.

**Parameters:**

**TimeValue t**

The time to get the limits.

**float &zmin**

The min limit.

**float &zmax**

The max limit.

**int &axis**

The axis that it is limited along: x=0, y=1, z=2.

**Return Value:**

TRUE if limits are supported; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

# Class Control

See Also: [Class ReferenceTarget](#), [Class IKeyControl](#), [Class IKDeriv](#), [Class IKEnumCallback](#), [Class StdControl](#), [List of Additional Controller Related Functions](#), [Class AngAxis](#), [Class Quat](#), [Class Interval](#), [Class Matrix3](#), [Class Point3](#), [Class ScaleValue](#), [Class JointParams](#), [Class SetXFormPacket](#).

```
class Control : public ReferenceTarget
```

## Description:

Control is the class from which you may derived controller objects. Controllers are the objects in 3ds max that control animation. Controllers come in different types based on the type of data they control. For example, Transform controllers control the 4x3 matrices used to define the position of nodes in the scene while float controllers control simple floating point values.

Note: Many controller plug-ins may be able to subclass from StdControl rather than Control. This simplifies the developers job. StdControl handles the processing of Out of Range Types, Ease Curves, and Multiplier Curves. See [Class StdControl](#) for more information.

Plug-In Information:

Class Defined In CONTROL.H

Super Class ID CTRL\_FLOAT\_CLASS\_ID - Used by float controllers.

CTRL\_POINT3\_CLASS\_ID - Used by Point3 controllers.

CTRL\_MATRIX3\_CLASS\_ID - Used by Matrix3 controllers.

CTRL\_POSITION\_CLASS\_ID - Used by position controllers.

CTRL\_ROTATION\_CLASS\_ID - Used by rotation controllers.

CTRL\_SCALE\_CLASS\_ID - Used by scale controllers.

CTRL\_MORPH\_CLASS\_ID - Used by morph controllers.

Standard File Name Extension DLC

Extra Include File Needed None

## Method Groups:

The following hyperlinks jump to the start of groups of related methods within the class:

[GetValue and SetValue](#)

[Inverse Kinematics \(IK\)](#)

[Ease Curve, Multiplier Curve and Out of Range Types](#)

[Access to Sub-Controllers](#)

[New Controller Assignment](#)

[Transform Controller methods](#)

[Lookat Controller methods](#)

[IsLeaf/IsKeyable / IsColorController](#)

[Copying and Pasting](#)

[Gizmo / Sub-Object methods](#)

[Load / Save](#)

[Post Clone Notification](#)

[Undo / Redo](#)

[Miscellaneous methods](#)

### **Methods:**

### **Prototype:**

```
virtual void GetValue(TimeValue t, void *val, Interval &valid,  
    GetSetMethod method=CTRL_ABSOLUTE)=0;
```

### **Remarks:**

Implemented by the Plug-In.

Retrieves the value of the controller at the specified time, and updates the validity interval passed in to reflect the interval of the controller. This method is responsible for handling Out of Range Types, Ease Curves and Multiplier Curves. See the sample code below.

### **Parameters:**

#### **TimeValue t**

Specifies the time to retrieve the value.

#### **void \*val**

This points to a variable to hold the computed value of the controller at the specified time. What the plug-in needs to do to store the value of the controller depends on the controller type. There are six controller types: **float**, **Point3**, **Position**, **Rotation**, **Scale**, and **Transform**. The way the value is stored also depends on the **GetSetMethod** parameter **method**. Below is list of the possible cases and how the value should be stored in each case.

#### **float**

If **method == CTRL\_ABSOLUTE**

**\*val** points to a float

The controller should simply store the value.

If **method == CTRL\_RELATIVE**

**\*val** points to a float

The controller should add its value to the existing floating point value.

### **Point3**

If **method == CTRL\_ABSOLUTE**

**\*val** points to a **Point3**

The controller should simply store the value.

If **method == CTRL\_RELATIVE**

**\*val** points to a **Point3**

The controller should add its value to the existing **Point3** value.

### **Position**

If **method == CTRL\_ABSOLUTE**

**\*val** points to a **Point3**

The controller should simply store the value.

If **method == CTRL\_RELATIVE**

**\*val** points to a **Matrix3**

The controller should apply its value to the matrix by pre-multiplying its position.

**Matrix3 \*mat = (Matrix3\*)val;**

**Point3 v = the computed value of the controller...**

**mat->PreTranslate(v);**

### **Rotation**

If **method == CTRL\_ABSOLUTE**

**\*val** points to a **Quat**

The controller should simply store the value.

If **method == CTRL\_RELATIVE**

**\*val** points to a **Matrix3**

The controller should apply its value to the matrix by pre-multiplying its rotation.

**Matrix3 \*mat = (Matrix3\*)val;**

**Quat q** = the computed value of the controller...

**PreRotateMatrix(\*mat,q);**

## Scale

If **method == CTRL\_ABSOLUTE**

**\*val** points to a **ScaleValue**

The controller should simply store the value.

If **method == CTRL\_RELATIVE**

**\*val** points to a **Matrix3**

The controller should apply its value to the matrix by pre-multiplying its scale.

**Matrix3 \*mat = (Matrix3\*)val;**

**ScaleValue s** = the computed value of the controller...

**ApplyScaling(\*mat,s);**

## Transform (Matrix3)

If **method == CTRL\_ABSOLUTE**

**\*val** points to a **Matrix3**

The controller should simply store the value.

**Important Note: Developers should only pass**

**CTRL\_RELATIVE when getting the value of a Matrix3 controller. This is because the controller may use the matrix as input to compute the value. Therefore it is not acceptable to use CTRL\_ABSOLUTE to get the value.**

If **method == CTRL\_RELATIVE**

**\*val** points to a **Matrix3**.

The controller should apply its value to the matrix by pre-multiplying. When **GetValue()** is called on a transform controller the method is **CTRL\_RELATIVE** and the matrix passed is usually the **parent** of the node.

**Important Note for Matrix3 Controllers:** when **SetValue()** is called **\*val** points to an instance of [Class SetXFormPacket](#). See that class for more details on how it is used.

### **Interval &valid**

The validity interval to update. The controllers validity interval should be intersected with this interval. This updates the interval to reflect the interval of the controller.

### **GetSetMethod method=CTRL\_ABSOLUTE**

One of the following values:

#### **CTRL\_RELATIVE**

Indicates the plug-in should apply the value of the controller to **\*val**. See Above.

#### **CTRL\_ABSOLUTE**

Indicates the controller should simply store its value in **\*val**. See Above.

### **Sample Code:**

The following code is from the StdControl implementation of this method. It demonstrates how the out of range and multiplier curves are handled.

```
void StdControl::GetValue(TimeValue t, void *val, Interval &valid,
GetSetMethod method)
{
    Interval range = GetTimeRange(TIMERANGE_ALL);
    Interval wvalid = FOREVER, cvalid = FOREVER;
    void *ptr = val;
    int ort;
    float m;
    TimeValue oldTime = t;

    if (method==CTRL_RELATIVE) {
        ptr = CreateTempValue();
    }

    // Grab the multiplier before the time warp.
    m = GetMultVal(t,valid);
```

```

// Apply the time warp.
t = ApplyEase(t,wvalid);

if (t<=range.Start()) {
    ort = GetORT(ORT_BEFORE);
} else {
    ort = GetORT(ORT_AFTER);
}

if (/*ort==ORT_CONSTANT*/TestAFlag(A_ORT_DISABLED) ||
    range.Empty() || range.InInterval(t)) {

    GetValueLocalTime(t,ptr,cvalid);
} else {
    switch (ort) {
    case ORT_CONSTANT:
        if (t<range.Start()) {
            GetValueLocalTime(range.Start(),ptr,cvalid,CTRL_ABSOLUTE);
            cvalid.Set(TIME_NegInfinity,range.Start());
        } else {
            GetValueLocalTime(range.End(),ptr,cvalid,CTRL_ABSOLUTE);
            cvalid.Set(range.End(),TIME_PosInfinity);
        }
        break;

    case ORT_LOOP:
    case ORT_CYCLE:
        GetValueLocalTime(CycleTime(range,t),ptr,
            cvalid,CTRL_ABSOLUTE);
        break;

    case ORT_OSCILLATE: {
        int cycles = NumCycles(range,t);
        TimeValue tp = CycleTime(range,t);
        if (cycles&1) {
            tp = range.End()-(tp-range.Start());
        }
    }
    }
}

```

```

    GetValueLocalTime(tp,ptr,cvalid,CTRL_ABSOLUTE);
    break;
}

case ORT_RELATIVE_REPEAT:
case ORT_IDENTITY:
case ORT_LINEAR:
    Extrapolate(range,t,ptr,cvalid,ort);
    break;
}
}

if (m!=1.0f) {
    MultiplyValue(ptr,m);
}
if (method==CTRL_RELATIVE) {
    ApplyValue(val,ptr);
    DeleteTempValue(ptr);
}

if (ort!=ORT_CONSTANT) {
    cvalid.Set(oldTime,oldTime);
}
valid &= cvalid;
valid &= wvalid;
// Time warps can cause this to happen.
if (valid.Empty()) valid.Set(oldTime,oldTime);
}

```

**Prototype:**

```

virtual void SetValue(TimeValue t, void *val, int commit=1,
    GetSetMethod method=CTRL_ABSOLUTE)=0;

```

**Remarks:**

Implemented by the Plug-In.

This method sets the value of the controller at the specified time. This method is responsible for handling Out of Range Types, Ease Curves and Multiplier

Curves. See the sample code below.

Note: Developers who want to create keys for a keyframe controller by calling **SetValue()** directly can do so, but the animate button should be turned on using the following code:

```
SuspendAnimate();  
AnimateOn();  
// Call SetValue() -- make sure commit=1  
ResumeAnimate();
```

#### Parameters:

##### **TimeValue t**

Specifies the time to save the value.

##### **void \*val**

Points to an instance of a data type that corresponds with the controller type. These are the same as **GetValue()** above with the following exceptions:

For rotation controllers, if the **GetSetMethod** is **CTRL\_RELATIVE**, **\*val** points to an **AngAxis**, while if it is **CTRL\_ABSOLUTE** it points to a **Quat**.

For **Matrix3** controllers **\*val** points to an instance of class **SetXFormPacket**. See [Class SetXFormPacket](#).

##### **int commit=1**

When **SetValue()** is called the controller should store its value (usually into a cache it maintains), and if this parameter is set, also 'commit' its value (usually by calling **CommitValue()**).

For example, consider a 3ds max keyframe controller: If **commit==1** and if the Animate button is on, then the cache should be updated and a key should be created. If the Animate button is off then the cache should be updated and the keys should all be offset. If **commit==0** then the cache value is set and its validity interval is set to the current time. If later **commit==1** then a key would be created from that cached value. If **SetValue()** is never called with **commit=1** then the key is never set. For instance with Inverse Kinematics, **SetValue()** is called many times over and over at the same **TimeValue** with **commit=0**. The controller doesn't create a key, it just changes its cached value. When an IK solution is finally reached, **SetValue()** is called with

**commit=1** and a key is created.

Note that calling **SetValue()** with **commit=0** and then calling **CommitValue()** should have the same effect as calling **SetValue()** with **commit=1**.

See the methods **CommitValue()** and **RestoreValue()** below.

**GetSetMethod method=CTRL\_ABSOLUTE**

One of the following values:

**CTRL\_RELATIVE**

Indicates the plug-in should add the value to the existing value **\*val** (i.e. Move/Rotate/Scale)

**CTRL\_ABSOLUTE**

Indicates the plug-in should just set the value.

**Important Note for Matrix3 Controllers:** When **SetValue()** is called this **method** parameter is ignored. The **\*val** pointer passed to **SetValue()** points to an instance of [Class SetXFormPacket](#).

See that class for more details on how it is used.

#### Sample Code:

The following code is from the StdControl implementation of this method. It demonstrates how the out of range and multiplier curves are handled.

```
void StdControl::SetValue(TimeValue t, void *val, int commit,
GetSetMethod method)
{
    Interval range = GetTimeRange(TIMERANGE_ALL);
    Interval wvalid, mvalid;
    int ort;
    float m;

    // Grab the multiplier before the time warp.
    m = GetMultVal(t,mvalid);
    if (m!=1.0f && m!=0.0f) {
        MultiplyValue(val,1.0f/m);
    }

    // Apply the time warp.
```

```

t = ApplyEase(t,wvalid);
if (range.Empty()) {
    SetValueLocalTime(t,val,commit,method);
    return;
}

if (t<=range.Start()) {
    ort = GetORT(ORT_BEFORE);
} else {
    ort = GetORT(ORT_AFTER);
}
if (TestAFlag(A_ORT_DISABLED)) ort = ORT_CONSTANT;

switch (ort) {
case ORT_LOOP:
case ORT_CYCLE:
    SetValueLocalTime(CycleTime(range,t),val,commit,method);
    break;

case ORT_OSCILLATE: {
    int cycles = NumCycles(range,t);
    if (cycles&1) {
        t = range.Duration() - CycleTime(range,t);
    } else {
        t = CycleTime(range,t);
    }
    SetValueLocalTime(t,val,commit,method);
    break;
}

// These ORTs aren't cyclic so we just set the value out of range.
case ORT_RELATIVE_REPEAT:
case ORT_CONSTANT:
case ORT_IDENTITY:
case ORT_LINEAR:
    SetValueLocalTime(t,val,commit,method);
    break;
}

```

```
}  
}
```

**Prototype:**

```
virtual bool GetLocalTMComponents(TimeValue t,  
TMComponentsArg& cmpts, Matrix3Indirect& parentMatrix);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the PRS components of the local matrix. In general, controller cannot decide on the local matrix without knowing the parent matrix. However, many controllers, such as default controllers, are well defined without the parent matrix. In these cases, it is more efficient to compute the local components directly without going through the world matrix.

Therefore, the argument `parentMatrix` is a reference to `Matrix3Indirect`. This would allow clients to supply a "delayed parent matrix," which will be computed only if it is necessary. It returns true for Matrix3, Position, Rotation, or Scale controllers, and return false otherwise.

The PRS components will be put in argument `cmpts` in the respective fields with corresponding validity intervals. NULL pointer, of `TMComponentsArg::position` for example, indicates that the client is not concerned about the component. When it is not NULL, the corresponding pointer to the validity interval MUST NOT be NULL. When it is not NULL, `TMComponentsArg::rotation` is a float[4]. `rotRep` tells what the numbers mean.

Position, Rotation, or Scale, controllers will put results at the respective component when the corresponding pointer is not NULL.

Upon entry, `parentMatrix` should represent the parent matrix up to the first requested components. For Matrix3 controllers, for example, if `cmpts.position==NULL && cmpts.rotation!=NULL`, then `parentMatrix` should be matrix that includes the parent node matrix plus the position of this node.

Upon return, this matrix may be modified.

**Parameters:****TimeValue t**

The time at which to get the local TM components.

## TMComponentsArg& cmpts

See [Structure TMComponentsArgs](#).

## Matrix3Indirect& parentMatrix

The parent matrix. Note the definition:  $\text{LocalMatrix} = \text{WorldMatrix} * \text{ParentWorldMatrix}^{-1}$

## Post Clone Notification

### Prototype:

**virtual void PostCloneNode();**

### Remarks:

This method is available in release 2.0 and later only.

This method is called on a transform controller after a node is cloned and the clone process has finished. This allows the controller to do any work it needs to after the clone is complete.

### Default Implementation:

```
{}
```

## Undo / Redo methods

Controllers, like objects and modifiers, need to be able to undo and redo themselves. Whenever a controller is about to modify its data, it checks the state of the global Hold object to see if it is holding. If so it must register a RestoreObject with the hold (see the Advanced Topics section [Undo/Redo](#)). Controllers also support another type of undo and redo through two methods: **CommitValue()** and **RestoreValue()**.

The purpose of this 'inner' hold and restore buffer is not to hold and restore the entire state of the controller, but to hold and restore the value of the controller at a single instant in time. When **SetValue()** is called on a controller with the **commit** parameter equal to zero, the controller records the new value, but does not necessarily modify any data. For example, a keyframe controller doesn't actually generate a new key -- it just updates a cache it maintains. Then, if the controller is asked to evaluate itself at the exact same **TimeValue** for which the controller was just set, it can just return the cached value.

The **RestoreValue()** method will simply throw out the temporary value

whereas the **CommitValue()** method will cause the value to be actually committed (a key generated in the case of a keyframe controller).

The purpose of this inner hold and restore is for iterative procedures that need to set values many times at a single **TimeValue** and don't want to incur the overhead of things like recalculating the tangents at adjacent keys. Some examples of these types of procedures are inverse kinematics and collision detection.

**Prototype:**

**virtual void CommitValue(TimeValue t)**

**Remarks:**

Implemented by the Plug-In.

This method, along with **RestoreValue()**, comprise an "inner" hold and restore mechanism (see above). When the controller's **SetValue()** method is called, if the **commit** parameter is nonzero, then the controller should save the value of the controller at the current time into its cache and also 'commit' the value. For example, this stores a key in the case of a keyframe controller. If the set value was not committed then **RestoreValue()** may be called to restore the previous value.

**Parameters:**

**TimeValue t**

Specifies the time to save the value.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void RestoreValue(TimeValue t)**

**Remarks:**

Implemented by the Plug-In.

This method is the other half of the "inner" hold and restore mechanism. This method is called to restore a previously saved value. This method restores the current cache value to the value that was set before **SetValue()** was last called. The way the standard 3ds max controllers handle this is as follows: When **SetValue()** is called a temporary hold mechanism (**TempStore** define

in **CONTROL.H**) is used to hold the current value. Then the new value is set. If **RestoreValue()** is later called then it restores the current value from the temporary storage. Note that in addition to restoring from the **TempStore**, another way a controller may restore the current value is to re-interpolate the keys.

**Parameters:**

**TimeValue t**

Specifies the time to restore the value.

**Default Implementation:**

**{**

## **New Controller Assignment**

**Prototype:**

**virtual void Copy(Control \*from)=0;**

**Remarks:**

Implemented by the Plug-In.

When a controller is assigned to a track in the track view, the new controller is plugged into the parameter and this method is called on the new controller. A pointer to the old controller is passed in to this method. The new controller can attempt to copy any data that it can from the old controller. At the very least it should initialize itself to the value of the old controller at frame 0.

**Parameters:**

**Control \*from**

A pointer to the previous controller.

**Prototype:**

**virtual BOOL IsReplaceable()**

**Remarks:**

Implemented by the Plug-In.

This method determines if another controller can replace this one. A controller can return FALSE from this method to not allow the user to assign a new controller in its place. This will also prevent the controller from being replaced

by a paste controller operation.

**Return Value:**

TRUE to allow the controller to be replaced; otherwise FALSE.

**Default Implementation:**

```
{return TRUE;}
```

## **Lookat Controller methods**

The following two methods work with lookat controllers. If the controller is not a lookat controller, these methods should be ignored.

**Prototype:**

```
virtual INode* GetTarget()
```

**Remarks:**

Implemented by the Plug-In.

This method retrieves a lookat controller's target.

**Return Value:**

The lookat controllers target node.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual RefResult SetTarget(INode *targ)
```

**Remarks:**

Implemented by the Plug-In.

This method stores a lookat controller's target.

**Parameters:**

**INode \*targ**

The target node to store.

**Return Value:**

One of the following values:

**REF\_SUCCEED**

Indicates the target was set.

## **REF\_FAIL**

Indicates the target was not set.

### **Default Implementation:**

```
{return REF_SUCCEED;}
```

## **Sub-Controllers**

### **Prototype:**

```
virtual Control *GetPositionController()
```

### **Remarks:**

Implemented by the Plug-In.

Implemented by transform controllers that have a position controller that can be edited in the motion branch. This method returns a pointer to the position controller of the transform controller.

### **Default Implementation:**

```
{return NULL;}
```

### **Prototype:**

```
virtual Control *GetRotationController()
```

### **Remarks:**

Implemented by the Plug-In.

Implemented by transform controllers that have a rotation controller that can be edited in the motion branch. This method returns a pointer to the rotation controller of the transform controller.

### **Return Value:**

### **Default Implementation:**

```
{return NULL;}
```

### **Prototype:**

```
virtual Control *GetScaleController()
```

### **Remarks:**

Implemented by the Plug-In.

Implemented by transform controllers that have a scale controller that can be edited in the motion branch. This method returns the a pointer to the scale controller of the transform controller.

**Return Value:**

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual Control *GetRollController()
```

**Remarks:**

Implemented by the Plug-In.

Implemented by lookat controllers that have a float valued roll controller so that the roll can be edited via the transform type-in. This method returns a pointer to the roll controller of the lookat controller.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual BOOL SetPositionController(Control *c);
```

**Remarks:**

Implemented by the Plug-In.

This method assigns a new position controller. Plug-Ins don't need to be concerned with freeing the previous controller if this method is called. Any previous controller assigned will be deleted by 3ds max if it is not used elsewhere in the scene.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL SetRotationController(Control *c);
```

**Remarks:**

Implemented by the Plug-In.

This method assigns a new rotation controller. Plug-Ins don't need to be concerned with freeing the previous controller if this method is called. Any previous controller assigned will be deleted by 3ds max if it is not used elsewhere in the scene.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL SetScaleController(Control *c);
```

**Remarks:**

Implemented by the Plug-In.

This method assigns a new scale controller. Plug-Ins don't need to be concerned with freeing the previous controller if this method is called. Any previous controller assigned will be deleted by 3ds max if it is not used elsewhere in the scene.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL SetRollController(Control *c);
```

**Remarks:**

Implemented by the Plug-In.

This method assigns a new roll controller. Plug-Ins don't need to be concerned with freeing the previous controller if this method is called. Any previous controller assigned will be deleted by 3ds max if it is not used elsewhere in the scene.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual Control *GetXController();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the 'X' sub-controller of this controller. If a controller has an 'X', 'Y', or 'Z' controller, it can implement this set of methods so that its sub-controllers can respect track view filters. Examples of controllers that have XYZ sub-controllers are the Euler angle controller or the Position XYZ controller.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual Control *GetYController();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the 'Y' sub-controller of this controller. If a controller has an 'X', 'Y', or 'Z' controller, it can implement this set of methods so that its sub-controllers can respect track view filters. Examples of controllers that have XYZ sub-controllers are the Euler angle controller or the Position XYZ controller.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual Control *GetZController();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the 'Z' sub-controller of this controller. If a controller has an 'X', 'Y', or 'Z' controller, it can implement this set of methods so that its sub-controllers can respect track view filters. Examples of controllers that have XYZ sub-controllers are the Euler angle controller or the Position XYZ controller.

**Default Implementation:**

```
{return NULL;}
```

## Transform Controller methods

### Prototype:

**virtual BOOL ChangeParents(TimeValue t,const Matrix3& oldP,  
const Matrix3& newP,const Matrix3& tm)**

### Remarks:

Implemented by the Plug-In.

This method is called on transform, position, rotation, and scale controllers when their input matrix is about to change. This happens when the user links an object (either from one object to another or when the user links an object for the first time). Because a controller's transformation is relative to its parent, when the user changes parents, the transform controller will need to change itself. If a plug-in returns FALSE the node will calculate a change and call **SetValue()** to make the necessary adjustments at the specific time passed.

Consider the following example of a position controller:

If a node in the scene that is NOT animated, is linked to another node, this method would be called. If the method returned FALSE then the node would calculate a change and call **SetValue()** to make the adjustment and this would be okay. If however the node was animated there would be a problem. Say for example that an unlinked node was bouncing up and down along the world Z axis. If this node is then linked to a node that was rotated such that its Z axis was pointed in the direction of the world X axis (so the object is flipped over on its side) the linked node (whose animation keys are stored relative to its previous parent (the world)) would then begin to bounce up and down along the world X axis instead. This is because it is still moving along its parent's Z axis, but its parent's Z axis is really the world X axis. Thus the object needs to be counter-rotated to compensate. Additionally, all the animation keys for the object also need to be counter-rotated. A position keyframe controller would need to implement this method to handle the correction of the object and its keyframes. See the sample code below.

### Parameters:

#### **TimeValue t**

The time of the change.

#### **const Matrix3& oldP**

The old parent matrix.

**const Matrix3& newP**

The new parent matrix.

**const Matrix3& tm**

The nodes current world transformation.

**Return Value:**

If FALSE the node will call **SetValue()** to make the necessary adjustments.

**Default Implementation:**

```
{return FALSE;}
```

**Sample Code:**

This is the code used inside 3ds max' position controller. It takes the difference between the two parents and transforms the position track by that amount. It computes the relative transformation which is the old parent times the inverse of the new parent.

A plug-in could provide an implementation for this method using a similar concept.

**INTERP\_CONT\_TEMPLATE**

**BOOL**

```
InterpControl<INTERP_CONT_PARAMS>::ChangeParents(  
    TimeValue t,  
    const Matrix3& oldP,const Matrix3& newP,const Matrix3& tm)  
{  
    if (SuperClassID()==CTRL_POSITION_CLASS_ID) {  
        HoldTrack();  
        // Position controllers need their path counter rotated to  
        // account for the new parent.  
        Matrix3 rel = oldP * Inverse(newP);  
        // Modify the controllers current value (the controllers cache)  
        *((Point3*)&curval) = *((Point3*)&curval) * rel;  
        // Then modify the keys...  
        for (int i=0; i<keys.Count(); i++) {  
            // All this casting keeps the compiler happy  
            // for non-Point3 versions of this template.  
            *((Point3*)&keys[i].val) =
```

```
        *((Point3*)&keys[i].val) * rel;
    }
    keys.KeysChanged(FALSE);
    ivalid.SetEmpty();
    return TRUE;
} else {
    return FALSE;
}
}
```

**Prototype:**

```
virtual RefResult PinNodeChanged(RefMessage message,Interval
changeInt, PartID &partID);
```

**Remarks:**

This method is available in release 2.0 and later only.

If a node is pinned to another node, and the node gets a **NotifyRefChanged()** message that its pinned node has changed, then this method is called on the transform controller of the node. Otherwise the controller wouldn't get notified since the controller doesn't have a reference to the pin node (but the node does). Most controllers don't really care, but the IK controller does.

**Parameters:**

**RefMessage message**

The message that was sent.

**Interval changeInt**

This is the interval of time over which the message is active. Currently, all controllers will receive **FOREVER** for this interval.

**PartID &partID**

This contains information specific to the message passed in. Some messages don't use the **partID** at all. See the section [List of Reference Messages](#) for more information about the meaning of the **partID** for some common messages.

**Return Value:**

The return value from this method is of type **RefResult**. This is usually **REF\_SUCCEED** indicating the message was processed. Sometimes, the return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

**Default Implementation:**

```
{return REF_SUCCEED;}
```

**Prototype:**

```
virtual void NodeIKParamsChanged();
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called on a transform controller when one of the node level IK parameters has been changed.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL InheritsParentTransform()
```

**Remarks:**

Implemented by the Plug-In.

This method is only implemented by transform controllers. Transform controllers that do not inherit their parent's transform should override this method.

When a transform controller is evaluated, the parent transform is passed in to the controller and the controller typically applies its value to the parent transform. However, some controllers (for example Biped) may choose to control the TM in an absolute manner and therefore ignore the incoming parent's TM. The system needs to know about this because normally if an object and its parent are selected and the user attempts to move them only the parent transform is modified because it is assumed that the child will inherit its parents TM.

Note: This method may still return TRUE even if all the bits returned from **GetInheritanceFlags()** are SET to indicate that nothing is inherited from

the parent. This is simply because these methods don't have the same level of 'granularity'. This method deals with the overall inheritance of the parent's transform whereas the inheritance flags relate to individual parts.

**Return Value:**

TRUE if the controller inherits its parents TM; otherwise FALSE. Returning FALSE will cause **SetValue()** to be called even in the case when the parent is also being transformed.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual DWORD GetInheritanceFlags();
```

**Remarks:**

Implemented by the Plug-In.

This method should be implemented by TM controllers that support filtering out inheritance. It returns the state of the transform inheritance flags. These are the values that show up in the Hierarchy branch, under the Link Info section, in the Inheritance rollup.

**Return Value:**

One or more of the following values:

Note: Each bit is used to represent a single inheritance. If the bit is **CLEAR (OFF)** it means inherit (checked in the 3ds max UI). If the bit is **SET** it means **DON'T** inherit (unchecked in the 3ds max UI).

- INHERIT\_POS\_X**
- INHERIT\_POS\_Y**
- INHERIT\_POS\_Z**
- INHERIT\_ROT\_X**
- INHERIT\_ROT\_Y**
- INHERIT\_ROT\_Z**
- INHERIT\_SCL\_X**
- INHERIT\_SCL\_Y**
- INHERIT\_SCL\_Z**
- INHERIT\_ALL**

**Default Implementation:**

```
{return INHERIT_ALL;}
```

**Prototype:**

```
virtual BOOL SetInheritanceFlags(DWORD f, BOOL keepPos);
```

**Remarks:**

Implemented by the Plug-In.

This method should be implemented by TM controllers that support filtering out inheritance.

Note: Each bit is used to represent a single inheritance. This method expects the bits of the flags passed to be **CLEAR (OFF)** to mean DON'T inherit (unchecked in the 3ds max UI). If they are **SET** it means inherit (checked in the 3ds max UI).

**Parameters:****DWORD f**

The inheritance flags. One or more of the following values:

```
INHERIT_POS_X  
INHERIT_POS_Y  
INHERIT_POS_Z  
INHERIT_ROT_X  
INHERIT_ROT_Y  
INHERIT_ROT_Z  
INHERIT_SCL_X  
INHERIT_SCL_Y  
INHERIT_SCL_Z  
INHERIT_ALL
```

**BOOL keepPos**

If TRUE the position of the node should remain the same; otherwise the node may move.

**Return Value:**

Return TRUE if TM controller supports inheritance; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

## **IsLeaf / IsKeyable / IsColor**

### **Prototype:**

**virtual BOOL IsLeaf()**

### **Remarks:**

Implemented by the Plug-In.

Indicates whether the controller is a leaf controller. If a controller is a leaf controller, then it **MUST NOT BY DEFINITION** have any sub-controllers or references. The controller should return **TRUE** if it has no sub-controllers. For example, a PRS controller is not a leaf controller (because it has sub-controllers for Position, Rotation and Scale), but a simple keyframed float controller is a leaf controller.

### **Return Value:**

**TRUE** if the controller is a leaf controller; **FALSE** otherwise.

### **Default Implementation:**

```
{return TRUE;}
```

### **Prototype:**

**virtual int IsKeyable()**

### **Remarks:**

Implemented by the Plug-In.

Indicates if the controller is a keyframe controller. This means the controller stores keys at certain frames and interpolates between keys at other times.

### **Return Value:**

Nonzero if the controller is a keyframe controller; zero otherwise.

### **Default Implementation:**

```
{return 1;}
```

### **Prototype:**

**virtual BOOL IsColorController()**

### **Remarks:**

Implemented by the Plug-In.

Implemented by any Point3 controller that wishes to indicate that it is intended to control floating point RGB color values. Returns TRUE to indicate that it controls float color values; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Ease / Multiplier Curve and ORT methods**

The following methods involve ease and multiplier curves. See the sample code after the methods **GetValue()** and **SetValue()** to see how these methods are used.

**Prototype:**

```
virtual BOOL CanApplyEaseMultCurves()
```

**Remarks:**

Implemented by the Plug-In.

This method determines if a controller may have ease or multiplier curves applied to it. This method defaults to returning TRUE, but can be implemented to return FALSE by a controller that does not wish to let ease or multiplier curves be applied to it.

**Return Value:**

TRUE to allow the application of ease and multiplier curves; otherwise FALSE.

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
TimeValue ApplyEase(TimeValue t,Interval &valid);
```

**Remarks:**

Implemented by the System.

The controller calls this method to pipe the TimeValue passed through the ease curve to get the modified TimeValue.

**Parameters:**

**TimeValue t**

The time to have modified by the ease curve.

**Interval &valid**

The validity interval of the TimeValue returned.

**Return Value:**

The modified TimeValue.

**Prototype:**

**void AppendEaseCurve(Control \*cont);**

**Remarks:**

Implemented by the System.

Adds an ease curve to the specified controller.

**Parameters:**

**Control \*cont**

The controller that the ease curve will be applied to.

**Prototype:**

**int NumEaseCurves();**

**Remarks:**

Implemented by the System.

Returns the number of ease curves applied to the controller.

**Prototype:**

**void DeleteEaseCurve(int i);**

**Remarks:**

Implemented by the System.

Deletes the 'i-th' ease curve from the controller.

**Parameters:**

**int i**

The index of the ease curve to delete.

**Prototype:**

**float GetMultVal(TimeValue t,Interval &valid);**

**Remarks:**

Implemented by the System.

Retrieves a floating point value that is the product of all the multiplier curves at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the

**Interval &valid**

The validity interval of the value.

**Return Value:**

The product of all the multiplier curves applied to the controller.

**Prototype:**

**void AppendMultCurve(Control \*cont);**

**Remarks:**

Implemented by the System.

Adds a multiplier curve to the specified controller.

**Parameters:**

**Control \*cont**

The controller to have the multiplier curve added.

**Prototype:**

**int NumMultCurves();**

**Remarks:**

Implemented by the System.

Returns the number of multiplier curves assigned to the controller.

**Prototype:**

**void DeleteMultCurve(int i);**

**Remarks:**

Implemented by the System.

Deletes the 'i-th' multiplier curve from this controller.

**Parameters:**

**int i**

The index of the curve to delete.

**When a user brings up the track view and sets the out of range types for the controller the following two methods are called.**

**Prototype:**

**virtual int GetORT(int type)**

**Remarks:**

Implemented by the System.

Returns the specified Out of Range Type used by the controller. The system handles this method but the controller needs to process the ORT in its implementation of **GetValue()** and **SetValue()**.

**Parameters:**

**int type**

One of the following values:

**ORT\_BEFORE** - leading up to the pattern

**ORT\_AFTER** - beyond the key pattern

**Return Value:**

One of the following values:

See [List of Out of Range Types](#).

**Prototype:**

**virtual void SetORT(int ort,int type);**

**Remarks:**

Implemented by the System.

Sets the specified Out of Range Type to be used by the controller. The system handles this method but the controller needs to process the ORT in its implementation of **GetValue()** and **SetValue()**.

**Parameters:**

**int ort**

See [List of Out of Range Types](#).

**int type**

One of the following values:

**ORT\_BEFORE** - leading up to the pattern

**ORT\_AFTER** - beyond the key pattern

**Prototype:**

**virtual void EnableORTs(BOOL enable);**

**Remarks:**

Implemented by the System.

Sets the enabled/disabled state for Out of Range Types. If disabled, this temporarily causes the Out of Range Types to behave as if set to constant. This can be used if you want to modify a controller but don't want ORT mapping for ORT\_LOOP, ORT\_CYCLE, or ORT\_OSCILLATE.

**Parameters:**

**BOOL enable**

TRUE to enable ORTs; FALSE to disable.

**The methods below are used with Inverse Kinematics. The following is an overview of the control flow of the system calling these IK methods on the controller during IK calculations:**

First the system calls **EnumIKParams()** once to get the number of IK parameters (degrees of freedom) from the plug-in. This lets the system know how many parameters the controller has. This happens only once when the entire chain is set up. For example during interactive IK, when the user presses the mouse button down the chain is set up. In applied IK when the user presses the Apply IK button the chain is set up.

Next the IK task is defined. For example, if the user is doing interactive IK they press the mouse button down and the IK chain is set up. Next they move the mouse a few pixels. This results in a delta for the end effector. This defines the task for the end effector. The end effector wants to move some delta or rotate some amount. This has defined a task to be solved.

In order to solve the task the system must iterate towards a solution.

To do this it first needs to compute the derivatives of all the parameters. The system calls **CompDerivs()** on the controller. The controller computes the derivative and tells the system by calling methods on the **IKDeriv** argument passed to the method. It provides its derivative for each of the parameters.

Next the system uses this derivative information to formulate a change in the parameter that will get closer to the solution. The system then computes an amount to change the parameter and then calls **IncIKParams()**. The delta passed to this method is the amount of change it wants in the parameter. The controller then applies this change (perhaps not all of it due to constraints it might have) and returns the amount that was applied.

This has completed a single iteration. All the parameters have been adjusted a bit and hopefully we have move closer to a solution. At this point the system calls **CompDerivs()** again because just moving the parameters a small amount may have actually changed the derivative. The cycle begins again.

When the IK solver has reached its solution the IK task is finished. Below are the methods that must be implemented to participate in this process.

**Prototype:**

**virtual void EnumIKParams(IKEnumCallback &callback)**

**Remarks:**

Implemented by the Plug-In.

This tells the system how many parameters the controller has. A controller can have as many IK parameters as it wants. An IK parameter corresponds to a degree of freedom in IK. The parameter is a floating point scalar value. For example a position controller has three degrees of freedom (X, Y, Z) and thus three parameters that IK can vary in its solution. The path controller has only a single parameter (degree of freedom) - the position along the path. The 3ds max user may set the number of degrees of freedom. For example, a user can specify that a rotation controller cannot rotate about one or more axes. These are then no longer degrees of freedom or IK parameters.

This method is called by the system so the plug-in can specify how many IK parameters it has. It does this by calling the provided callback object **proc()** method once for each parameter it has. It passes a pointer to itself and the index of the IK parameter. For example a position controller with three degrees of freedom (and thus three IK parameters) would call the **callback.proc()** three time passing an index of 0, then 1, then 2. See the sample code below.

**Parameters:**

**IKEnumCallback &callback**

This callback is provided by the system and should be called by the plug-in once for each IK parameter the plug-in has. See [Class IKEnumCallback](#).

**Default Implementation:**

```
{
```

**Sample Code:**

```
void QuatEnumIKParams(Control *cont,IKEnumCallback
&callback)
{
    JointParams *jp = (JointParams*)cont-
>GetProperty(PROPID_JOINTPARAMS);
    for (int i=0; i<3; i++) {
        if (!jp || jp->Active(i)) {
            callback.proc(cont,i);
        }
    }
}
```

**Prototype:**

```
virtual BOOL CompDeriv(TimeValue t,Matrix3& ptm,
    IKDeriv& derivs,DWORD flags)
```

**Remarks:**

Implemented by the Plug-In.

This method is used to determine what effect a change in the parameter has on the end effector. This is the derivative of the end effector with respect to the parameter. What the derivative means in this case is what happens to the end effector if the parameter is changed by some small delta.

The plug-in provides the derivatives to the system calling **derivs.DP()** and **derivs.DR()**. It should call **derivs.DP()** and **derivs.DR()** in the same order as the **callback.proc()** was called in the **NumIKParams()** method implementation.

When the controller computes the derivative it should apply itself to the parent matrix. For example a position controller would compute its derivative based on the parent and the position of the end effector and then apply itself to the

parent matrix. If it does apply itself to the parent it should return TRUE. If it does not apply itself it should return FALSE.

**Parameters:**

**TimeValue t**

Specifies the time to compute the derivative.

**Matrix3& ptm**

The parents transformation.

**IKDeriv& derivs**

This class provides methods the plug-in calls to set the derivatives. See [Class IKDeriv](#).

**DWORD flags**

One of the following values:

**POSITION\_DERIV**

Indicates that **derivs.DP()** should be called.

**ROTATION\_DERIV**

Indicates that **derivs.DR()** should be called.

**Return Value:**

If a controller isn't participating in IK then it should return FALSE and the client (usually PRS) will apply the controller's value to the parent TM.

**Default Implementation:**

```
{return FALSE;}
```

**Sample Code:**

The following sample code shows how the quaternion controller has implemented this method. Note that the method loops based on the number of end effectors, and calls **derivs.NextDOF()** after each iteration.

```
BOOL QuatCompDeriv(Control *cont,TimeValue t,Matrix3& ptm,
```

```
IKDeriv& derivs,DWORD flags)
```

```
{
```

```
JointParams *jp = (JointParams*)cont-
```

```
>GetProperty(PROPID_JOINTPARAMS);
```

```
Quat q;
```

```
Interval valid;
```

```

for (int i=0; i<3; i++) {
  if (!jp || jp->Active(i)) {
    for (int j=0; j<derivs.NumEndEffectors(); j++) {
      Point3 r = derivs.EndEffectorPos(j) - ptm.GetRow(3);
      if (flags&POSITION_DERIV) {
        derivs.DP(CrossProd(ptm.GetRow(i),r),j);
      }
      if (flags&ROTATION_DERIV) {
        derivs.DR(ptm.GetRow(i),j);
      }
    }
    derivs.NextDOF();
  }
}
return FALSE;
}

```

**Prototype:**

**virtual float IncIKParam(TimeValue t,int index,float delta);**

**Remarks:**

Implemented by the Plug-In.

When the system has computed a change in the parameter it will call this method. The controller should increment the specified parameter by the specified delta. The controller can increment the parameter less than this delta if it needs to. This could be for several reasons:

- 1 Its parameter may be constrained to lie within a specific interval. It would not want to add a delta that took the parameter outside of this interval.
- 2 It was asked to calculate a constant partial derivative for a linkage that could be nonlinear. Therefore the derivative may have only been an instantaneous approximation. Due to the locality of the IK solution, the controller might not want to allow a delta that was too large.

After the controller has applied the delta, it needs to indicate to the system how much of the delta was used.

**Parameters:**

**TimeValue t**

The time of the increment.

**int index**

Specifies the IK parameter to increment.

**float delta**

The delta to apply to the parameter. The controller can increment the parameter less than this delta if it needs to in order to accommodate a limit it has. This methods returns the amount that was actually incremented.

**Return Value:**

The amount the parameter was actually incremented. This allows the IK solver to know the value was not incremented the full amount.

**Default Implementation:**

```
{return 0.0f;}
```

**Sample Code:**

```
float QuatIncIKParam(Control *cont,TimeValue t,int index,float
delta)
{
    JointParams *jp = (JointParams*)cont-
>GetProperty(PROPID_JOINTPARAMS);
    if ((float)fabs(delta)>MAX_IKROT) delta = MAX_IKROT *
SGN(delta);
    if (jp) {
        float v=0.0f;
        if (jp->Limited(index)) {
            Quat q;
            Interval valid;
            cont->GetValue(t,&q,valid,CTRL_ABSOLUTE);
            v = GetRotation(q,index);
        }
        delta = jp->ConstrainInc(index,v,delta);
    }
    Point3 a(0,0,0);
    a[index] = 1.0f;
    AngAxis aa(a,-delta);
```

```

cont->SetValue(t,&aa,FALSE,CTRL_RELATIVE);
return delta;
}
static float GetRotation(Quat& q,int axis)
{
Matrix3 tm;
q.MakeMatrix(tm);
MRow* t = tm.GetAddr();
int n = (axis+1)%3, nn = (axis+2)%3;
if (fabs(t[n][axis]) < fabs(t[nn][axis])) {
return (float)atan2(t[n][nn],t[n][n]);
} else {
return -(float)atan2(t[nn][n],t[nn][nn]);
}
}
}

```

**Prototype:**

**virtual void ClearIKParam(Interval iv,int index)**

**Remarks:**

Implemented by the Plug-In.

This method is called to have the controller delete its keys. If the user has the 'Clear Keys' check box checked when they press the 'Apply IK' button, this method is called to have the controller deletes keys in the given interval for the specified degree of freedom.

**Parameters:**

**Interval iv**

The interval over which the keys should be deleted.

**int index**

Specified the degree of freedom (parameter) that the keys should be deleted for.

**Default Implementation:**

**{return;}**

**Prototype:**

**virtual void InitIKJoints(InitJointData \*posData,InitJointData \*rotData)**

**Remarks:**

Implemented by the Plug-In.

This is an optional method that can be implemented by controllers that support IK to initialize their joint parameters based on data loaded from 3D Studio R4/DOS files.

**Parameters:**

**InitJointData \*posData**

The position data from the 3DS file. See [Class InitJointData](#).

**InitJointData \*rotData**

The rotation data from the 3DS file.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual void InitIKJoints2(InitJointData2 \*posData, InitJointData2 \*rotData);**

**Remarks:**

Implemented by the Plug-In.

This method is available in release 4.0 and later only.

This is an optional method that can be implemented by controllers that support IK to initialize their joint parameters based on data loaded from 3D Studio R4/DOS files.

**Parameters:**

**InitJointData2 \*posData**

The position data from the 3DS file. See [Class InitJointData2](#).

**InitJointData2 \*rotData**

The rotation data from the 3DS file.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BOOL GetIKJoints(InitJointData *posData,InitJointData *rotData);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method retrieves the IK joint parameter data from the UI.

**Parameters:**

**InitJointData \*posData**

Points to the object to hold the position data. See [Class InitJointData](#).

**InitJointData \*rotData**

Points to the object to hold the rotation data.

**Return Value:**

TRUE if the data was retrieved; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL GetIKJoints2(InitJointData2 *posData, InitJointData2 *rotData);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method retrieves the IK joint parameter data from the UI.

**Parameters:**

**InitJointData2 \*posData**

Points to the object to hold the position data. See [Class InitJointData2](#).

**InitJointData2 \*rotData**

Points to the object to hold the rotation data.

**Return Value:**

TRUE if the data was retrieved; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

**virtual BOOL GetDOFParams(TimeValue t, Matrix3 &ptm,  
DOFParams &dofs, BOOL nodeSel);**

**Remarks:**

This method is available in release 2.0 and later only.

The new IK system has some axes gizmos which show the degrees of freedom, etc. This method is called by the system

**Parameters:**

**TimeValue t**

The current time.

**Matrix3 &ptm**

The parent matrix.

**DOFParams &dofs**

This is the structure to be filled in. See [Class DOFParams](#).

**BOOL nodeSel**

TRUE if the node is currently selected; otherwise FALSE.

**Return Value:**

TRUE if the method is implemented; FALSE otherwise.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

**virtual BOOL CreateLockKey(TimeValue t, int which);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to create a locking key. This is a key that looks back to the previous key and creates a new key at the specified time which matches the previous key in value. It also adjusts the parameters for the key such that the value stays constant from the previous key to this key. For instance, the TCB controller will set the previous and new continuity to 0. The Bezier controller sets the out tangent type of the previous key to linear and the in tangent type of the new key to linear.

**Parameters:****TimeValue t**

The time to create the key.

**int which**

Specifies which type of key to create: **0** for position, **1** for rotation.

**Return Value:**

TRUE if the method is implemented; FALSE otherwise.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void MirrorIKConstraints(int axis, int which, BOOL  
pasteMirror=FALSE);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to mirror the specified IK constraints about the specified axis. When IK constraints are mirrored they need to be updated to reflect the new orientation. For instance, if you set the constraints for a left arm to bend only +90 degrees along one axis and then copied these to a right arm the joint would bend backwards. What you need to do is provides the appropriate compensation so the orientation is kept proper.

**Parameters:****int axis**

Specifies the axis of reflection: **0** for X, **1** for Y, **2** for Z.

**int which**

Specifies which type of constraints are being mirrored: **0** for position, **1** for rotation.

**BOOL pasteMirror=FALSE**

TRUE if the mirror is being done as part of a paste operation; otherwise FALSE (for example if the mirror was being done with the mirror tool).

**Return Value:**

TRUE if the method is implemented; FALSE otherwise.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL TerminateIK();
```

**Remarks:**

This method is available in release 2.0 and later only.

User can specify a node as a terminator. This method gives the associated controller the chance to specify that it's terminated.

**Return Value:**

TRUE if the method is implemented; FALSE otherwise.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void NodeIKParamsChanged();
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called on a transform controller when one of the node level IK parameters has been changed

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void TMInvalidated();
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called in a transform controller when a node invalidates its TM cache

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL OKToBindToNode(INode *node);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method lets a TM controller determine if it's OK to IK bind to a particular node.

**Parameters:**

```
INode *node
```

Points to the node to check.

**Return Value:**

TRUE if it's okay to bind; FALSE if it's not.

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
virtual BOOL PreventNodeDeletion();
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called on TM controllers so that system slave controllers can prevent the **Interface::DeleteNode()** API from deleting them. Note that **DeleteNode()** has an optional parameter to override this so master controllers can easily delete slave nodes if they want to.

**Return Value:**

TRUE to prevent deletion; FALSE to allow it.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual float EvalVisibility(TimeValue t, View &view, Box3 pbox,  
Interval &valid);
```

**Remarks:**

This method is available in release 2.0 and later only.

The Level of Detail utility lets you construct an object that alters its geometric complexity (or level of detail) based on its size in the rendered image. You do this by creating several versions of the same object -- each with different levels of detail, grouping them as one, and then assigning the Level of Detail utility, which automatically creates a special LOD controller as a Visibility track. The LOD controller then hides and unhides the various objects in the group, depending on their size in the rendered scene.

This method is called on visibility float controllers with view related parameters. This is used by the Level of Detail controller to allow view dependent visibility.

**Parameters:**

**TimeValue t**

The time at which to evaluate.

**View &view**

This class contains information about the view being rendered. This includes information such as the image width and height, the projection type, and matrices to convert between world to view and world to camera. See [Class View](#).

**Box3 pbox**

The bounding box of the node that's being evaluated.

**Interval &valid**

This interval should be updated to reflect the validity of the visibility controller.

**Return Value:**

The visibility of the object at the specified time.

**Default Implementation:**

The default implementation will simply call **GetValue()**.

**Prototype:**

**virtual BOOL VisibleInViewports();**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called on visibility controllers. This gives them the option to

completely hide an object in the viewports.

**Return Value:**

TRUE if the object is visible in the viewport; FALSE if invisible.

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
virtual BOOL CanInstanceController();
```

**Remarks:**

This method is available in release 2.0 and later only.

Called on transform controllers or visibility controllers when a node is cloned and the user has chosen to instance

**Return Value:**

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
void CloneControl(Control *ctrl,RemapDir &remap);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is implemented by the System. It should be called by any leaf controller's **Clone()** method so that ease and multiplier curves are cloned.

**Parameters:**

**Control \*ctrl**

Points to the cloned controller (the new one).

**RemapDir &remap**

The **RemapDir** passed to this controller's **Clone()** method.

**Sample Code:**

```
RefTargetHandle Clone(RemapDir& remap=NoRemap()) {  
    ExprControl *ctrl = new ExprControl(this->type, *this);  
    CloneControl(ctrl,remap);  
}
```

```
return ctrl;  
}
```

The following functions are not part of class Control but are provided for use. These functions are for use by Position or Rotation controllers. Position controllers would use the Point3 versions and Rotation controllers would use the Quat versions

```
void QuatEnumIKParams(Control *cont,IKEnumCallback  
&callback);
```

```
BOOL QuatCompDeriv(Control *cont,TimeValue t,Matrix3&  
ptm,  
IKDeriv& derivs,DWORD flags);
```

```
float QuatIncIKParam(Control *cont,TimeValue t,int index,float  
delta);
```

```
void QuatBeginIKParams(Control *cont,IObjParam *ip,  
ULONG flags,  
Animatable *prev);
```

```
void Point3EnumIKParams(Control *cont,IKEnumCallback  
&callback);
```

```
BOOL Point3CompDeriv(Control *cont,TimeValue t,Matrix3&  
ptm,  
IKDeriv& derivs,DWORD flags);
```

```
float Point3IncIKParam(Control *cont,TimeValue t,int  
index,float delta);
```

```
void Point3BeginIKParams(Control *cont,IObjParam *ip,  
ULONG flags,Animatable *prev);
```

**The methods below deal with copying and pasting IK parameters in the Hierarchy branch.**

**Prototype:**

**virtual BOOL CanCopyIKParams(int which)**

**Remarks:**

Implemented by the Plug-In.

This method returns TRUE if the controller has IK parameters it can copy and FALSE otherwise.

**Parameters:**

**int which**

One of the following values:

**COPYPASTE\_IKPOS  
COPYPASTE\_IKROT**

**Return Value:**

TRUE if the controller can copy the specified IK parameters; otherwise FALSE.

**Default Implementation:**

**{return FALSE;}**

**Prototype:**

**virtual IKClipObject \*CopyIKParams(int which)**

**Remarks:**

Implemented by the Plug-In.

This method is called to have the controller copy the specified IK parameters to an IKClipObject and return a pointer to it. The plug-in should derive a class from the IKClipObject, put its data in the class, and return a new instance of it. See [Class IKClipObject](#).

**Parameters:**

**int which**

One of the following values:

**COPYPASTE\_IKPOS  
COPYPASTE\_IKROT**

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual BOOL CanPasteIKParams(IKClipObject *co,int which)
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the controller can paste the specified IK parameters; otherwise FALSE.

**Parameters:**

**IKClipObject \*co**

A pointer to the current **IKClipObject** in the clipboard. This class identifies the creator of the clip object. See [Class IKClipObject](#). The plug-in should look at the IDs in the **IKClipObject** to make sure it matches this controller. If it does not, the plug-in should return FALSE.

**int which**

One of the following values:

**COPYPASTE\_IKPOS**

**COPYPASTE\_IKROT**

**Return Value:**

TRUE if the controller can paste the specified IK parameters; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void PasteIKParams(IKClipObject *co,int which)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to have the controller paste the specified IK parameters from the specified IKClipObject to itself.

**Parameters:**

**IKClipObject \*co**

A pointer to an IKClipObject. See [Class IKClipObject](#).

**int which**

One of the following values:

**COPYPASTE\_IKPOS**

**COPYPASTE\_IKROT**

**Default Implementation:**

```
{}
```

**Controllers that wish to have an gizmo (apparatus) available in the scene will implement these methods.**

**Prototype:**

```
virtual int Display(TimeValue t, INode* inode,  
ViewExp *vpt, int flags);
```

**Remarks:**

Implemented by the Plug-In.

This is called by the system to have the controller display its gizmo. When a controller is being edited in the Motion branch, this method is called to allow it to display any apparatus it may have in the scene. Note that **Display()** is only called on Transform Controllers. It is not called on any sub-controllers, for example it wouldn't be called on the position controller of a PRS transform controller.

In R4 and higher however the display method **WILL** be called on Position, Rotation and scale controllers as well

**Parameters:**

**TimeValue t**

The time to display the object.

**INode\* inode**

The node to display.

**ViewExp \*vpt**

An interface pointer that exposes methods the plug-in may call related to the viewports.

**int flags**

See [List of Display Flags](#).

**Return Value:**

Nonzero if the item was displayed; otherwise 0.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual int HitTest(TimeValue t, INode* inode, int type,  
int crossing, int flags, IPoint2 *p, ViewExp *vpt)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to determine if the specified screen point intersects the controller gizmo. The method returns nonzero if the gizmo was hit; otherwise 0.

**Parameters:**

**TimeValue t**

The time to perform the hit test.

**INode\* inode**

A pointer to the node whose gizmo should be tested.

**int type**

The type of hit testing to perform. See [Hit Test Types](#) for details.

**int crossing**

The state of the crossing setting. If TRUE crossing selection is on.

**int flags**

The hit test flags. See [Hit Test Flags](#) for details.

**IPoint2 \*p**

The screen point to test.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**Return Value:**

Nonzero if the controller gizmo was hit; otherwise 0.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual void GetWorldBoundingBox(TimeValue t,INode* inode,  
ViewExp *vpt, Box3& box)
```

**Remarks:**

Implemented by the Plug-In.

This is the world space bounding box of the controllers gizmo.

**Parameters:**

**TimeValue t**

The time to retrieve the bounding box.

**INode \* inode**

The node to calculate the bounding box for.

**ViewExp\* vp**

An interface pointer that exposes portions of View3D that are exported for use by plug-ins.

**Box3& box**

The bounding box is returned through **box**.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void ActivateSubobjSel(int level, XFormModes& modes)
```

**Remarks:**

Implemented by the Plug-In.

When the user changes the selection of the sub-object drop down, this method is called to notify the plug-in. This method should provide instances of a class derived from **CommandMode** to support move, rotate, non-uniform scale, uniform scale, and squash modes. These modes replace their object mode counterparts however the user still uses the move/rotate/scale tool buttons in

the toolbar to activate them. If a certain level of sub-object selection does not support one or more of the modes NULL may be passed. If NULL is specified the corresponding toolbar button will be grayed out.

**Parameters:**

**int level**

The sub-object selection level the command modes should be set to support. A level of 0 indicates object level selection. If level is greater than or equal to 1 the index refers to the types registered by the object in the order they appeared in the list when registered by **Interface::RegisterSubObjectTypes()**. See [Class Interface](#).

**XFormModes& modes**

The command modes to support. See [Class XFormModes](#).

**Default Implementation:**

{ }

**Prototype:**

**virtual void SelectSubComponent(CtrlHitRecord \*hitRec,  
BOOL selected, BOOL all, BOOL invert=FALSE)**

**Remarks:**

Implemented by the Plug-In.

This method is called to change the selection state of the component identified by **hitRec**.

**Parameters:**

**CtrlHitRecord \*hitRec**

Identifies the component whose selected state should be modified. See [Class CtrlHitRecord](#).

**BOOL selected**

TRUE if the item should be selected; FALSE if the item should be de-selected.

**BOOL all**

TRUE if the entire object should be selected; FALSE if only the portion of the identified by **hitRec**.

**BOOL invert=FALSE**

This is set to TRUE when **all** is also set to TRUE and the user is holding down

the Shift key while region selecting in select mode. This indicates the items hit in the region should have their selection state inverted

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void ClearSelection(int selLevel)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to clear the selection for the given sub-object level. All sub-object elements of this type should be deselected.

**Parameters:**

**int selLevel**

Specifies the selection level to clear.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual int SubObjectIndex(CtrlHitRecord *hitRec)
```

**Remarks:**

Implemented by the System.

Returns the index of the sub-object element identified by the CtrlHitRecord **hitRec**. The sub-object index identifies a sub-object component. The relationship between the index and the component is established by the controller. For example a controller may allow the user to select a group of footprints and these groups may be identified as group 0, group 1, group 2, etc. Given a hit record that identifies a footprint, the controller's implementation of this method would return the group index that the footprint belonged to.

**Parameters:**

**CtrlHitRecord \*hitRec**

Identifies the component whose index should be returned. See [Class](#)

[CtrlHitRecord](#).

**Return Value:**

The index of the sub-object element.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void SelectAll(int selLevel);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to select every element of the given sub-object level.

This will be called when the user chooses Select All from the 3ds max Edit menu.

**Parameters:**

**int selLevel**

Specifies the selection level to select.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void InvertSelection(int selLevel);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to invert the specified sub-object level. If the element is selected it should be deselected. If it's deselected it should be selected. This will be called when the user chooses Select Invert from the 3ds max Edit menu.

**Parameters:**

**int selLevel**

Specifies the selection level to invert.

**Default Implementation:**

```
{}
```

**When the user is in a sub-object selection level, the system needs to get the reference coordinate system definition from the current controller being edited so that it can display the axis.**

Two methods allows the system to do this:

**Prototype:**

```
virtual void GetSubObjectCenters(SubObjAxisCallback *cb,  
    TimeValue t,INode *node)
```

**Remarks:**

Implemented by the Plug-In.

When the user is in a sub-object selection level, the system needs to get the reference coordinate system definition from the current controller being edited so that it can display the axes. This method specifies the position of the center. The plug-in enumerates its centers and calls the callback **cb** once for each. See [Sub-Object Coordinate Systems](#).

**Parameters:**

**SubObjAxisCallback \*cb**

The callback object whose methods may be called. See [Class SubObjAxisCallback](#).

**TimeValue t**

The time to enumerate the centers.

**INode \*node**

A pointer to the node.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void GetSubObjectTMs(SubObjAxisCallback *cb,  
    TimeValue t,INode *node)
```

**Remarks:**

Implemented by the Plug-In.

When the user is in a sub-object selection level, the system needs to get the reference coordinate system definition from the current controller being edited

so that it can display the axes. This method returns the axis system of the reference coordinate system. The plug-in enumerates its TMs and calls the callback **cb** once for each. See [Sub-Object Coordinate Systems](#).

**Parameters:**

**SubObjAxisCallback \*cb**

The callback object whose methods may be called. See [Class SubObjAxisCallback](#).

**TimeValue t**

The time to enumerate the TMs.

**Inode \*node**

A pointer to the node.

**Default Implementation:**

**{**

## Modify sub-object apparatuses

### Prototype:

```
virtual void SubMove( TimeValue t, Matrix3& partm,  
    Matrix3& tmAxis, Point3& val,  
    BOOL localOrigin=FALSE )
```

### Remarks:

Implemented by the Plug-In.

When this method is called the plug-in should respond by moving its selected sub-object components.

### Parameters:

#### **TimeValue t**

The time of the transformation.

#### **Matrix3& partm**

The 'parent' transformation matrix. This matrix represents a transformation that would take points in the controller's space and convert them into world space points.

#### **Matrix3& tmAxis**

The matrix that represents the axis system. This is the space in which the transformation is taking place.

#### **Point3& val**

This value is a vector with X, Y, and Z representing the movement along each axis.

#### **BOOL localOrigin=FALSE**

When TRUE the transformation is occurring about the sub-object's local origin.

### Default Implementation:

```
{}
```

### Prototype:

```
virtual void SubRotate( TimeValue t, Matrix3& partm,  
    Matrix3& tmAxis, Quat& val, BOOL localOrigin=FALSE ){}
```

**Remarks:**

Implemented by the Plug-In.

When this method is called the plug-in should respond by rotating its selected sub-object components.

**Parameters:****TimeValue t**

The time of the transformation.

**Matrix3& partm**

The 'parent' transformation matrix. This matrix represents a transformation that would take points in the controller's space and convert them into world space points.

**Matrix3& tmAxis**

The matrix that represents the axis system. This is the space in which the transformation is taking place.

**Quat& val**

The amount to rotate the selected components.

**BOOL localOrigin=FALSE**

When TRUE the transformation is occurring about the sub-object's local origin. Note: This information may be passed onto a transform controller (if there is one) so they may avoid generating 0 valued position keys for rotation and scales. For example if the user is rotating an item about anything other than its local origin then it will have to translate in addition to rotating to achieve the result. If a user creates an object, turns on the animate button, and rotates the object about the world origin, and then plays back the animation, the object does not do what the was done interactively. The object ends up in the same position, but it does so by both moving and rotating. Therefore both a position and a rotation key are created. If the user performs a rotation about the local origin however there is no need to create a position key since the object didn't move (it only rotated). So a transform controller can use this information to avoid generating 0 valued position keys for rotation and scales.

**Prototype:**

```
virtual void SubScale(TimeValue t, Matrix3& partm, Matrix3& tmAxis, Point3& val, BOOL localOrigin=FALSE)
```

**Remarks:**

Implemented by the Plug-In.

When this method is called the plug-in should respond by scaling its selected sub-object components.

**Parameters:****TimeValue t**

The time of the transformation.

**Matrix3& partm**

The 'parent' transformation matrix. This matrix represents a transformation that would take points in the modifier's space and convert them into world space points.

**Matrix3& tmAxis**

The matrix that represents the axis system. This is the space in which the transformation is taking place.

**Point3& val**

This value is a vector with X, Y, and Z representing the scale along X, Y, and Z respectively.

**BOOL localOrigin=FALSE**

When TRUE the transformation is occurring about the sub-object's local origin. See the note above in the Rotate method.

**Prototype:**

**virtual BOOL RescaleTime(Interval oseg, Interval nseg);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is called when the user rescales time in the time configuration dialog. If FALSE is returned from this method then

**Animatable::MapKeys()** will be used to perform the scaling. Controllers can override this method to handle things like rescaling tagents that

**MapKeys()** won't affect and return TRUE if they don't want **MapKeys()** to be called.

**Parameters:**

**Interval oseg**

The old time segment.

### **Interval nseg**

The new time segment.

#### **Default Implementation:**

```
{return FALSE;}
```

#### **Prototype:**

```
virtual int GetDrawPixelStep();
```

#### **Remarks:**

This method is available in release 4.0 and later only.

Prior to R4 TrackView was using static defines to determines the number samples/pixel it used to draw and compute curve extents. Now a controller can override these defaults by implementing GetDrawPixelStep() and GetExtentTimeStep().

This method allows a control to get sampled at a different rate than what trackview does by default so the controller can speed up redraws. It returns the pixel sample rate for when the curve is drawn.

#### **Default Implementation:**

```
{return 5;}
```

#### **Prototype:**

```
virtual int GetExtentTimeStep();
```

#### **Remarks:**

This method is available in release 4.0 and later only.

Prior to R4 TrackView was using static defines to determines the number samples/pixel it used to draw and compute curve extents. Now a controller can override these defaults by implementing GetDrawPixelStep() and GetExtentTimeStep().

This method returns the ticks sample rate used when the curve is checked for its Y extents.

#### **Default Implementation:**

```
{return 40;}
```

## Load / Save

### Prototype:

**IOResult Save(ISave \*isave);**

### Remarks:

Implemented by the System.

The default implementation of **Save()** handles the saving of the out of range types. The plug-in should call this method from its implementation of **Save()**. The plug-in should call this method before it saves any of its chunks.

The out of range types are saved in these chunks:

**CONTROLBASE\_CHUNK**

**INORT\_CHUNK**

**OUTORT\_CHUNK**

### Parameters:

**ISave \*isave**

This pointer may be used to call methods to write data to disk. See [Class ISave](#).

### Return Value:

One of the following values:

**IO\_OK** - The result was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

### Prototype:

**IOResult Load(ILoad \*iload);**

### Remarks:

Implemented by the System.

The default implementation of **Load()** handles the loading of the out of range types. The plug-in should call this method from its implementation of **Load()**. The plug-in should call this method before it loads any of its chunks.

The out of range types are saved in these chunks:

**CONTROLBASE\_CHUNK**

**INORT\_CHUNK**

**OUTORT\_CHUNK**

**Parameters:****ILoad \*iLoad**

This pointer may be used to call methods to load data from disk. See [Class ILoad](#).

**Return Value:**

One of the following values:

**IO\_OK** - The result was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**These are implemented to handle ease curves.**

If a controller is a leaf controller, then it MUST NOT BY DEFINITION have any sub-controllers or references. If it is a leaf controller, then these are implemented to handle the ease curve list. If it is NOT a leaf controller, then these can be overridden.

From ReferenceMaker. See this class for details on these methods.

**int NumRefs();**

**RefTargetHandle GetReference(int i);**

**void SetReference(int i, RefTargetHandle rtarg);**

From Animatable. See this class for details on these methods.

**int NumSubs();**

**Animatable\* SubAnim(int i);**

**TSTR SubAnimName(int i);**

Default implementations of some Animatable methods:

**void\* GetInterface(ULONG id);**

**int PaintFCurves(ParamDimensionBase \*dim,**

**HDC hdc, Rect& rcGraph, Rect& rcPaint,**

**float tzoom, int tscroll, float vzoom, int vscroll, DWORD flags );**

**int GetFCurveExtents(ParamDimensionBase \*dim,**

**float &min, float &max, DWORD flags);**

## Class StdControl

See Also: [Class Control](#), [Class Interval](#), [Class Matrix3](#), [Class Point3](#), [Class Quat](#), [Class ScaleValue](#), [List of Additional Control Related Functions](#)

```
class StdControl : public Control
```

### Description:

StdControl is a class from which you may derived controller objects. Controllers are the objects in 3ds max that control animation. Any controller that does not evaluate itself as a function of its input can subclass off this class.

The purpose of this class is to simplify some aspects of implementing controllers. The only restriction when using this class is that the controller can not evaluate itself as a function of its input. For example, position, rotation, and scale controllers are passed a TM when they are evaluated. They are supposed to calculate their value and apply it to the TM (pre-multiply). It is possible that the controller could calculate its value as some function of this input matrix. For example, a rotation controller could look at the position of the matrix and the position of some other node in the scene and calculate the rotation such that the object is looking at the other node. Most controllers don't do this so they can subclass off this class which handles processing ORTs (Out of Range Types), ease curves and multiplier curves.

This class implements **GetValue()** and **SetValue()** but requires the derived class to implement two new methods: **GetValueLocalTime()** and **SetValueLocalTime()**.

The implementations of **GetValue()** and **SetValue()** handle processing the ORTs and ease and multiplier curves.

Plug-In Information:

Class Defined In CONTROL.H

Super Class ID CTRL\_FLOAT\_CLASS\_ID - Used by float controllers.

CTRL\_POINT3\_CLASS\_ID - Used by Point3 controllers.

CTRL\_MATRIX3\_CLASS\_ID - Used by Matrix3 controllers.

CTRL\_POSITION\_CLASS\_ID - Used by position controllers.

CTRL\_ROTATION\_CLASS\_ID - Used by rotation controllers.

CTRL\_SCALE\_CLASS\_ID - Used by scale controllers.

CTRL\_MORPH\_CLASS\_ID - Used by morph controllers.

Standard File Name Extension DLC

Extra Include File Needed None

### Methods:

#### Prototype:

```
virtual void GetValueLocalTime(TimeValue t, void *val,  
Interval &valid, GetSetMethod method=CTRL_ABSOLUTE)=0;
```

#### Remarks:

Implemented by the Plug-In.

This method is called to have the controller evaluate itself at the given time. In this class the system implements the method **GetValue()**. **GetValue()** calls this method to retrieve the value of the controller at the specified time. The implementation of **GetValue()** then takes care of handling the ORTs, ease curves and multiplier curves. The plug-in must only return the value of the controller.

#### Parameters:

##### **TimeValue t**

The time to retrieve the value.

##### **void \*val**

This points to a variable to hold the computed value of the controller at the specified time. What the plug-in needs to do to store the value of the controller depends on the controller type. There are six controller types: **float, Point3, Position, Rotation, Scale, and Transform.**

##### **float**

**\*val** points to a float

##### **Point3**

**\*val** points to a Point3

##### **Position**

**\*val** points to a Point3

##### **Rotation**

**\*val** points to a Quat

##### **Scale**

**\*val** points to a ScaleValue

## **Transform**

**\*val** points to a Matrix3

## **Interval & valid**

The interval to update. The controllers validity interval should be intersected with this interval. This updates the interval to reflect the interval of the controller.

## **GetMethod method=CTRL\_ABSOLUTE**

This will always be: **CTRL\_ABSOLUTE**

This indicates the controller should simply store its value in **\*val**.

## **Prototype:**

```
virtual void SetValueLocalTime(TimeValue t, void *val,  
    int commit=1, GetSetMethod method=CTRL_ABSOLUTE)=0;
```

## **Remarks:**

Implemented by the Plug-In.

In this class the system implements the method **SetValue()**. **SetValue()** calls this method to store the value of the controller at the specified time. The system takes care of handling the ORTs and multiplier curves. The plug-in must only store the value of the controller.

## **Parameters:**

### **TimeValue t**

The time to store the value.

### **void \*val**

Storage for the value to set. See **\*val** in **Control::SetValue()** for the possible data types passed here.

### **int commit=1**

If this parameter is zero, the controller should save the value at the given time before setting the value. If commit is nonzero, the controller doesn't need to actually update its keys or tangents. See Control methods **CommitValue()** and **RestoreValue()**.

## **GetMethod method=CTRL\_ABSOLUTE**

One of the following values:

**CTRL\_RELATIVE**

Indicates the plug-in should add the value to the existing value **\*val** (i.e. Move/Rotate/Scale)

### **CTRL\_ABSOLUTE**

Indicates the plug-in should just set the value.

#### **Prototype:**

```
void GetValue(TimeValue t, void *val, Interval &valid,  
GetSetMethod method=CTRL_ABSOLUTE);
```

#### **Remarks:**

This method is implemented by the system. Controller that subclass from StdControl only need to implement **GetValueLocalTime()**. See above.

#### **Prototype:**

```
void SetValue(TimeValue t, void *val, int commit=1,  
GetSetMethod method=CTRL_ABSOLUTE);
```

#### **Remarks:**

This method is implemented by the system. Controller that subclass from StdControl only need to implement **SetValueLocalTime()**. See above.

#### **Prototype:**

```
virtual void Extrapolate(Interval range, TimeValue t,  
void *val, Interval &valid, int type)=0;
```

#### **Remarks:**

Implemented by the Plug-In.

This method is used to calculate some of the Out of Range Types (ORTs). There are several kinds of extrapolations that need to be done based on the ORT type. There are template functions implemented in **CONTROL.H** to do them. See [List of Miscellaneous Controller Functions](#) for a description of those available.

#### **Parameters:**

##### **Interval range**

The range that must be extrapolated.

## **TimeValue t**

The time outside the range to extrapolate.

## **void \*val**

Storage for the extrapolated value. See **\*val** in **GetValueLocalTime()** for the possible data types passed here.

## **Interval &valid**

The validity interval of the extrapolated value.

## **int type**

See [List of Out of Range Types](#).

### **Sample Code:**

The following sample code shows the use of the template functions to implement this method to calculate the ORTs.

#### **INTERP\_CONT\_TEMPLATE**

```
void InterpControl<INTERP_CONT_PARAMS>::Extrapolate(Interval  
range,TimeValue t,void *val,
```

```
    Interval &valid,int type)
```

```
{
```

```
    T val0, val1, val2, res;
```

```
    switch (type) {
```

```
        case ORT_LINEAR:
```

```
            if (t<range.Start()) {
```

```
                GetValueLocalTime(range.Start(),&val0,valid);
```

```
                GetValueLocalTime(range.Start()+1,&val1,valid);
```

```
                res = LinearExtrapolate(range.Start(),t,val0,val1,val0);
```

```
            } else {
```

```
                GetValueLocalTime(range.End()-1,&val0,valid);
```

```
                GetValueLocalTime(range.End(),&val1,valid);
```

```
                res = LinearExtrapolate(range.End(),t,val0,val1,val1);
```

```
            }
```

```
            break;
```

```
        case ORT_IDENTITY:
```

```
            if (t<range.Start()) {
```

```
                GetValueLocalTime(range.Start(),&val0,valid);
```

```
                res = IdentityExtrapolate(range.Start(),t,val0);
```

```
            } else {
```

```

    GetValueLocalTime(range.End(),&val0,valid);
    res = IdentityExtrapolate(range.End(),t,val0);
    }
    break;

case ORT_RELATIVE_REPEAT:
    GetValueLocalTime(range.Start(),&val0,valid);
    GetValueLocalTime(range.End(),&val1,valid);
    GetValueLocalTime(CycleTime(range,t),&val2,valid);
    res = RepeatExtrapolate(range,t,val0,val1,val2);
    break;
    }
valid.Set(t,t);
*((T*)val) = res;
}

```

**Prototype:**

**virtual void \*CreateTempValue()=0;**

**Remarks:**

Implemented by the Plug-In.

When processing the ORTs the system might need a temporary variable to hold an intermediate value. Since the system doesn't know the type of the data that the controller is controlling it can't allocate the right amount of temporary storage. It calls this method to do so. The plug-in's implementation of this method should allocate storage to hold its type of data and return a pointer to it.

**Prototype:**

**virtual void DeleteTempValue(void \*val)=0;**

**Remarks:**

Implemented by the Plug-In.

This method simply deletes the memory allocated by **CreateTempValue()**.

**Parameters:**

**void \*val**

Points to the memory allocated by **CreateTempValue()**.

**Prototype:**

**virtual void ApplyValue(void \*val, void \*delta)=0;**

**Remarks:**

Implemented by the Plug-In.

Applies the given value to the given input value. For position, rotation, and scale controllers, the input value will be a matrix and the value being applied will be a Point3, Quaternion, or ScaleValue, respectively. For other controllers the input value is the same type as the value being applied.

**Parameters:**

**void \*val**

The value to update.

**void \*delta**

The value to apply.

**Prototype:**

**virtual void MultiplyValue(void \*val, float m)=0;**

**Remarks:**

Implemented by the Plug-In.

If the controller has multiplier curves then the system will calculate the factor from all the multiplier curves and then ask the controller to multiply the scalar value to the particular data type.

**Parameters:**

**void \*val**

The value to update.

**float m**

The scalar value to multiply **\*val** by.

# Class SceneImport

See Also: [Class ImpInterface](#), [Class Interface](#).

class SceneImport

## Description:

This is a base class for creating file import plug-ins. The plug-in implements methods of this class to describe the properties of the import plug-in and a method that handles the actual import process.

## Methods:

### Prototype:

**SceneImport();**

### Remarks:

Constructor.

### Prototype:

**virtual ~SceneImport();**

### Remarks:

Destructor.

### Prototype:

**virtual int ExtCount() = 0;**

### Remarks:

Implemented by the Plug-In.

Returns the number of file name extensions supported by the plug-in.

### Prototype:

**virtual const TCHAR \*Ext(int i) = 0;**

### Remarks:

Implemented by the Plug-In.

Returns the 'i-th' file name extension (i.e. "3DS").

### Parameters:

**int i**

The index of the file name extension to return.

**Prototype:**

**virtual const TCHAR \*LongDesc() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns a long ASCII description of the file type being imported (i.e. "Autodesk 3D Studio File").

**Prototype:**

**virtual const TCHAR \*ShortDesc() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns a short ASCII description of the file type being imported (i.e. "3D Studio").

**Prototype:**

**virtual const TCHAR \*AuthorName() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the ASCII Author name.

**Prototype:**

**virtual const TCHAR \*CopyrightMessage() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the ASCII Copyright message for the plug-in.

**Prototype:**

**virtual const TCHAR \*OtherMessage1() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the first message string that is displayed.

**Prototype:**

**virtual const TCHAR \*OtherMessage2() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the second message string that is displayed.

**Prototype:**

**virtual unsigned int Version() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the version number of the import plug-in. The format is the version number \* 100 (i.e. v3.01 = 301).

**Prototype:**

**virtual void ShowAbout(HWND hWnd) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to have the plug-in display its "About..." box.

**Parameters:**

**HWND hWnd**

The parent window handle for the dialog.

**Prototype:**

**virtual int DoImport(const TCHAR \*name, ImpInterface \*ii, Interface \*i, BOOL suppressPrompts=FALSE) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method actually performs the file import.

**Parameters:**

**const TCHAR \*name**

The file name chosen by the user to import.

**ImpInterface \*ii**

An import interface pointer that may be used to create objects and nodes in the scene.

**Interface \*i**

Pass the 3ds max interface pointer here.

**BOOL suppressPrompts=FALSE**

This parameter is available in release 2.0 and later only.

When TRUE, the plug-in must not display any dialogs requiring user input. It is up to the plug-in as to how to handle error conditions or situations requiring user input. This is an option set up for the 3ds max API in order for plug-in developers to create batch import plugins which operate unattended. See

**Interface::ImportFromFile().****Return Value:**

One of the following three values should be returned

```
#define IMPEXP_FAIL 0
```

```
#define IMPEXP_SUCCESS 1
```

```
#define IMPEXP_CANCEL 2
```

**Prototype:**

```
virtual int ZoomExtents();
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to control the zoom extents done after the import is accomplished. It returns a value that indicates if the plug-in should override the user preference setting.

Also see the method **Interface::GetImportZoomExtents()** which returns the state of the system zoom extents flag.

**Return Value:**

One of the following values:

```
ZOOMEXT_NOT_IMPLEMENTED
```

Indicates to use the preference setting.

## **ZOOMEXT\_YES**

Indicates to do a zoom extents after import regardless of the preference setting.

## **ZOOMEXT\_NO**

Indicates to **not** do a zoom extents regardless of the preference setting.

### **Default Implementation:**

```
{ return ZOOMEXT_NOT_IMPLEMENTED; }
```

# Class SceneExport

See Also: [Class ExpInterface](#), [Class Interface](#).

class SceneExport

## Description:

This is a base class for creating file export plug-ins. The plug-in implements methods of this class to describe the properties of the export plug-in and a method that handles the actual export process.

## Methods:

### Prototype:

**SceneExport();**

### Remarks:

Constructor.

### Prototype:

**virtual ~SceneExport();**

### Remarks:

Destructor.

### Prototype:

**virtual int ExtCount() = 0;**

### Remarks:

Implemented by the Plug-In.

Returns the number of file name extensions supported by the plug-in.

### Prototype:

**virtual const TCHAR \*Ext(int i) = 0;**

### Remarks:

Implemented by the Plug-In.

Returns the 'i-th' file name extension (i.e. "3DS").

### Parameters:

**int i**

The index of the file name extension to return.

**Prototype:**

**virtual const TCHAR \*LongDesc() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns a long ASCII description of the file type being exported (i.e. "Autodesk 3D Studio File").

**Prototype:**

**virtual const TCHAR \*ShortDesc() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns a short ASCII description of the file type being exported (i.e. "3D Studio").

**Prototype:**

**virtual const TCHAR \*AuthorName() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the ASCII Author name.

**Prototype:**

**virtual const TCHAR \*CopyrightMessage() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the ASCII Copyright message for the plug-in.

**Prototype:**

**virtual const TCHAR \*OtherMessage1() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the first message string that is displayed.

**Prototype:**

**virtual const TCHAR \*OtherMessage2() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the second message string that is displayed.

**Prototype:**

**virtual unsigned int Version() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the version number of the export plug-in. The format is the version number \* 100 (i.e. v3.01 = 301).

**Prototype:**

**virtual void ShowAbout(HWND hWnd) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to have the plug-in display its "About..." box.

**Parameters:**

**HWND hWnd**

The parent window handle for the dialog.

**Prototype:**

**virtual int DoExport(const TCHAR \*name,ExpInterface \*ei,Interface \*i, BOOL suppressPrompts=FALSE, DWORD options=0) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called for the plug-in to perform its file export.

**Parameters:****const TCHAR \*name**

The export file name.

**ExpInterface \*ei**

A pointer the plug-in may use to call methods to enumerate the scene.

**Interface \*i**

An interface pointer the plug-in may use to call methods of 3ds max.

**BOOL suppressPrompts=FALSE**

This parameter is available in release 2.0 and later only.

When TRUE, the plug-in must not display any dialogs requiring user input. It is up to the plug-in as to how to handle error conditions or situations requiring user input. This is an option set up for the 3ds max API in order for plug-in developers to create batch export plugins which operate unattended. See

**Interface::ExportToFile().****DWORD options=0**

This parameter is available in release 3.0 and later only.

In order to support export of selected objects (as well as future enhancements), this method now has this additional parameter. The only currently defined option is:

**SCENE\_EXPORT\_SELECTED**

When this bit is set the export module should only export the selected nodes.

**Return Value:**

One of the following three values should be returned

**#define IMPEXP\_FAIL 0****#define IMPEXP\_SUCCESS 1****#define IMPEXP\_CANCEL 2****Prototype:****virtual BOOL SupportsOptions(int ext, DWORD options);****Remarks:**

This method is available in release 3.0 and later only.

This method is called by 3ds max to determine if one or more export options are supported by a plug-in for a given extension. It should return TRUE if all

option bits set are supported for this extension; otherwise FALSE.

Note that the method has a default implementation defined in order to provide easy backward compatibility. It returns FALSE, indicating that no options are supported.

**Parameters:**

**int ext**

This parameter indicates which extension the options are being queried for, based on the number of extensions returned by the

**SceneExport::ExtCount()** method. This index is zero based.

**DWORD options**

This parameter specifies which options are being queried, and may have more than one option specified. At present, the only export option is

**SCENE\_EXPORT\_SELECTED**, but this may change in the future. If

more than one option is specified in this field, the plugin should only return

TRUE if all of the options are supported. If one or more of the options are not supported, the plugin should return FALSE.

**Default Implementation:**

```
{return FALSE;}
```

## Class Atmospheric

See Also: [Class SpecialFX](#), [Class SFXParamDlg](#), [Class IRendParams](#), [Class ShadeContext](#), [Class Point3](#), [Class Color](#), [Class Interval](#).

```
class Atmospheric : public SpecialFX
```

### Description:

This is the base class for the creation of Atmospheric plug-ins. Developers may look to the text below for information on the techniques used to create these types of effects:

Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, Texturing and Modeling A Procedural Approach (Cambridge, MA: Academic Press, Inc., 1994). ISBN: 0-12-228760-6.

Chapter 6 on Hypertexture, particularly the section on 'Raymarching', describes essentially the method 3ds max uses.

### Methods:

#### Prototype:

```
virtual AtmosParamDlg *CreateParamDialog(IRendParams *ip)
```

#### Remarks:

Implemented by the Plug-In.

This method creates and returns a new instance of a class derived from **AtmosParamDlg** to manage the user interface. This put up a modal dialog that lets the user edit the plug-ins parameters.

#### Parameters:

```
IRendParams *ip
```

This is the interface given to the atmospheric effect so it may display its parameters.

#### Return Value:

A new instance of a class derived from **AtmosParamDlg**.

Note: **typedef SFXParamDlg AtmosParamDlg;**

#### Default Implementation:

```
{return NULL;}
```

**Prototype:**

```
virtual BOOL SetDlgThing(AtmosParamDlg* dlg);
```

**Remarks:**

This method is available in release 3.0 and later only.

You should implement this if you are using the ParamMap2 AUTO\_UI system and the effect has secondary dialogs that have something other than the incoming effect as their 'thing'. Called once for each secondary dialog for you to install the correct thing. Return TRUE if you process the dialog, FALSE otherwise, in which case the incoming effect will be set into the dialog.

Note: Developers needing more information on this method can see the remarks for [MtlBase::CreateParamDlg\(\)](#) which describes a similar example of this method in use (in that case it's for use by a texture map plug-in).

**Parameters:**

```
AtmosParamDlg* dlg
```

Points to the ParamDlg.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void Shade(ShadeContext& sc,const Point3& p0,const  
Point3& p1,Color& color,Color& trans, BOOL isBG=FALSE)=0;
```

**Remarks:**

Implemented by the Plug-In.

This is the function that is called to apply the atmospheric effect. The line segment defined by the world space points **p0** and **p1** represent a segment of volume that needs to be shaded. This line segment is a line along the ray defined by the line between the viewer's eye (the camera) and the pixel being rendered in the image plane and continuing through world space. This ray is broken up into segments, with the boundaries defined by surfaces. If there are no surfaces, there will just be a single segment from the eye point going off into space (**p1** will be a large number). If there is a surface that is hit, and the surface is opaque, there will still be one segment from the eye to the surface. If the surface is transparent there may be two segments, the segment before and the segment after. Therefore the ray may be broken up into many segments

depending on the number of transparent surfaces hit.

The shader does not need to be directly concerned with this however. It only knows it's shading the segment between **p0** and **p1**. It will get called repeatedly for different pixels and different segments.

The output of this method is the computed **color** and **transparency**.

An example implementation of this method is 3ds max's Simple Fog. It takes the distance of the line segment and essentially interpolates towards the fog color based on the distance. This is a very simple effect.

3ds max's Volume Fog traverses along the segment and evaluates its 3D noise function. It integrates the density across the segment, and uses the density to compute the fog color.

### **Parameters:**

#### **ShadeContext& sc**

The ShadeContext.

#### **const Point3& p0**

The start point of the segment to shade. This point (and **p1**) are in an undefined 'internal' space specific to the renderer (which for the 3ds max renderer is in fact is camera space). To get to world space the plug-in would call **sc.PointTo(p0,REF\_WORLD)**.

#### **const Point3& p1**

The end point of the segment to shade.

#### **Color& color**

This method shades the volume between p0 and p1 and modifies this color.

#### **Color& trans**

This method shades the volume between p0 and p1 and modifies this transparency.

#### **BOOL isBG=FALSE**

TRUE if the background is being shaded; otherwise FALSE. If TRUE then p1 will be infinity (a large number). This is used when the option to not fog the background is on.

### **Prototype:**

```
IOResult Save(ISave *isave);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the system.

To facilitate naming atmospheric effects, a 'name' string has been added to the base class. This method should be called from the developers sub-classed Atmospheric plug-in to save the name.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the system.

To facilitate naming atmospheric effects, a 'name' string has been added to the base class. This method should be called from the developers sub-classed Atmospheric plug-in to load the name.

# Class Mtl

See Also: [Class MtlBase](#), [Class Texmap](#), [Class ShadeContext](#), [ShadeOutput](#), [Class Interval](#), [Class Color](#).

```
class Mtl : public MtlBase
```

## Description:

This is the base class for the creation of material plug-ins. This class provides methods to do things such as compute the properties of the material for a given location, return the number of sub-materials and access the properties of the material for the interactive renderer.

## Plug-In Information:

Class Defined In IMTL.H

Super Class ID MATERIAL\_CLASS\_ID

Standard File Name Extension DLT

Extra Include File Needed IMTL.H

## Method Groups:

The following hyperlinks take you to the start of groups of related methods within the class:

[Naming Methods](#)

[Shade](#)

[Get/SetActiveTexmap](#)

[Get Property Methods](#)

[Set Property Methods](#)

[Sub-Mtl Access Methods](#)

[Dynamic Properties Methods](#)

[Displacement Mapping](#)

[Replace Material Dialog](#)

## Methods:

## Prototype:

```
Mtl();
```

## Remarks:

Constructor. The active texture map is set to NULL.

## Naming Methods

### Prototype:

**virtual TSTR GetSubMtlSlotName(int i);**

### Remarks:

Implemented by the Plug-In.

This method returns the slot name of the 'i-th' sub-material. This name appears in the materials editor dialog. For instance, if you are in a material and then you go down into a sub-material, this is the name that appears just below the 'Get Material' icon. For example, in the Multi/Sub-Object material when you choose one of the sub-materials, the map name appears to let you know which slot you are working on. For the Multi/Sub-Object material, this is the number of the slot, i.e. "#1:", "#2:", "#3:", etc.

### Parameters:

**int i**

The index of the sub-materials whose slot name should be returned.

### Prototype:

**TSTR GetSubMtlTVName(int i);**

### Remarks:

Implemented by the Plug-In.

Returns the name of the 'i-th' sub-material that should appear in track view.

### Parameters:

**int i**

The index of the sub-materials whose track view name should be returned.

## Shade

### Prototype:

**virtual void Shade(ShadeContext& sc)=0;**

### Remarks:

Implemented by the Plug-In.

This is the main method of the material. This is called by the renderer to compute the color and transparency output returned in **sc.out**.

**Parameters:**

**ShadeContext& sc**

Describes properties of the pixel to be shaded. The result of this method is returned in the **ShadeOutput** data member of **sc**.

## Get/SetActiveTexmap

**Prototype:**

**MtlBase\* GetActiveTexmap();**

**Remarks:**

Implemented by the System.

Returns a pointer to the active texture map used in the interactive renderer.

**Prototype:**

**void SetActiveTexmap(MtlBase \*txm);**

**Remarks:**

Implemented by the System.

Stores the pointer to the active texture map used in the interactive renderer.

Note that this method does not do everything required to update the viewports with the new texmap. To accomplish that call

**Interface::ActivateTexture()**.

## Get Properties Methods

The following methods are used for displaying materials in the 3ds max viewports.

**Prototype:**

**virtual Color GetAmbient(int mtlNum=0, BOOL  
backFace=FALSE)=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the ambient color of the specified material for use in the interactive renderer.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Prototype:**

```
virtual Color GetDiffuse(int mtlNum=0, BOOL  
backFace=FALSE)=0;
```

**Remarks:**

Implemented by the Plug-In.

Returns the diffuse color of the specified material for use in the interactive renderer.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Prototype:**

```
virtual Color GetSpecular(int mtlNum=0, BOOL  
backFace=FALSE)=0;
```

**Remarks:**

Implemented by the Plug-In.

Returns the specular color of the specified material for use in the interactive renderer.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Prototype:**

**virtual float GetShininess(int mtlNum=0, BOOL  
backFace=FALSE)=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the shininess value of the specified material for use in the interactive renderer.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Prototype:**

**virtual float GetShinStr(int mtlNum=0, BOOL  
backFace=FALSE)=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the shininess strength value of the specified material for use in the interactive renderer.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Prototype:**

```
virtual float GetXParency(int mtlNum=0, BOOL  
backFace=FALSE)=0;
```

**Remarks:**

Implemented by the Plug-In.

Returns the transparency value of the specified material for use in the interactive renderer.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Prototype:**

```
virtual float GetSelfIllum(int mtlNum=0, BOOL  
backFace=FALSE);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Returns the self illumination value of the specified material for use in the interactive renderer.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Default Implementation:**

```
{ return 0.0f; }
```

**Prototype:**

**virtual BOOL GetSelfIllumColorOn(int mtlNum=0, BOOL backFace=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the Self Illumination Color is on (checked) for the specified material; otherwise FALSE.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

**virtual Color GetSelfIllumColor(int mtlNum=0, BOOL backFace=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the Self Illumination Color of the specified material for use in the interactive renderer.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Default Implementation:**

```
{ Color c( .0f,.0f,.0f); return c; }
```

**Prototype:**

**virtual float WireSize(int mtlNum=0, BOOL backFace=FALSE)**

**Remarks:**

Implemented by the Plug-In.  
Returns the wire size of the material.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Default Implementation:**

```
{ return 1.0f; }
```

**Prototype:**

```
virtual Sampler *GetPixelSampler(int mtlNum=0, BOOL  
backFace=FALSE);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Returns a pointer to the Sampler used for the specified sub-material.

**Parameters:**

**int mtlNum=0**

This is the material index for mult-materials.

**BOOL backFace=FALSE**

If the surface normal of the face is pointing away from the viewer this will be TRUE; otherwise FALSE.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual RenderData *GetRenderData();
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the auxiliary data attached to the material by the renderer.

**Default Implementation:**

```
{ return renderData; }
```

**Prototype:**

```
virtual BOOL IsOutputConst(ShadeContext& sc, int stdID);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns TRUE if the evaluated color/value (output) is constant over all possible inputs described by the shade context. If it cannot determine the correct answer, it returns FALSE.

**Parameters:**

**ShadeContext& sc**

This describes the context of the question.

**int stdID**

The ID of the channel in question. See [List of Texture Map Indices](#)

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual BOOL EvalColorStdChannel(ShadeContext& sc, int stdID,  
Color&  
outClr);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method evaluates the material on a single standard texmap channel over an area described in the ShadeContext. A return value of FALSE indicates that the color could not be evaluated.

If there's no texmap defined for a channel or the output of the texmap is not "meaningful", the raw color stored by the material or shader is returned. (The output of a texmap is meaningful in a given ShadeContext if it is the same as when the scene is rendered. If the map cannot determine whether the output value is the same as when rendered, it should not be meaningful.)

Note that the output color is not clamped. If the method is called with a monochrome channel ID, the result value is copied in the R, G and B components of the Color structure.

As a default implementation, this method sets the output color to black and returns FALSE.

**Parameters:**

**ShadeContext& sc**

This describes the context in which the material should be evaluated.

**int stdID**

The ID of the channel to perform evaluation on. See [List of Texture Map Indices](#)

**Color& outClr**

The result of the evaluation.

**Default Implementation:**

```
{ return FALSE; }
```

## Set Properties Methods

**Prototype:**

```
virtual void SetAmbient(Color c, TimeValue t)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method saves the specified color at the specified time.

**Parameters:**

**Color c**

The color to store.

**TimeValue t**

The time to set this color.

**Prototype:**

```
virtual void SetDiffuse(Color c, TimeValue t)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method saves the specified color at the specified time.

**Parameters:**

**Color c**

The color to store.

**TimeValue t**

The time to set this color.

**Prototype:**

**virtual void SetSpecular(Color c, TimeValue t)=0;**

**Remarks:**

Implemented by the Plug-In.

This method saves the specified color at the specified time.

**Parameters:**

**Color c**

The color to store.

**TimeValue t**

The time to set this color.

**Prototype:**

**virtual void SetShininess(float v, TimeValue t)=0;**

**Remarks:**

Implemented by the Plug-In.

This method saves the specified shininess at the specified time.

**Parameters:**

**float v**

The shininess value to store.

**TimeValue t**

The time to set this color.

**Prototype:**

**virtual void SetRenderData(RenderData \*rdata);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows the renderer to attach auxiliary data to each material.

**Parameters:**

**RenderData \*rdata**

The auxiliary data you wish to attach.

**Default Implementation:**

```
{ renderData = rdata; }
```

**Sub-material Access Methods****Prototype:**

```
virtual int NumSubMtls()
```

**Remarks:**

Implemented by the Plug-In.

Returns the number of sub-materials managed by this material.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual Mtl* GetSubMtl(int i)
```

**Remarks:**

Implemented by the Plug-In.

Returns a pointer to the 'i-th' sub-material of this material.

**Parameters:**

**int i**

The index of the material to return.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void SetSubMtl(int i, Mtl *m)
```

**Remarks:**

Implemented by the Plug-In.

Stores the 'i-th' sub-material of this material.

**Parameters:****int i**

The index of the material to store.

**Mtl \*m**

The material pointer to store.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int SubMtlOn(int i);
```

**Remarks:**

This method is not currently used.

**Prototype:**

```
virtual int VPDisplaySubMtl();
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns which sub-mtl is to display in the viewport, a return value of:-1 indicates not implemented.

Note that when a material, such as Blend, has a method of selecting which sub-map is to be shown in the viewport, and implements this method, the materials editor lets you turn on Show Map In Viewport (SMIV) in all the sub maps at once. When the material, such as top-bottom, doesn't have a selector, and doesn't implement this method, then the materials editor will only let you turn on SMIV for one map/mtl in the entire sub-tree of the material.

**Default Implementation:**

```
{ return -1; }
```

**Dynamic Properties Methods**

**Prototype:**

```
float GetDynamicsProperty(TimeValue t, int mtlNum, int propID);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the specified dynamics property of the material at the specified time.

**Parameters:****TimeValue t**

The time to return the dynamics property.

**int mtlNum**

The index of the sub-material or zero if this is a base material.

**int propID**

Specifies the type of property. One of the following values:

**DYN\_BOUNCE**

The bounce coefficient. Values in the range 0.0 to 1.0.

**DYN\_STATIC\_FRICTION**

The static friction property. Values in the range 0.0 to 1.0.

**DYN\_SLIDING\_FRICTION**

The sliding friction property. Values in the range 0.0 to 1.0.

**Default Implementation:**

The default implementation for Mtl will handle all multi-materials. All root level materials (for instance Standard) need to implement this method.

**Prototype:**

```
void SetDynamicsProperty(TimeValue t, int mtlNum, int propID, float value);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the specified dynamics property of the material at the specified time.

**Parameters:**

## **TimeValue t**

The time at which to set the value.

## **int mtlNum**

The sub-material number for a multi-material.

## **int propID**

Specifies the type of property. One of the following values:

### **DYN\_BOUNCE**

The bounce coefficient. Values in the range 0.0 to 1.0.

### **DYN\_STATIC\_FRICTION**

The static friction property. Values in the range 0.0 to 1.0.

### **DYN\_SLIDING\_FRICTION**

The sliding friction property. Values in the range 0.0 to 1.0.

## **float value**

The value to set.

### **Default Implementation:**

The default implementation for Mtl will handle all multi-materials. All root level materials (for instance Standard) need to implement this method.

## **Displacement Mapping**

### **Prototype:**

```
virtual float EvalDisplacement(ShadeContext& sc);
```

### **Remarks:**

This method is available in release 2.0 and later only.

Returns the amount of displacement along the normal of the surface at the point as specified by the **ShadeContext**.

### **Parameters:**

**ShadeContext& sc**

This contains the details of the point being displaced.

### **Default Implementation:**

```
{ return 0.0f; }
```

**Prototype:****virtual Interval DisplacementValidity(TimeValue t);****Remarks:**

This method is available in release 2.0 and later only.

Returns the validity interval of the displacement mapping around the specified time.

**Parameters:****TimeValue t**

The Interval returned reflects the validity around this time.

**Default Implementation:****{ return FOREVER; }****Replace Material Dialog****Prototype:****virtual BOOL DontKeepOldMtl();****Remarks:**

This method is available in release 3.0 and later only.

Return TRUE to prevent the Replace Material (Discard old material? / Keep old material as sub-material?) dialog from being presented to the user; FALSE to allow it to be presented. This allows a plug-in to control the display of this dialog when being created in a Material Editor slot.

**Default Implementation:****{ return FALSE; }****Prototype:****virtual BOOL SupportsShaders();****Remarks:**

This method is available in release 4.0 and later only.

This method returns TRUE if the material supports shaders, otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

## Class Texmap

See Also: [Class MtlBase](#), [Class ShadeContext](#), [Class UVGen](#), [Class XYZGen](#), [Class TexHandleMaker](#), [Class NameAccum](#), [Class AColor](#), [Class Matrix3](#), [List of Procedural Texture Clamping, Noise and Misc Functions](#).

```
class Texmap: public MtlBase
```

### Description:

This is the base class for the creation of texture map plug-ins. It provides methods for things such as calculating the color of the map for a given location, and computing a perturbation to apply to a normal for bump mapping.

Note: Developers creating procedural textures should be aware that these textures may appear less than perfect when a 3ds max user is using the "Quick Renderer" in the Materials Editor. This is not the fault of the plug-in texture, it is simply that the "Quick Renderer" uses an algorithm that is quicker but less accurate than the standard scanline renderer. Therefore, don't be concerned if your texture does not show up perfectly when using "Quick Renderer".

### Plug-In Information:

Class Defined In IMTL.H

Super Class ID TEXMAP\_CLASS\_ID

Standard File Name Extension DLT

Extra Include Files Needed IMTL.H (and optionally TEXUTIL.H)

### Method Groups:

The hyperlinks below jump to the start of groups of related methods within the class:

[Eval Methods](#)

[Output Level](#)

[Texture Display in the Viewports](#)

[Loading Map Files](#)

[Slot Type](#)

[Get UV Transform / Texture Tiling](#)

[UVGen and XYZGen Access](#)

[Additional Details related to Bump Mapping](#)

**Methods:****Prototype:**

**Texmap();**

**Remarks:**

Constructor. The number of active uses of this texture is set to 0.

**Eval Methods****Prototype:**

**virtual AColor EvalColor(ShadeContext& sc)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to evaluate the color of the texture map for the context.

This is for channels that have a color such as diffuse, specular, ambient, etc.

This method is called for every pixel of the texture.

**Parameters:**

**ShadeContext& sc**

Describes the properties of the pixel to evaluate.

**Return Value:**

An **AColor** object which stores the **r, g, b, a** values. Note: The alpha is premultiplied, and the alpha value goes into **AColor::a**.

**Prototype:**

**virtual float EvalMono(ShadeContext& sc);**

**Remarks:**

Implemented by the Plug-In.

Evaluate the map for a "mono" channel. Mono channels are those that don't have a color, but rather a single value. This is for things like shininess, transparency, etc. This just permits a bit of optimization.

**Parameters:**

**ShadeContext& sc**

Describes the properties of the pixel to evaluate.

**Return Value:**

A floating point value for the mono channel.

**Default Implementation:**

```
{return Intens(EvalColor(sc));}
```

**Prototype:**

```
virtual BOOL EvalColorMonoChannel(ShadeContext& sc, int  
stdID, Color&  
outClr);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method evaluates the material on a single standard texmap channel over an area described in the ShadeContext. A return value of FALSE indicates that the value could not be evaluated.

If there's no texmap defined for a channel or the output of the texmap is not "meaningful", the raw value stored by the material or shader is returned. For the definition of the term "meaningful" see

**Texmap::IsOutputMeaningful()**.

Note that the output value is not clamped. If the method is called on a color channel, the intensity of the RGB value is returned. The intensity is computed as defined by the global helper method Interns in the 3ds max SDK.

As a default implementation, this calls **EvalColorStdChannel()** method and sets the result to the intensity of the color.

**Parameters:**

**ShadeContext& sc**

This describes the context in which the material should be evaluated.

**int stdID**

The ID of the channel to perform evaluation on. See List of Texture Map Indices *\*\*aztodo\*\** [link this](#)

**Color& outClr**

The result of the evaluation.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual Point3 EvalNormalPerturb(ShadeContext& sc)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is used for bump mapping to retrieve a perturbation to apply to a normal.

**Parameters:**

```
ShadeContext& sc
```

Describes the properties of the pixel to evaluate.

**Return Value:**

A deflection vector for perturbing the normal.

**Prototype:**

```
virtual BOOL HandleOwnViewPerturb();
```

**Remarks:**

This method is available in release 2.0 and later only.

This query is made of maps plugged into the Reflection or Refraction slots: Normally the view vector is replaced with a reflected or refracted one before calling the map: if the plugged in map doesn't need this, it should return TRUE.

**Default Implementation:**

```
{ return FALSE; }
```

## Output Level

**Prototype:**

```
virtual void SetOutputLevel(TimeValue t, float v)
```

**Remarks:**

Implemented by the Plug-In.

Sets the output level at the specified time. It is used to set the output level of the imbedded Texout object, principally by importing plug-ins. It is implemented in all Texmaps.

**Parameters:**

**TimeValue t**

The time to set the output level.

**float v**

The value to set.

**Default Implementation:**

```
{
```

**Prototype:**

```
void RecursInitSlotType(int sType);
```

**Remarks:**

Implemented by the System.

This method is used internally to set the slot type for all subtexmaps in a tree.

**Prototype:**

```
virtual BITMAPINFO* GetVPDisplayDIB(TimeValue t,  
TexHandleMaker& thmaker, Interval &valid, BOOL  
mono=FALSE, int forceW=0, int forceH=0);
```

**Remarks:**

This method is available in release 4.0 and later only.

This gets the viewport display bitmap in DIB format, useful when combining several maps for hardware-supported multiple texture display. If **mono** is TRUE, the map should do a mono evaluation and place the result in RGB.

**forceW** and **forceH**, if non-zero, override dimensions specified by **thmaker**.

**Parameters:****TimeValue t**

The time to get the bitmap at.

**TexHandleMaker& thmaker**

This class provides methods for creating a texture handle from a 3ds max bitmap and a Windows DIB. It also has a method to retrieve the required size of the texture map. See [Class TexHandleMaker](#).

**Interval &valid**

The validity interval of the returned bitmap.

**BOOL mono**

Indicates whether a map should do mono evaluation.

**int forceW**

Overrides the bitmap width usually supplied by **thmaker**.

**int forceH**

Overrides the bitmap height usually supplied by **thmaker**.

**Get UV Transform / Texture Tiling****Prototype:**

**virtual void GetUVTransform(Matrix3 &uvtrans)**

**Remarks:**

Implemented by the Plug-In.

This method is called to retrieve the UV transformation matrix for use in the viewports. If a developer is using an instance of UVGen, a method of that class may be called to retrieve the value:

(i.e. **{ uvGen->GetUVTransform(uvtrans); }** ).

**Parameters:****Matrix3 &uvtrans**

The transformation matrix is returned here.

**Default Implementation:**

**{ }**

**Prototype:**

**virtual int GetTextureTiling()**

**Remarks:**

Implemented by the Plug-In.

This method is called to get the tiling state of the texture for use in the viewports. This is described by a set of symmetry flags that may be ORed together. If you are using an instance of UVGen to handle the UV user interface you may simply call a method of UVGen to handle this.

For example: **{ return uvGen->GetTextureTiling(); }**

**Return Value:**

See [List of Texture Symmetry Flags](#).

**Default Implementation:**

```
{ return U_WRAP|V_WRAP; }
```

**Prototype:**

```
virtual int GetUVWSource();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns a value indicating where to get the texture vertices for the Texmap.

**Return Value:**

One of the following values:

**UVWSRC\_EXPLICIT**

Use explicit mesh texture vertices from one of the mapping channels (see **GetMapChannel()** below to determine which one). This uses the UVW coordinates assigned to the object, either through the Generate Mapping Coordinates option in the object's creation parameters, or through mapping modifiers, such as UVW Map.

**UVWSRC\_EXPLICIT2**

Use explicit mesh texture vertices from the Vertex Color Channel.

**UVWSRC\_OBJXYZ**

Generate planar UVW mapping coordinates from the object local XYZ on-the-fly. This corresponds to the "Planar from Object XYZ" option.

**UVWSRC\_WORLDXYZ**

This value is available in release 3.0 and later only.

Generate planar UVW mapping coordinates from the world XYZ on-the-fly. This corresponds to the "Planar From World XYZ" option. Note: this value used for the UVW is the world XYZ, taken directly, with out normalization to the objects bounding box. This differs from "Planar from Object XYZ", where the values are normalized to the object's bounding box.

**Default Implementation:**

```
{ return UVWSRC_EXPLICIT; }
```

**Prototype:**

```
virtual int GetMapChannel();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the map channel being used by the texmap if **GetUVWSource()** returns **UVWSRC\_EXPLICIT**. The return value should be at least 1. A value of 0 is not acceptable.

**Default Implementation:**

```
{ return 1; }
```

## UVGen and XYZGen Access

**Prototype:**

```
virtual UVGen *GetTheUVGen();
```

**Remarks:**

This method is available in release 2.0 and later only.

Texture maps that use a UVGen should implement this method to return a pointer to it.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual XYZGen *GetTheXYZGen();
```

**Remarks:**

This method is available in release 2.0 and later only.

Texture maps that use a XYZGen should implement this method to return a pointer to it.

**Default Implementation:**

```
{ return NULL; }
```

## Loading Bitmap Files

**Prototype:**

## **virtual int LoadMapFiles(TimeValue t)**

### **Remarks:**

Implemented by the Plug-In.

This method is called prior to rendering to allow the plug-in to load any bitmap files it requires.

Note that **LoadMapFiles()** is called to load map files only, not to list the missing files. The missing files are listed using the **EnumAuxFiles()** method, which allows enumerating them without loading them.

Also Note: There is currently not an **UnloadMapFiles()** method. There are a couple of ways to do this however. One is to call

**Interface::FreeAllBitmaps()**. That method traverses the scene reference hierarchy and calls **Animatable::FreeAllBitmaps()** on each item. Another approach is to evaluate the Material / TextureMap hierarchy on each material.

Then call **Animatable::FreeAllBitmaps()** yourself in the

**MtlEnum::proc()** shown below.

```
class MtlEnum {  
public:  
    virtual void proc(MtlBase *m) = 0;  
};  
  
void EnumMtlTree(MtlBase *mb, MtlEnum &tenum) {  
    tenum.proc(mb);  
    for (int i=0; i<mb->NumSubTexmaps(); i++) {  
        Texmap *st = mb->GetSubTexmap(i);  
        if (st)  
            EnumMtlTree(st,tenum);  
        }  
    if (IsMtl(mb)) {  
        Mtl *m = (Mtl *)mb;  
        for (i=0; i<m->NumSubMtls(); i++) {  
            Mtl *sm = m->GetSubMtl(i);  
            if (sm)  
                EnumMtlTree(sm,tenum);  
            }  
    }
```

```
}  
}
```

Now just define a subclass of MtlEnum that does what you want, and call EnumMtlTree. In this particular case it is more efficient than enumerating the entire reference hierarchy. If you do want to enumerate the entire reference hierarchy, here's how

```
class RefEnumProc {  
    public:  
    virtual void proc(ReferenceMaker *rm)=0;  
};  
  
void EnumRefs(ReferenceMaker *rm, RefEnumProc &proc)  
{  
    proc.proc(rm);  
    for (int i=0; i<rm->NumRefs(); i++) {  
        ReferenceMaker *srm = rm->GetReference(i);  
        if (srm) EnumRefs(srm,proc);  
    }  
}
```

Just define a subclass of RefEnumProc that does what you want, and call EnumRefs on the part of the reference hierarchy you want to enumerate. For example

```
class MyEnum: public RefEnumProc {  
    void proc(ReferenceMaker *rm) { /* do something */ }  
}  
  
void afunction(Mtl* m) {  
    MyEnum enumer;  
    EnumRefs(m,&enumer);  
}
```

**Parameters:**

**TimeValue t**

The time the maps are being loaded.

**Return Value:**

Always return nonzero from this method.

**Default Implementation:**

```
{ return 1; }
```

**Prototype:**

```
virtual void RenderBitmap(TimeValue t, Bitmap *bm, float  
scale3D=1.0f, BOOL filter = FALSE);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to render a 2D bitmap version of this texmap.

**Parameters:**

**TimeValue t**

The time at which to render the texmap to the bitmap.

**Bitmap \*bm**

A pointer to a bitmap to render to. This bitmap must be created at the resolution you wish to render to.

**float scale3D=1.0f**

This is a scale factor applied to 3D Texmaps. This is the scale of the surface in 3d space that is mapped to UV. This controls how much of the texture appears in the bitmap representation.

**BOOL filter = FALSE**

If TRUE the bitmap is filtered. It is quite a bit slower to rescale bitmaps with filtering on.

## Slot Type

**Prototype:**

```
virtual void InitSlotType(int sType)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to initialize the slot type. This sets the proper button in the coordinate user interface rollup page. If you are using an instance of UVGen to handle the UV user interface you may simply call a method of

UVGen to handle this. For example: **{ if (uvGen) uvGen->InitSlotType(sType); }**

**Parameters:**

**int sType**

See [List of Map Slot Types](#).

**Default Implementation:**

**{ }**

**Prototype:**

**virtual void RenderBitmap(TimeValue t, Bitmap \*bm, float scale3D=1.0f, BOOL filter = FALSE);**

**Remarks:**

This method is available in release 2.0 and later only.

Renders the texmap to the specified bitmap.

**Parameters:**

**TimeValue t**

The time at which to render the bitmap.

**Bitmap \*bm**

The result is stored in this bitmap. The properties of this bitmap define the resolution, color depth, etc.

**float scale3D=1.0f**

This is the scale of the surface in 3d space that is mapped to UV.

**BOOL filter = FALSE**

If TRUE the bitmap is filtered. It is quite a bit slower to rescale bitmaps with filtering on.

**Default Implementation:**

The default implementation calls **Interface::RenderTexmap()**.

**Prototype:**

**virtual BOOL IsOutputMeaningful(ShadeContext& sc);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the system.

Returns TRUE only if all submaps and itself have a meaningful output.

Returns FALSE if at least one sub-texmap or itself does not have a meaningful output

The output of a texmap is meaningful in a given ShadeContext if it is the same as when the scene is rendered. If the map cannot determine whether the output value is the same as when rendered, it should not be meaningful. This method can be called before **EvalColor()** or **EvalMono()** on a texmap in order to decide whether to call these methods at all or if their return values should be used in further calculations.

**Parameters:**

**ShadeContext& sc**

This describes the context of the question.

**Prototype:**

**virtual BOOL IsLocalOutputMeaningful(ShadeContext& sc);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns TRUE if the output of **this** texmap is meaningful for the given context; it should not take into account subtexmaps. This method is called by **IsOutputMeaningful()**.

**Parameters:**

**ShadeContext& sc**

This describes the context of the question.

**Default Implementation:**

**{ return FALSE; }**

**Prototype:**

**virtual int IsHighDynamicRange( ) const;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns nonzero if the texture is returning high dynamic range data; otherwise zero.

### Default Implementation:

```
{ return false; }
```

### Additional Details related to Bump Mapping

Note the following information concerning bump mapping:

The following function evaluates the normal perturbation vector in the Bitmap texture.

```
Point3 BMTex::EvalNormalPerturb(ShadeContext& sc) {
    Point3 dPdu, dPdv;
    Point2 dM;
    if (gbufID) sc.SetGBufferID(gbufID);
    if (thebm==NULL)
        return Point3(0,0,0);
    uvGen->GetBumpDP(sc,dPdu,dPdv); // get bump basis vectors
    if (alphaAsMono)
        dM=(.01f)*uvGen-
>EvalDeriv(sc,&alphasamp,filterType!=FILTER_NADA);
    else
        dM=(.01f)*uvGen-
>EvalDeriv(sc,&mysamp,filterType!=FILTER_NADA);
    return texout->Filter(dM.x*dPdu+dM.y*dPdv);
}
```

The function **GetBumpDP()** returns the "bump basis vectors". These are the gradient vectors of the UVW mapping in the object coordinate space.

The following function is used to compute the U and V bump basis vectors for a triangle given the texture coordinates at the three vertices of the triangle ( tv[] ) and the 3D coordinates at the vertices ( v[] ). It is simply a solution using linear algebra for the U and V axes in terms of the XYZ coordinates. It returns

**b[0] = DP/DU**

**b[1] = DP/DV**

This function does not compute DP/DW (the W bump basis vector), which at present is a shortcoming of the scanline renderer.

```
void ComputeBumpVectors(const Point3 tv[3], const Point3 v[3],
```

```

Point3 bvec[3]) {
float uva,uvb,unc,uvd,uvk;
Point3 v1,v2;
uva = tv[1].x-tv[0].x;
uvb = tv[2].x-tv[0].x;
unc = tv[1].y-tv[0].y;
uvd = tv[2].y-tv[0].y;
uvk = uvb*unc - uva*uvd;
v1 = v[1]-v[0];
v2 = v[2]-v[0];
if (uvk!=0) {
    bvec[0] = (unc*v2-uvd*v1)/uvk;
    bvec[1] = (uva*v2-uvb*v1)/uvk;
}
else {
    if (uva!=0)
        bvec[0] = v1/uva;
    else if (uvb!=0)
        bvec[0] = v2/uvb;
    else
        bvec[0] = Point3(0.0f,0.0f,0.0f);
    if (unc!=0)
        bvec[1] = v1/unc;
    else if (uvd!=0)
        bvec[1] = v2/uvd;
    else
        bvec[1] = Point3(0.0f,0.0f,0.0f);
}
bvec[2] = Point3(0,0,1); // TBD- How to compute this ??
}

```

The three Point3's returned in bvec are stored away, and then simply returned in the function **SContext::GetBumpDP()**;

The function **UVGen::EvalDeriv()** evaluates the local derivative of the texture map in the U and V directions, taking into account symmetry and scaling.

The resulting derivative,  $dM$ , is scaled down from the value returned by `EvalDeriv` to keep the bump perturbations in a more reasonable range.

The perturbation vector is calculated as:

$$dM.x * dPdu + dM.y * dPdv$$

And then passed through the `texout->Filter` function, which is just the following:

```
Point3 Texout::Filter(Point3 c) {  
if (outAmt!=1.0f) c *= outAmt;  
if (flags&TEXOUT_INVERT) c = -c;  
return c;  
}
```

## Class Tex3D

See Also: [Class Texmap](#), [Working with Materials and Textures](#).

```
class Tex3D : public Texmap
```

### Description:

Developers that have created a 3D Studio/DOS SXP and a corresponding 3ds max texture plug-in may want to subclass from this class. It provides a way to have an instance of your 3ds max texture plug-in created automatically when the corresponding SXP is found in a 3DS file being imported.

This works as follows:

In the 3ds max texture plug-in's implementation of **DllMain()** the following function is called:

```
void RegisterSXPReader(TCHAR *sxpName, Class_ID cid);
```

The plug-in passes its own SXP name (i.e. "MARBLE\_I.SXP") and its own Class\_ID.

The system then remembers this. When the 3DStudio import plug-ins is loading a .3DS file and it encounters an SXP with this name, it will create an instance of the plug-in class (using the Class\_ID) and call the method of this class **ReadSXPData()**. The plug-in can then initialize itself with proper values by reading the old SXP data.

A sample plug-in that uses this technique is the 3ds max Marble texture. It imports the settings from the 3D Studio Marble SXP. See the sample code in **\MAXSDK\SAMPLES\MATERIALS\MARBLE.CPP**.

### Methods:

#### Prototype:

```
virtual void ReadSXPData(TCHAR *name, void *sxpdata)=0;
```

#### Remarks:

This method is called when the 3D Studio/DOS import plug-in encounters an SXP with the name registered by **RegisterSXPReader()**.

#### Parameters:

```
TCHAR *name
```

The name of the SXP.

**void \*sxpdata**

This is the SXP's initialization data. The plug-in can look at this data to see what numbers it was initialized to in the .3DS file being imported. It can then set its initial value to match the SXP settings.

## Class ImageFilter

See Also: [Class ImageFilterInfo](#), [Class Bitmap](#), [Class ITrackViewNode](#), [Class TimeChange](#), [Class UndoNotify](#), [Working with Bitmaps](#).

class ImageFilter

### Description:

Image processing filter plug-ins are derived from **ImageFilter**. This class has virtual methods the developer implements to provide information about the plug-in version, and description. The developer also implements the **Capability()** method to indicate the properties of the plug-in such as if it is a one pass filter or compositor, and whether it has a control dialog to be displayed.

The **Render()** method is the one that actually alters the source image to perform the work of the application.

Filter plug-ins have access to several bitmaps associated with the video post data stream. All filter plug-ins will have at least a pointer to data member **srcmap**.

This is Video Post's main image pipeline. Composition and transition (layer) filters will also receive a second bitmap (**frgmap**) which should be composited above the main bitmap (**srcmap**). If **mskmap** is not NULL, it will contain a pointer to a grayscale image to be used as a mask for the process. Note that developers should not delete these bitmaps as they are maintained by 3ds max.

If a plug-in is both a filter and a compositor, the plug-in can tell if it is running as a filter when the **frgmap** pointer is NULL.

Note: If changes are made to an **ImageFilter** plug-in, the system will not automatically put up the 'The scene has been modified. Do you want to save your changes?' dialog if the user attempts to exit without saving. So that your plug-in does not go unsaved, you should call the following global function if you make changes. This will indicate to the system that the save requester needs to be brought up:

**void SetSaveRequired(int b=TRUE);**

Sets the 'save dirty bit'. This will indicate to the system that the save requester needs to be presented to the user.

### Method Groups:

The hyperlinks below jump to beginning of related methods within the class:

[Constructor / Destructor](#)

[Max Interface Pointer](#)  
[Description/Copyright/Author/Version Methods](#)  
[Capabilities](#)  
[Dialog Methods](#)  
[Parameter Block Configuration Data Methods](#)  
[Linear Interpolation](#)  
[Rendering](#)  
[Preview Creation](#)  
[Channels](#)  
[Filter Control Dialog Interactivity](#)  
[Track View Node Methods](#)

### **Data Members:**

protected:

#### **BOOL interactive;**

This data member is available in release 2.0 and later only.  
TRUE if the setup dialog is interactive; otherwise FALSE.

#### **HWND vpSetuphWnd;**

The window handle of the video post setup dialog.

#### **HWND vphWnd;**

The window handle of the Video Post dialog.

#### **HWND dlghWnd;**

The window handle of the filter's setup dialogue when in "Interactive" mode.

#### **Bitmap \*srcmap**

The Source Bitmap (background). Note: The Video Post output resolution may be retrieved using this pointer. The width is **srcmap->Width()** and the height is **srcmap->Height()**.

#### **Bitmap \*mskmap**

The Image Mask Bitmap (for grayscale masking). This bitmap is at the Video Post output resolution size when the developer needs to access it in the **Render()** method.

#### **Bitmap \*frgmap**

The Foreground Bitmap (for layering/transitions). This bitmap is at the Video Post output resolution size when the developer needs to access it in the

**Render()** method.

**ImageFilterInfo \*ifi**

A pointer to an instance of the class that provides information about this filter and the video post queue.

public:

**TimeChange timeChange;**

This data member is available in release 2.0 and later only.

This class maintains this time change object so it may send

**FLT\_TIMECHANGED** messages.

**UndoNotify\* undonotify;**

This data member is available in release 2.0 and later only.

Points to an instance of the class that can be used so an **ImageFilter** plug-in can get notified on a change to its Track View Node.

**Methods:**

**Constructor / Destructor**

**Prototype:**

**ImageFilter();**

**Remarks:**

Constructor. The **srcmap**, **mskmap** and **frgmap** are set to NULL. The undo notify pointer is set to NULL and the interactive flag is set to FALSE.

**Prototype:**

**virtual ~ImageFilter();**

**Remarks:**

Destructor.

**Max Interface Pointer**

**Prototype:**

**virtual Interface \*Max();**

**Remarks:**

Implemented by the System.

This method returns an interface pointer for calling methods implemented in 3ds max. See [Class Interface](#).

**Description/Copyright/Author/Version****Prototype:**

```
virtual const TCHAR *Description() = 0;
```

**Remarks:**

Implemented by the Plug-In.

Returns an ASCII description of the filter plug-in (i.e. "Convolution Filter").

**Prototype:**

```
virtual const TCHAR *AuthorName() = 0
```

**Remarks:**

Implemented by the Plug-In.

Returns the name of the plug-in's author.

**Prototype:**

```
virtual const TCHAR *CopyrightMessage() = 0
```

**Remarks:**

Implemented by the Plug-In

Returns the plug-in ASCII Copyright message.

**Prototype:**

```
virtual UINT Version() = 0;
```

**Remarks:**

Implemented by the Plug-In.

Returns the plug-in version number \* 100 (i.e. v3.01 = 301).

**Capabilities**

**Prototype:**

**virtual DWORD Capability() = 0**

**Remarks:**

Implemented by the Plug-In.

This method returns a set of flags that describe the capabilities of this filter plug-in. These capabilities indicate if the plug-in is a filter, compositor, or has a control panel. To create a flag, "OR" the capabilities together, ie.

**(IMGFLT\_CONTROL | IMGFLT\_COMPOSITOR)**. Note: It is valid for a plug-in to both a Filter and a Compositor. If both flags are set, the user will be able to select it from both the Filter list and from the Compositor list. The plug-in will know it is running as a filter when the foreground map pointer, **frgmap**, is NULL.

**Return Value:**

See [List of Image Filter Capability Flags](#).

## Dialogs

**Prototype:**

**virtual void ShowAbout(HWND hWnd) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called by the system to display the About Box of the plug-in. This dialog is mandatory so the developer must implement this method.

**Parameters:**

**HWND hWnd**

The parent window handle of the dialog.

**Prototype:**

**virtual BOOL ShowControl(HWND hWnd)**

**Remarks:**

Implemented by the Plug-In.

This method is called by the system to display the control panel for the plug-in. This control panel is optional and its existence should be flagged by the

capability flag returned from **Capability()** (**IMGFLT\_CONTROL**). If a plug-in does not have a control panel, don't implement this method and let it default to FALSE.

**Parameters:**

**HWND hWnd**

The parent window handle of the dialog.

**Return Value:**

TRUE if the user selects OK to exit the dialog, and FALSE if the user selects Cancel.

**Default Implementation:**

```
{ return FALSE; }
```

## Rendering

**Prototype:**

```
virtual BOOL Render(HWND hWnd) = 0
```

**Remarks:**

Implemented by the Plug-In.

This is the method the plug-in implements to alter the image. This method performs the work of the filter or compositor.

**Parameters:**

**HWND hWnd**

The window handle to which the plug-in will be sending progress and abort check messages.

**Return Value:**

TRUE if the render was completed; otherwise FALSE (error or canceled by user).

**Sample Code:**

Below is an example of a render loop through each horizontal band of the image demonstrating the posting of messages. At the start of the loop the progress and check abort messages are sent. The progress message updates the Execute Video Post dialog with how much of the image has been processed. The check abort message allows the plug-in to detect if the user has canceled the operation.

```

BOOL result = TRUE;
BOOL abort = FALSE;
for (int iy = 0; iy < srcmap->Height(); iy++) {
    // Progress Report
    SendMessage(hWnd,FLT_PROGRESS,iy,srcmap->Height()-1);
    // Check for Abort
    SendMessage(hWnd,FLT_CHECKABORT,0,(LPARAM)(BOOL
*)&abort);
    if (abort) {
        result = FALSE;
        break;
    }
}
return(result);

```

Messages are sent via the Window API **SendMessage()** function. See [List of ImageFilter Related Messages](#).

## **Parameter Block Configuration Data**

Parameter Block Load and Save. A 'parameter block' is really nothing more than data a developer wants to load and save. See the sample code below to see how these methods are used to save the data in the UserSettable structure.

```

typedef struct userSettable {
int data1;
float data2;
} UserSettable;

```

```

DWORD MyFilter::EvaluateConfigure () {
return sizeof (UserSettable);
}

```

```

BOOL MyFilter::LoadConfigure (void *ptr) {
UserSettable *buf = (UserSettable *) ptr;
memcpy (&userSettings, ptr, sizeof(UserSettable));
return TRUE;

```

```
}
```

```
BOOL MyFilter::SaveConfigure (void *ptr) {  
if (ptr) {  
memcpy (ptr, &userSettings, sizeof(UserSettable));  
return TRUE;  
} else  
return FALSE;  
}
```

**Prototype:**

```
virtual DWORD EvaluateConfigure()
```

**Remarks:**

Implemented by the Plug-In.

The system will call this method to determine the buffer size required by the plug-in. The plug-in can save its parameter block in this buffer by implementing the **SaveConfigure()** method.

**Return Value:**

The number of bytes required by the plug-in's parameter block.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual BOOL LoadConfigure(void *ptr)
```

**Remarks:**

Implemented by the Plug-In.

This method will be called so the plug-in can load its parameter block.

Memory management is performed by 3ds max using standard **LocalAlloc()** and **LocalFree()**.

**Parameters:**

```
void *ptr
```

A pre-allocated buffer.

**Return Value:**

TRUE if the data was loaded OK; otherwise FALSE.

**Default Implementation:**

```
{ return (FALSE) };
```

**Prototype:**

```
virtual BOOL SaveConfigure(void *ptr)
```

**Remarks:**

Implemented by the Plug-In.

This method will be called so the plug-in can transfer its parameter block to the host.

**Parameters:**

```
void *ptr
```

A pre-allocated buffer the plug-in may write to.

**Return Value:**

TRUE if the data was saved OK; otherwise FALSE.

**Default Implementation:**

```
{ return (FALSE); }
```

**Linear Interpolation****Prototype:**

```
virtual int Lerp(int a, int b, int l);
```

**Remarks:**

This method is used internally.

**Prototype:**

```
virtual int Lerp(int a, int b, float f);
```

**Remarks:**

This method is used internally.

**Preview**

**Prototype:**

```
virtual BOOL CreatePreview(HWND hWnd, Bitmap **back, int
frame, int width, int height, float aspect, Bitmap **fore = NULL,
DWORD flag = PREVIEW_UP);
```

**Remarks:**

Implemented by the System.

This method provides a preview facility for use by plug-ins. This can be used by plug-ins that want to have a preview bitmap while displaying a control dialog.

Note: When you add a new Video Post event, an event object is created and it is added to the event queue when you hit the OK button. However, because the event is added to the queue only when you exit the dialogue, you cannot create a preview at that stage. Later, when you "Edit" the event, it is part of the queue and you can create a preview. Internally, there is no way for 3ds max to tell if the "Setup" button was called from an "Add" event or from an "Edit" event. Plug-In developers can tell if they are in the "Add" event mode by looking at the return value from this method. It will be FALSE if in "Add" mode since the call will fail.

**Parameters:****HWND hWnd**

This window handle will receive progress notification messages. The messages are defined in both BITMAP.H and FILTERS.H:

**FLT\_PROGRESS**

wParam: Current

lParam: Total

**FLT\_CHECKABORT**

wParam: 0

lParam: BOOL\*

**FLT\_TEXTMSG**

wParam: 0

lParam: LPCTSTR

**BMM\_PROGRESS**

wParam: Current

lParam: Total

### **BMM\_CHECKABORT**

wParam: 0

lParam: \*BOOL

### **BMM\_TEXTMSG**

wParam: 0

lParam: LPCTSTR

### **Bitmap \*\*back**

A pointer to the Bitmap Pointer (the Background). If the Bitmap pointer is NULL, a new bitmap is created using the given dimensions. This pointer must be NULL the first time this function is called as the bitmap must be created by Video Post. Once this function is called and a bitmap is returned, it is OK to call it again using this map. In this case, Video Post will simply use it instead of creating a new one. Note: You must NOT delete the bitmap when done -- Video Post will take care of it.

### **int frame**

The desired frame in TICKS. Note that this is the position of the Video Post frame slider (in TICKS) and not the main 3ds max frame slider. See the Advanced Topics section on [Time](#) for details on ticks.

### **int width**

The desired width of the preview.

### **int height**

The desired height of the preview.

### **Float aspect**

The desired aspect ratio of the preview.

### **Bitmap \*\*fore = NULL**

A pointer to the Bitmap Pointer (the Foreground). For layer plug-ins, this points to the foreground image. This is only valid if flag is set to **PREVIEW\_BEFORE**. In this case **back** will hold Video Post main queue and **fore** will have the foreground image to be composited. This is useful if you, a layer plug-in, want to collect the images and run a realtime preview. If flag is not **PREVIEW\_BEFORE**, **fore** will be a NULL pointer indicating there is no bitmap.

## **DWORD flag = PREVIEW\_UP**

The flag controls how much of the queue to run. The options are:

### **PREVIEW\_BEFORE**

The queue is run up to the event before the filter calling it.

### **PREVIEW\_UP**

The queue is run up to the event (filter) calling this function.

### **PREVIEW\_WHOLE**

The whole queue is run including events after this filter.

### **Return Value:**

TRUE if the creation was successful; otherwise FALSE.

## **Channels**

### **Prototype:**

```
virtual DWORD ChannelsRequired();
```

### **Remarks:**

Implemented by the Plug-In.

If a filter wants to work with the G-buffer (geometry/graphics buffer) it implements this method. It is used to indicate what image channels this plug-in requires. Prior to rendering 3ds max will scan the plug-ins in the chain of events and find out all the channels being requested. At the time the plug-in's **Render()** method is called, it will have access to these channels. The methods of class Bitmap may be used to access the channels.

Note: The generation of these channels should not normally be a default setting for the plug-in. These channels are memory intensive and if the plug-in won't use the channel it should not ask for it. Normally the plug-in would ask the user which channels to use and only then set the proper flags.

### **Return Value:**

See [List of Image Channels](#).

### **Default Implementation:**

```
{ return BMM_CHAN_NONE; }
```

## **Filter Control Dialog Interactivity**

**Prototype:**

```
void MakeDlgInteractive(HWND hWnd);
```

**Remarks:**

Implemented by the System.

This method is available in release 2.0 and later only.

This method may be used to allow a filter's dialog to operate interactively.

This means that a user can have the filter's control dialog up and still operate 3ds max and Track View at the same time. This method should be called from the filter's **Control()** method as part of the **WM\_INITDIALOG** code.

Note that even though this method can be called safely for any reason, developers should only call it when using animated parameters. It doesn't make sense to use it for filters with non-animated or no parameters. For sample code using this method see

**\MAXSDK\SAMPLES\POSTFILTERS\NEGATIVE\NEGATIVE.C**

**Parameters:**

**HWND hWnd**

The parent window handle.

**Sample Code:**

```
BOOL ImageFilter_Negative::Control(HWND hWnd,UINT  
message,WPARAM wParam,LPARAM lParam) {  
  switch (message) {  
    case WM_INITDIALOG: {  
      //-- Make Dialogue Interactive  
      MakeDlgInteractive(hWnd);  
      ...
```

**Prototype:**

```
virtual void FilterUpdate();
```

**Remarks:**

This method is available in release 2.0 and later only.

Whenever a filter instance is created or updated (i.e. the user went through the Filter Edit Control dialog) this method is called. The filter may use it to create or update its node controls. For an example see

**\MAXSDK\SAMPLES\POSTFILTERS\NEGATIVE\NEGATIVE.C**

**Default Implementation:**

```
{}
```

**Track View Node Methods**

**Prototype:**

```
BOOL IsInteractive();
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Returns TRUE if the control dialog is interactive; otherwise FALSE. This means a user can have the filter's control dialog up and still operate 3ds max and Track View at the same time.

**Prototype:**

```
virtual ITrackViewNode *CreateNode();
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

This method may be called to create a new Track View Node.

**Prototype:**

```
virtual ITrackViewNode *Node();
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

This method is used to return the Track View node for this filter.

## Class GBuffer

See Also: [Class GBufReader](#), [Class GBufWriter](#), [List of Image Channels](#), [List of GBuffer Channel Types](#), [Class Bitmap](#).

```
class GBuffer : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This class is used by Bitmaps to implement an enhanced G-Buffer (providing more functionality than the 2.x G-Buffer). The new G-Buffer stores multiple layers at each pixel. The Virtual Frame Buffer includes a new spinner to the right of the "channels" drop down which lets you look at the different layers. (It only appears when there are G-Buffer channels present).

The multiple layers of the G-Buffer allow rendering effects to do better antialiasing and handling of transparency. The frontmost layer is still stored in the G-Buffer as full, screen-sized arrays, one for each channel. The subsequent layers are stored in a variable number of "layer records" all of which are the same size (which depends on which channels have been requested via the **GBuffer::CreateChannels()** function). Even scenes with no transparency can have more than one layer at a given pixel. This can occur along the edges of an object, where the object partially covers some pixels, and another object (or objects) is visible in the remaining part of the pixel. Each visible face in the pixel will be represented by a fragment. A given object will not always appear on the same layer. If it is the only object in a pixel it will be in the frontmost layer, but where it is partially occluded it will lie in a layer behind the occluding object.

Another way multiple layers can occur (aside from transparency) is when objects are given the "Render Occluded Objects" property in the object properties dialog. When an object has this property, it acts as if it were transparent, and objects behind it, though invisible, are rendered and stored in the G-Buffer.

Another G-Buffer layer is added to contain the background color in its color channel for any pixel which is not empty and on which the background gets evaluated. This means the background layer will be present behind transparent objects or objects with "render occluded" turned on. It will also be present along the edges of objects where the background is partially visible in the pixel. Note: This does not depend at all on having the **GB\_BG** flag set. It is necessary to have the **GB\_COLOR** channel, however.

All methods of this class are implemented by the System.

**Note on RPF files:** The following information relates to the layer storage scheme used in the 3ds max RPF files.

The layer information for each scan line is stored as a series of run-encoded channels, each containing the same number of data elements, one per layer record. The first channel is a “psuedo-channel” that contains the x value for each layer record. If there are multiple layers for a given x, this will be reflected as several identical x values in a row.

For instance, if the x channel contained

5,6,7,7,8,8,8,9,10

And the Z channel contained

100, 100, 100,200, 100,200,300, 100, 100

This would mean at pixel 7 there are 2 layers, and pixel 8 there are 3 layers.

Pixel depths

5 100

6 100

7 100, 200

8 100, 200, 300

9 100

10. 100

As an extra note regarding RPF files, a new data member was added to [class RenderInfo](#), which is saved with the RPF (and RLA) files. In order to avoid getting out of sync with previous file format implementations a “version” word preceding the RenderInfo record has been added into the format. The version is set to a value that will distinguish it from the first word of the RenderInfo data, enabling it to determine the version. If we make further additions to RenderInfo in the future, the version will allow us to keep it straight. To load both older and newer RPF and RLA files correctly, the code that loads the RenderInfo been modified, and therefore any plugins and 3rd party code that read these files need to be changed. Here's my the code:

```

#define RENDINFO_VERS1 1000
int RLARender::ReadRenderInfo(RenderInfo *ri) {
short vers;
if (fread(&vers,sizeof(short),1,fd)!=1) return 0;
int size = sizeof(RenderInfo);
if (vers != RENDINFO_VERS1) {
// the old version didn't start with a version word, but
// with projType which is 0 or 1.
size -= sizeof(Rect); // The old record didn't have the region Rect.
fseek(fd,-sizeof(short),SEEK_CUR); // Undo the version read
}
if (ri) {
if (fread(ri,size,1,fd)!=1) return 0;
}
else
fseek(fd, size, SEEK_CUR);
return 1;
}

```

**The following functions are not part of this class but are available for use:**

**Function:**

**GBuffer \*NewDefaultGBuffer();**

**Remarks:**

Creates and returns a pointer to a new default G-Buffer. The buffer is essentially created empty.

A 'default' G-Buffer is one that 3ds max itself creates. See

**GBuffer::IsDefaultGBuffer()** below.

**Function:**

**void SetMaximumGBufferLayerDepth(int m);**

**Remarks:**

Sets the maximum GBuffer layer depth.

**Parameters:**

**int m**

The number to set.

**Function:**

**int GetMaximumGBufferLayerDepth();**

**Remarks:**

Returns the maximum GBuffer layer depth.

**Function:**

**int GBDataSize(int i);**

**Remarks:**

Returns the number of bytes per pixel for the specified channel.

**Parameters:**

**int i**

The index of the channel. See [List of G-Buffer Channel Indexes](#).

**Function:**

**TCHAR \*GBChannelName(int i);**

**Remarks:**

Returns the name of the specified channel.

**Parameters:**

**int i**

The index of the channel. See [List of G-Buffer Channel Indexes](#).

**Return Value:**

The name returned for the specified index:

GB\_Z (0): "Z"

GB\_MTL\_ID (1): "Material Effects"

GB\_NODE\_ID (2): "**Object**"  
GB\_UV (3): "**UV Coordinates**"  
GB\_NORMAL (4): "**Normal**"  
GB\_REALPIX (5): "**Non-Clamped Color**"  
GB\_COVERAGE (6): "**Coverage**"  
GB\_BG (7): "**Coverage Background**"  
GB\_NODE\_RENDER\_ID (8): "**Node Render ID**"  
GB\_COLOR (9): "**Color**"  
GB\_TRANSP10): "**Transparency**"  
GB\_VELOC (11): "**Velocity**"  
GB\_WEIGHT (12): "**Sub-Pixel Weight**"  
GB\_MASK (2): "**Sub-Pixel Coverage Mask**"

**Methods:**

public:

**Prototype:**

**virtual void SetRasterSize(int ww, int hh)=0;**

**Remarks:**

This method is used internally to set the size of the G-Buffer.

**Parameters:**

**int ww**

The width to set in pixels.

**int hh**

The height to set in pixels.

**Prototype:**

**virtual int Width()=0;**

**Remarks:**

Returns the width of the GBuffer.

**Prototype:**

**virtual int Height()=0;**

**Remarks:**

Returns the height of the GBuffer.

**Prototype:**

**virtual int InitBuffer()=0;**

**Remarks:**

Initializes the GBuffer. Call this method before writing to the buffer. If present, the GB\_Z channels is set to -BIGFLOAT and the GB\_NODE\_RENDER\_ID channel is set to 0xffff. The Render ID Name Table is cleared.

**Return Value:**

Nonzero on success; otherwise zero.

**Sample Code:**

```
GBuffer *gbuf = tobm->GetGBuffer();  
if (gbuf) gbuf->InitBuffer();
```

**Prototype:**

**virtual ULONG CreateChannels(ULONG channelMask)=0;**

**Remarks:**

This method creates the specified channels in this G-Buffer.

**Parameters:**

**ULONG channelMask**

Specifies the channels to create. See [List of Image Channels](#).

**Return Value:**

The channels that are currently present.

**Prototype:**

**virtual void DeleteChannels(ULONG channelMask)=0;**

**Remarks:**

This method delete specified channels.

**Parameters:**

## **ULONG channelMask**

Specifies the channels to create. See [List of Image Channels](#).

### **Return Value:**

The channels that are currently present.

### **Prototype:**

```
virtual ULONG ChannelsPresent()=0;
```

### **Remarks:**

Returns the channels that are currently present. See [List of Image Channels](#).

### **Prototype:**

```
virtual void *GetChannel(ULONG channelId, ULONG&  
chanType)=0;
```

### **Remarks:**

Returns a pointer to the specified channel of the G-Buffer, and determines its type in terms of bits per pixel. NULL is returned if the channel can't be found.

### **Parameters:**

**ULONG channelId**

The channel to get. See [List of Image Channels](#).

**ULONG& chanType**

The channel type. See [List of G-Buffer Channel Types](#).

### **Prototype:**

```
virtual GBufReader *CreateReader()=0;
```

### **Remarks:**

Creates and returns a pointer to a **GBufReader** object.

### **Prototype:**

```
virtual void DestroyReader(GBufReader *pRdr)=0;
```

### **Remarks:**

Deletes the specified **GBufReader** object.

### **Parameters:**

**GBufReader \*pRdr**

Points to the reader to delete.

**Prototype:**

**virtual GBufWriter \*CreateWriter()=0;**

**Remarks:**

Creates and returns a pointer to a **GBufWriter** object.

**Prototype:**

**virtual void DestroyWriter(GBufWriter \*pRdr)=0;**

**Remarks:**

Deletes the specified **GBufWriter** object.

**Parameters:**

**GBufWriter \*pRdr**

Points to the writer to delete.

**Prototype:**

**virtual BOOL IsDefaultGBuffer();**

**Remarks:**

Returns TRUE if this is a default G-Buffer; otherwise FALSE. A 'default' G-Buffer is one that 3ds max itself creates. Since GBuffer is a virtual class a plug-in developer could create their own G-Buffers (for some very special purpose). In that case it would not be a 'default' G-Buffer and this method would return FALSE.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Deletes this G-Buffer.

**Prototype:**

**virtual void Copy(GBuffer \*gbfrom)=0;**

**Remarks:**

Copies the data from the specified G-Buffer to this one. This method maintains the multi-layer data correctly. The channels present in each G-Buffer must match.

**Parameters:**

**GBuffer \*gbfrom**

Points to the source of the copy.

**Prototype:**

**virtual void CopyScale(GBuffer \*gbfrom, int cw=-1, int ch=-1)=0;**

**Remarks:**

Copies the data from the specified G-Buffer and optionally scales its size while maintaining the multi-layer data correctly. This is used in Video Post whenever there is an input bitmap that is a different size than the current Video Post image dimensions.

**Parameters:**

**GBuffer \*gbfrom**

Points to the source of the copy.

**int cw=-1**

The width for the copy. If -1 is specified the copy is the same size as the original.

**int ch=-1**

The height for the copy. If -1 is specified the copy is the same size as the original.

**Prototype:**

**virtual void Position(int srcy, int trgy, int srcx, int trgx, int trgw, int trgh)=0;**

**Remarks:**

This method is used internally and is something that normally a developer shouldn't call directly. It is used inside the Bitmap routines for manipulating bitmaps. It takes a sub rectangle of the G-Buffer which has an upper left corner at (srcx,srcy), and dimensions of (trgw,trgh), and moves it to be

positioned at (trgx,trgy). Portions of the G-Buffer outside of this rectangle are discarded.

**Parameters:**

**int srcy**

The source y location.

**int trgy**

The target y location.

**int srcx**

The source x location.

**int trgx**

The target x location.

**int trgw**

The target width.

**int trgh**

The target height.

**Prototype:**

**virtual int NumberLayerRecords(int y)=0;**

**Remarks:**

Returns the total number of layer records for the specified scan line. On each scan line there is a single array of layer records. Each layer record contains an x value which tells at what pixel it is located, and then all of the G-Buffer data that is being stored in this particular G-Buffer. The layer records for a given pixel all have the same x value, and are ordered front-to-back in scene depth.

This method is used by the RLA and RPF writer code. See **\MAXSDK\SAMPLES\IO\RLA\RLA.CPP** for an example.

**Parameters:**

**int y**

The zero based index of the scan line.

**Prototype:**

**virtual int GetLayerChannel(int y, int ichan, char \*data)=0;**

**Remarks:**

This method goes through all the layer records and pulls out the values for a particular channel into an array, which is handy for run-encoding when saving to a file.

**Parameters:**

**int y**

The zero based index of the scan line.

**int ichan**

The channel to check. See [List of G-Buffer Channel Indexes](#). When **ichan=-1**, it gives an array of the x values from each of the layer records.

**char \*data**

Points to storage for the layer data.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual int CreateLayerRecords(int y, int num)=0;**

**Remarks:**

This method creates the specified number of layer records for the scan line passed.

**Parameters:**

**int y**

The zero based index of the scan line.

**int num**

The number of records to create.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual int SetLayerChannel(int y, int ichan, char \*data)=0;**

**Remarks:**

Sets the layer record data for the specified channel for the scan line passed.

**Parameters:**

**int y**

The zero based index of the scan line.

**int ichan**

The channel to set. See [List of G-Buffer Channel Indexes](#). When **ichan=-1**, it sets an array of the x values from each of the layer records.

**char \*data**

Points to storage for the layer data.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual void UpdateChannelMinMax()**=0**;**

**Remarks:**

This method searches the G-Buffer Z, UV and VELOC channels and stores the minimum and maximum values. This enables the **GetChannelMin()** and **GetChannelMax()** methods to work. This is called internally when 3ds max generates a G-Buffer. When developers get one from the renderer this method will already have been called.

**Prototype:**

**virtual BOOL GetChannelMin(int chan, void \*data)**=0**;**

**Remarks:**

Retrieves the minimum value in the specified channel.

**Parameters:**

**int chan**

The channel to check. One of the following (the other channels are not supported):

**GB\_Z**

Z-Buffer depth - **float**

**GB\_UV**

UV coordinates - **Point2**

**GB\_VELOC**

Velocity - **Point2**

**void \*data**

Points to storage for the value to get.

**Return Value:**

TRUE if **chan** was valid and the value could be stored in **data**; otherwise FALSE.

**Prototype:**

**virtual BOOL GetChannelMax(int chan, void \*data)=0;**

**Remarks:**

Retrieves the maximum value in the specified channel.

**Parameters:**

**int chan**

The channel to check. One of the following (the other channels are not supported):

**GB\_Z**

Z-Buffer depth - **float**

**GB\_UV**

UV coordinates - **Point2**

**GB\_VELOC**

Velocity - **Point2**

**void \*data**

Points to storage for the value to get.

**Return Value:**

TRUE if **chan** was valid and the value could be stored in **data**; otherwise FALSE.

**Prototype:**

**virtual NameTab &NodeRenderIDNameTab()=0;**

**Remarks:**

This method returns a reference to the table of node names indexed by their **NodeRenderId**. See [Class NameTab](#).

**Prototype:**

**virtual Tab<float> &ImageBlurMultiplierTab()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns a reference to a table of floats. These are image blur multipliers indexed by NodeRenderID.

**Prototype:**

**INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);**

**Remarks:**

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API. This is currently reserved for future use.

**Parameters:**

**int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value (defined uniquely for each **cmd**).

## Class Effect

See Also: [Class SpecialFX](#), [Class SFXParamDlg](#), [Class IRendParams](#), [Class RenderGlobalContext](#), [Class Bitmap](#), [Class ISave](#), [Class ILoad](#), [Class INode](#), [Class Interface](#).

```
class Effect : public SpecialFX
```

### Description:

This class is available in release 3.0 and later only.

This is the base class used in the creation of Rendering Effect plug-ins. In 3ds max 3.0 these plug-in are added in sequence after a rendering is complete without the use of Video Post. A developer creates a sub-class of this class and implements or calls the methods shown below.

### Plug-In Information:

Class Defined In RENDER.H

Super Class ID RENDER\_EFFECT\_CLASS\_ID

Standard File Name Extension DLV

Extra Include File Needed None

There are also methods in the Interface class for manipulating the Effects List:

```
virtual int NumEffects()=0;  
virtual Effect *GetEffect(int i)=0;  
virtual void SetEffect(int i,Effect *e)=0;  
virtual void AddEffect(Effect *eff)=0;  
virtual void DeleteEffect(int i)=0;
```

### Methods:

public:

### Prototype:

```
virtual EffectParamDlg *CreateParamDialog(IRendParams *ip);
```

### Remarks:

Implemented by the Plug-In.

This method creates and returns a new instance of a class derived from **EffectParamDlg** to manage the user interface. This put up a modal dialog that lets the user edit the plug-ins parameters.

**Parameters:****IRendParams \*ip**

This is the interface given to the rendering effect so it may display its parameters.

**Return Value:**

A new instance of a class derived from **EffectParamDlg**.

Note: **typedef SFXParamDlg EffectParamDlg;**

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual BOOL SetDlgThing(EffectParamDlg* dlg);
```

**Remarks:**

You should implement this if you are using the ParamMap2 AUTO\_UI system and the effect has secondary dialogs that have something other than the incoming effect as their 'thing'. Called once for each secondary dialog for you to install the correct thing. Return TRUE if you process the dialog, FALSE otherwise, in which case the incoming effect will be set into the dialog.

Note: Developers needing more information on this method can see the remarks for [MtlBase::CreateParamDlg\(\)](#) which describes a similar example of this method in use (in that case it's for use by a texture map plug-in).

**Parameters:****EffectParamDlg\* dlg**

Points to the ParamDlg.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual DWORD GBufferChannelsRequired(TimeValue t);
```

**Remarks:**

Implemented by the Plug-In.

Returns a DWORD that indicates the channels that this Effect requires in the

output bitmap.

**Parameters:**

**TimeValue t**

The time at which the channels are required.

**Return Value:**

The required channels. See [List of Image Channels](#).

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual void Apply(TimeValue t, Bitmap *bm,  
RenderGlobalContext *gc, CheckAbortCallback *cb)=0;
```

**Remarks:**

Implemented by the Plug-In.

This is the method that is called to apply the effect to the bitmap passed.

**Parameters:**

**TimeValue t**

The time at which to apply the effect.

**Bitmap \*bm**

The bitmap the effect modifies and stores the result in.

**RenderGlobalContext \*gc**

This can be used to retrieve information about the global rendering environment.

**CheckAbortCallback \*cb**

Points to an object whose **Check()** method may be called to determine if the user wants to abort. See [Class CheckAbortCallback](#).

**Prototype:**

```
IOResult Save(ISave *isave);
```

**Remarks:**

Implemented by the System.

To facilitate naming rendering effects a 'name' string exists in the base class.

This method should be called at the start of the developer's sub-classed Effect plug-in to save the name.

**Parameters:**

**ISave \*isave**

An interface for saving data.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

Implemented by the System.

To facilitate naming rendering effects a 'name' string exists in the base class.

This method should be called at the start of the developer's sub-classed Effect plug-in to load the name.

**Parameters:**

**ILoad \*iload**

An interface for loading data.

## Class Osnap

See Also: [Class OsnapHit](#), [Class IOsnapManager](#), [Class OsnapMarker](#), [Structure SnapInfo](#) The Advanced Topics section on [Snapping](#).

```
class Osnap
```

### Description:

This class is available in release 2.0 and later only.

This is the base class for the derivation of a new object snap plug-ins.

Conceptually, the osnap class represents a "rule" for locating points in an object's local space. Typically, an instance of this class will only make sense for certain object types. It's the job of the **ValidInput()** method to filter out uninteresting nodes in the scene. When the scene is traversed, each object which passes the input test will be passed into the **Snap()** method. This method is the workhorse of object snap plug-ins and is responsible for computing, allocating and recording its hits.

For convenience, an object snap plug-in may encompass multiple sub-rules. For example, the shape snap contains rules for computing both tangents and perpendicular points on splines. Therefore many of the methods have an index argument which identifies which sub-snap it applies to.

For sample code see

```
\MAXSDK\SAMPLES\SNAPS\SPHERE\SPHERE.CPP.
```

### Friend Classes:

```
friend class OsnapHit;  
friend class OsnapManager;
```

### Methods:

#### Prototype:

```
Osnap();
```

#### Remarks:

Constructor. The **OsnapManger** interface pointer is initialized.

#### Prototype:

```
virtual ~Osnap();
```

**Remarks:**

Destructor. Internal data structure to the class are freed.

**Prototype:**

```
virtual int numsubs();
```

**Remarks:**

Returns the number of sub-snaps this plug-in provides.

**Default Implementation:**

```
{return 1;}
```

**Prototype:**

```
virtual BOOL IsActive();
```

**Remarks:**

Implemented by the system.

Returns TRUE if any of the sub-snaps are active; otherwise FALSE.

**Prototype:**

```
virtual BOOL GetActive(int index);
```

**Remarks:**

Implemented by the system.

Returns TRUE if any of the indexed sub-snap is active.

**Prototype:**

```
virtual void SetActive(int index, BOOL state);
```

**Remarks:**

Implemented by the system.

Sets the indexed sub-snap to the specified state.

**Prototype:**

```
virtual TCHAR *Category();
```

**Remarks:**

Returns the category string for this plug-in. If the plug-in fails to override this

method, the snap will be added to the "standard" page of the UI.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual Class_ID ClassID();
```

**Remarks:**

Returns the Class\_ID for this plug-in.

**Default Implementation:**

```
{ return Class_ID( 0, 0); }
```

**Prototype:**

```
virtual boolean ValidInput(SClass_ID scid, Class_ID cid)=0;
```

**Remarks:**

This method is used to check if the object whose super class ID and class ID are passed is valid input for this object snap plug-in to snap to.

**Parameters:**

**SClass\_ID scid**

The Super Class ID to check.

**Class\_ID cid**

The Class ID to check.

**Return Value:**

Returns TRUE if the object is OK to snap to; otherwise FALSE.

**Sample Code:**

```
boolean SphereSnap::ValidInput(SClass_ID scid, Class_ID cid){  
    boolean c_ok = FALSE, sc_ok = FALSE;  
    sc_ok |= (scid == GEOMOBJECT_CLASS_ID)? TRUE : FALSE;  
    c_ok |= (cid == Class_ID(SPHERE_CLASS_ID,0))? TRUE :  
    FALSE;  
    return sc_ok && c_ok;  
}
```

**Prototype:**

```
virtual void Snap(Object* pObj, IPoint2 *p, TimeValue t);
```

**Remarks:**

Implemented by the Plug-In.

This is the workhorse of a snap plug-in. This method should compute and record hits on the given object.

**Parameters:****Object\* pObj**

A pointer to an object which passed the valid input test. Note that if this method is called, you can make certain assumption about the class of the object and do appropriate casting as needed.

**IPoint2 \*p**

The cursor position.

**TimeValue t**

The time at which to check.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL UseCallbacks();
```

**Remarks:**

Developers have the option of placing all their code in a single **Snap()** method or of breaking it up into multiple callbacks. Developers wishing to use callbacks should override this method to return TRUE. Note: if callbacks are used, the **Snap()** method is **not** called.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual int NumCallbacks();
```

**Remarks:**

Returns the number of callbacks used.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual SnapCallback GetSnapCallback(int sub);
```

**Remarks:**

Returns the specified callback to be used for snapping.

**Parameters:**

**int sub**

The sub-snap index.

**Return Value:**

Note the following typedef -- a **SnapCallback** is simply a pointer to a function passed two arguments:

```
typedef void (*SnapCallback) (Object* pObj, IPoint2 *p) ;
```

**Default Implementation:**

```
{ return NULL;}
```

**Prototype:**

```
virtual BOOL GetSupportedObject(INode *iNode, TimeValue t,  
ObjectState *os);
```

**Remarks:**

This method is provided for future use so that snaps can evaluate the object at arbitrary points in the pipeline. Returns TRUE if the object associated with the node passed is supported; otherwise FALSE. The default implementation calls **ValidInput()** and fills the storage pointed to by os with the object state at the end of the geometry pipeline. This is the same object state returned by EvalWorldState.

**Parameters:**

**INode \*iNode**

The node whose object being checked.

**TimeValue t**

The time at which to check the object.

## **ObjectState \*os**

This pointer should be updated to the **ObjectState** of the object associated with the node by calling **INode::EvalWorldState()**.

### **Default Implementation:**

```
BOOL Osnap::GetSupportedObject(INode *inode, TimeValue t, ObjectState *os) {  
    *os = inode->EvalWorldState(t);  
    assert(os);  
    Class_ID thistype = os->obj->ClassID();  
    unsigned long supertype = os->obj->SuperClassID();  
    return ValidInput(supertype,thistype)?TRUE:FALSE;  
}
```

### **Prototype:**

```
virtual TSTR *snapname(int index)=0;
```

### **Remarks:**

Returns a pointer to the snap's name to be displayed in the user interface.

### **Parameters:**

**int index**

The index of the sub-snap whose name is returned.

### **Prototype:**

```
virtual TSTR *tooltip(int index);
```

### **Remarks:**

Reserved for future use.

### **Parameters:**

**int index**

The index of the sub-snap whose name is returned.

### **Default Implementation:**

```
{return NULL;}
```

**Prototype:**

**virtual OsnapMarker \*GetMarker(int index)=0;**

**Remarks:**

Implemented by the Plug-In.

This method should return a pointer to a (typically static) OsnapMarker. These markers define the identifying markers which get displayed in the viewports.

**Parameters:**

**int index**

The subsnap whose marker the system requires.

**Return Value:**

A pointer to an OsnapMarker. If this method returns NULL, a default marker will be displayed.

**Prototype:**

**virtual WORD HiliteMode();**

**Remarks:**

Implemented by the Plug-In.

Returns a value to indicate the type of highlighting to be done for this snap.

Typically, some part of the objects geometry is illuminated. In some cases it is desirable to illuminate the objects bounding box and occasionally to draw a world space crosshair. The default implementations should normally be used.

**Return Value:**

One or more of the following values:

**HILITE\_NORMAL**

This is the default and indicates that some part of the objects geometry will be hilited. The description of this geometry is recorded in the hitmesh member of the class OsnapHit.

**HILITE\_BOX**

This is return value indicates that the objects bounding box should be drawn as the result of a hit on this object.

**HILITE\_CROSSHAIR**

Reserved for grid snapping. This return value indicates that a world space crosshair should be drawn through the hitpoint.

**Default Implementation:**

```
{return HILITE_NORMAL;}
```

**Prototype:**

```
virtual boolean BeginUI(HWND hwnd);
```

**Remarks:**

This method is reserved for future use.

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
virtual void EndUI(HWND hwnd);
```

**Remarks:**

This method is reserved for future use.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual HBITMAP getTools()=0;
```

**Remarks:**

Returns a handle to a bitmap that contains the icons to be displayed for this snap. If there are N subsnaps, this bitmap should contain N icons. The size of an individual icon is 16x15.

**Prototype:**

```
virtual HBITMAP getMasks()=0;
```

**Remarks:**

Returns a handle to a bitmap that contains the masks for the UI icons for this snap plug-in.

**Prototype:**

```
virtual BOOL HitTest(Object* pObj, IPoint2 *p, TimeValue t);
```

**Remarks:**

Implemented by the Plug-In.

Developers may override this method to do additional hit testing on each object as an additional rejection criteria. The default implementation returns TRUE and consequently filters nothing. Note that if this method returns FALSE for a given object, the snap method will never be called for it.

Note: Nodes are always trivially rejected if the cursor position does not fall within the screen space bounding box of the node.

**Parameters:****Object\* pObj**

A pointer to the object returned by GetSupportedObject.

**IPoint2 \*p**

The cursor position.

**TimeValue t**

The time at which to hittest.

**Return Value:**

Returns TRUE if the object is being hit and should be considered for snapping.

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
virtual WORD AccelKey(int index)=0;
```

**Remarks:**

This method is no longer used.

# Class BitmapIO

See Also: [Class Bitmap](#), [Class BitmapStorage](#), [Class BitmapInfo](#), [Working with Bitmaps](#).

```
class BitmapIO : public BaseInterfaceServer
```

## Description:

This is the base class used by developers creating image loader / saver plug-ins. Developers implement pure virtual methods of this class to load the image, open it for output, write to it, close it, and to provide information about the image loader/saver they are creating. These are properties such as the author name, copyright, image format description, filename extension(s) used, and the capabilities of the image loader / saver.

When a BitmapIO derived image reader reads an image, it creates a storage class that makes sense to it. For example, a paletted 8 bit is perfect for loading GIF's but not for loading 32 bit Targas. The inverse is also true. There is no point in creating a TRUE\_64 storage to load a GIF. Because this is how image buffers are managed, it is also important to note that if a developer writes an image loader that creates images from scratch (a gradient generator for instance), there is no need to have any real memory allocated. The plug-in would simply derive a new type of **BitmapStorage** and provide the pixels through the common methods (virtual buffer), creating them as they are requested.

## Method Groups:

The hyperlinks below jump to the start of related methods within the class:

[Loading](#)

[Output/Write/Close](#)

[Filename Extension](#)

[Author/Description/Copyright/Version](#)

[Capabilities](#)

[Parameter Block Methods](#)

[ShowAbout/ShowImage/ShowControl](#)

[Image Information](#)

[Gamma Setting](#)

[Dithering](#)

[DIB Access](#)

[Palette Calculation](#)

[Storage / Bitmap Access](#)

[Silent Mode Setting](#)

[G-Buffer Channels Required](#)

[Internal Methods](#)

## Data Members:

protected:

**float gamma;**

The gamma setting.

**Bitmap \*map;**

The Bitmap using this OUTPUT handler.

**BitmapStorage \*storage;**

The storage used by this INPUT handler.

**int openMode;**

The mode that the IO module is open for. See [List of Bitmap Open Mode Types](#).

**BitmapIO \*prevIO;**

A linked list pointer to the previous IO module for multiple outputs of a single bitmap.

**BitmapIO \*nextIO;**

A linked list pointer to the next IO module for multiple outputs of a single bitmap.

public:

**BitmapInfo bi;**

Describes the properties of the bitmap being handled by the instance of this class.

## Methods:

### Output Pixels

public:

#### Prototype:

```
int GetOutputPixels(int x,int y,int pixels,BMM_Color_64  
*ptr,BOOL preMultAlpha=TRUE);
```

**Remarks:**

Implemented by the System.

This method is used by the subclassed BitmapIO to get pixels for output with the appropriate output gamma correction.

**Parameters:****int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_64 \*ptr**

Pointer to storage for the retrieved pixels. See [Structure BMM\\_Color\\_64](#).

**BOOL preMultAlpha=TRUE**

This parameter is available in release 3.0 and later only.

Setting this parameter to FALSE will cause pixels with non-premultiplied alpha to be returned.

**Return Value:**

Nonzero if the pixels were retrieved; otherwise zero.

**Prototype:**

```
int GetDitheredOutputPixels(int x,int y,int pixels,BMM_Color_32 *ptr,BOOL preMultAlpha=TRUE);
```

**Remarks:**

Implemented by the System.

This method is used by the subclassed BitmapIO to get 32 bit pixels for output with the appropriate output gamma correction and dither. Note that this method works on only a single scanline of pixels at a time.

**Parameters:****int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_32 \*ptr**

Pointer to storage for the retrieved pixels. See [Structure BMM\\_Color\\_32](#).

**BOOL preMultAlpha=TRUE**

This parameter is available in release 3.0 and later only.

Setting this parameter to FALSE will cause pixels with non-premultiplied alpha to be returned.

**Return Value:**

Nonzero if the pixels were retrieved; otherwise zero.

**DIB Access****Prototype:**

**PBITMAPINFO GetOutputDib(int depth = 24);**

**Remarks:**

Implemented by the System.

This method is used by the subclassed BitmapIO to get a DIB for output with the appropriate output gamma correction.

**Parameters:**

**int depth = 24**

Specifies the depth of the DIB. This may be either 24 or 32.

**Prototype:**

**PBITMAPINFO GetDitheredOutputDib(int depth = 24);**

**Remarks:**

Implemented by the System.

This methods is used by the subclassed BitmapIO to get a DIB for output with the appropriate output gamma correction and dither.

**Parameters:**

**int depth = 24**

Specifies the depth of the DIB. This may be either 24 or 32.

## Output Gamma Setting

### Prototype:

**float OutputGamma();**

### Remarks:

Implemented by the System.  
Returns the output gamma setting.

## Dithering

### Prototype:

**BOOL DitherTrueColor();**

### Remarks:

Implemented by the System.  
If a BitmapIO wants to do its own dithering, it should call this method to find out if dithering is wanted for true color images. If it is a 24 bit or 32 bit format, it would usually just call **GetDitheredOutputPixels()** instead.

### Return Value:

TRUE if dithering is desired; otherwise FALSE.

### Prototype:

**BOOL DitherPaletted();**

### Remarks:

Implemented by the System.  
If a BitmapIO wants to do its own dithering, it should call this method to find out if dithering is wanted for paletted images.

### Return Value:

TRUE if dithering is desired; otherwise FALSE.

## Palette Computation

### Prototype:

**int CalcOutputPalette(int palsize, BMM\_Color\_48 \*pal);**

**Remarks:**

Calculate a color palette for output color packing for the map that is using this output handler (this is the map pointed at by the protected data member **Bitmap \*map**). This method performs gamma correction. See [Class ColorPacker](#), [Class Quantizer](#).

**Parameters:****int palsize**

The size of the palette to compute.

**BMM\_Color\_48 \*pal**

Storage for the palette.

**Return Value:**

Nonzero if the palette was computed; otherwise zero.

## Open Mode Setting

**Prototype:****inline int OpenMode()****Remarks:**

Implemented by the System.

Returns the open mode setting. See [Bitmap Open Mode Types](#).

## Storage / Bitmap Access

**Prototype:****BitmapStorage \*Storage()****Remarks:**

Implemented by the System.

Returns a pointer to the BitmapStorage for this image input handler.

**Prototype:****inline Bitmap \*Map()****Remarks:**

Implemented by the System.

Returns a pointer to the Bitmap using this output handler.

## Filename Extensions

### Prototype:

```
virtual int ExtCount() = 0
```

### Remarks:

Implemented by the Plug-In.

Returns the number of filename extensions supported by this IO module. For example the EPS plug-in supports "EPS" and "PS", and thus returns 2.

### Prototype:

```
virtual const TCHAR *Ext( int n ) = 0;
```

### Remarks:

Implemented by the Plug-In.

The extensions are accessed using a virtual array. This method returns the 'i-th' filename extension supported by the IO module, (i.e. "EPS").

### Parameters:

**int i**

Specifies which filename extension to return.

## Author/Desc/Copyright/Version

### Prototype:

```
virtual const TCHAR *LongDesc( ) = 0
```

### Remarks:

Implemented by the Plug-In.

Returns a long ASCII description of the image format (i.e. "Targa 2.0 Image File").

### Prototype:

```
virtual const TCHAR *ShortDesc() = 0
```

### Remarks:

Implemented by the Plug-In.

Returns a short ASCII description of the image format (i.e. "Targa").

**Prototype:**

**virtual const TCHAR \*AuthorName() = 0**

**Remarks:**

Implemented by the Plug-In.

Returns the ASCII Author name of the IO module.

**Prototype:**

**virtual const TCHAR \*CopyrightMessage() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the ASCII Copyright message for the IO module.

**Prototype:**

**virtual UINT Version() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the IO module version number \* 100 (i.e. v3.01 = 301)

## Capabilities

**Prototype:**

**virtual int Capability() = 0;**

**Remarks:**

Implemented by the Plug-In.

Returns the IO module capability flags. These describe the capabilities of the plug-in such as if it supports reading images, writing images, multiple files, and whether it has its own information and control dialog boxes. See

[BitmapIO Capability Flags](#).

## **ShowAbout / ShowImage / ShowControl**

### **Prototype:**

**virtual void ShowAbout( HWND hWnd ) = 0;**

### **Remarks:**

Implemented by the Plug-In.

This method is called to show the plug-in's "About" box. This is called, for example, from the About button of the Add Image Input Event dialog in Video Post.

### **Parameters:**

**HWND hWnd**

The handle of the owner window.

### **Prototype:**

**virtual BOOL ShowImage(HWND hWnd, BitmapInfo \*bi)**

### **Remarks:**

Implemented by the Plug-In.

If the **BMMIO\_OWN\_VIEWER** flag is set in the flags returned from the **Capability()** method, this method will be called whenever the user wants to view an image for this device. This is for devices which can "play" image sequences such as AVIs, FLCs, etc.

### **Parameters:**

**HWND hWnd**

The handle of the owner window.

**BitmapInfo \*bi**

The bitmap to view.

### **Return Value:**

TRUE if the viewing the image was successful; otherwise FALSE.

### **Default Implementation:**

```
{ return FALSE; }
```

### **Prototype:**

## **virtual BOOL ShowControl( HWND hWnd, DWORD flag )**

### **Remarks:**

Implemented by the Plug-In.

Displays the Control Panel of the IO module. This function is only called if the plug-in has defined it supports it (through the Capability flag returned from **Capability()**, ie. **BMMIO\_CONTROLREAD**, etc.). See [BitmapIO Capability Flags](#).

### **Parameters:**

#### **HWND hWnd**

The handle of the owner window.

#### **DWORD flag**

Indicates to the plug-in what operation the control is for (read, write, or generic). See [BitmapIO Capability Flags](#)

### **Return Value:**

If the user exits the box through an OK, this function should return TRUE. If the user cancels out, it will should FALSE. FALSE indicates nothing has changed so the system won't bother asking the plug-in if it wants to save data.

### **Default Implementation:**

```
{ return FALSE; }
```

## **Parameter Block Methods**

The following methods (**EvaluateConfigure()**, **LoadConfigure()**, **SaveConfigure()**) deal with parameter block loading and saving. See the sample code below to see how the EPS BitmapIO plug-in uses these methods.

```
typedef struct userSettable {  
    int units; // Inches or MM  
    int binary; // Whether want binary image data or not  
    int preview; // Whether want TIFF preview in file  
    int orientation; // Options are portrait or landscape  
    int colorType; // Whether image is output as rgb or gray  
    float paperHeight; // Height of output (for centering image)  
    float paperWidth; // Width of output (for centering image)  
    float xResolution; // In dots per inch
```

```
float yResolution; // In dots per inch
} UserSettable;
```

```
DWORD BitmapIO_EPS::EvaluateConfigure () {
return sizeof (UserSettable);
}
```

```
BOOL BitmapIO_EPS::LoadConfigure (void *ptr) {
UserSettable *buf = (UserSettable *) ptr;
memcpy (&userSettings, ptr, sizeof(UserSettable));
return TRUE;
}
```

```
BOOL BitmapIO_EPS::SaveConfigure (void *ptr) {
if (ptr) {
memcpy (ptr, &userSettings, sizeof(UserSettable));
return TRUE;
} else
return FALSE;
}
```

**Prototype:**

```
virtual DWORD EvaluateConfigure( ) = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called by 3ds max to determine the buffer size required by the plug-in. The plug-in implements this method and returns the number of bytes of configuration data it needs to save.

**Return Value:**

The buffer size required by the plug-in (in bytes).

**Prototype:**

```
virtual BOOL LoadConfigure( void *ptr ) = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called by 3ds max to allow the plug-in to load any configuration data.

**Parameters:**

**void \*ptr**

Pointer initialized to point to the previously saved configuration data.

**Return Value:**

Returns TRUE if the data was loaded properly; otherwise FALSE.

**Prototype:**

**virtual BOOL SaveConfigure( void \*ptr ) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called by 3ds max to allow the plug-in to save any configuration data.

**Parameters:**

**void \*ptr**

Pointer initialized to a pre-allocated buffer where the plug-in may save data.

**Return Value:**

Returns TRUE if the data was saved; otherwise FALSE.

**Silent Mode Setting****Prototype:**

**BOOL SilentMode()**

**Remarks:**

Implemented by the System.

Returns the state of the silent mode flag. If this flag is TRUE the plug-in should NOT post a dialog displaying any error messages.

**Prototype:**

## **BMMRES GetFrame(BitmapInfo \*fbi, int \*frame);**

### **Remarks:**

Implemented by the System.

This method is for use with multi-frame sequences. It processes the desired frame based on the user options. For example the user can tell the system to hold on the last frame of the sequence, loop back to the beginning, or return an error. This method does all the checking automatically based on the BitmapInfo object passed and computes the proper frame number.

### **Parameters:**

#### **BitmapInfo \*fbi**

A pointer to the BitmapInfo that contains the user options. This is the instance passed to **Load()**.

#### **int \*frame**

A pointer to an integer to receive the frame number

### **Return Value:**

One of the following values:

**BMMRES\_SUCCESS**

**BMMRES\_BADFRAME**

## Critical Error Handling

### Prototype:

**BMMRES ProcessImageIOError(BitmapInfo \*bi, TCHAR \*string = NULL);**

### Remarks:

Implemented by the System.

This method may be called to present the user with the 3ds max Image IO Error dialog box. The dialog displays the bitmap file name or device name, and the specified error message. The user may choose Cancel or Retry from the dialog. An appropriate value is returned based on the users selection.

This method is used to handle hardware I/O errors automatically or display the given string.

In this method, if Silent Mode is on (for example network rendering is being done), no dialog is presented and **BMMRES\_ERRORTAKENCARE** is returned.

### Parameters:

#### **BitmapInfo \*bi**

A pointer to the BitmapInfo. This is used to retrieve the file or device name (using **bi->Name()** or **bi->Device()**).

#### **TCHAR \*string = NULL**

The error message to present in the dialog. If NULL this method will query the operating system for the last I/O error and give its own interpretation of the error. This will work for all "File Not Found", "Permission Denied", etc. type errors.

### Return Value:

One of the following values:

#### **BMMRES\_ERRORTAKENCARE**

Returned if the user selected OK from the dialog or the error was taken care of (silent mode was on).

#### **BMMRES\_ERRORRETRY**

The user has selected Retry from the dialog.

### Prototype:

## **BMMRES ProcessImageIOError(BitmapInfo \*bi, int errorcode);**

### **Remarks:**

Implemented by the System.

This method may be called to present the user with the 3ds max Image IO Error dialog box displaying the specified error message based on the error code passed. The user may choose Cancel or Retry from the dialog. An appropriate value is returned based on the users selection. If Silent Mode is on (for example network rendering is being done) no dialog is presented and **BMMRES\_ERRORTAKENCARE** is returned.

### **Parameters:**

#### **BitmapInfo \*bi**

A pointer to the BitmapInfo. This is used to retrieve the file or device name (using **bi->Name()** or **bi->Device()**).

#### **int errorcode**

The error code. Pass one of the following values and the string shown to its right will be presented.

**BMMRES\_INTERNALERROR** - Internal Error

**BMMRES\_NUMBEREDFILENAMEERROR** - Error Creating  
Numbered File Name.

**BMMRES\_BADFILEHEADER** - Invalid Image File Header

**BMMRES\_CANTSTORAGE** - Error Creating Image Storage

**BMMRES\_MEMORYERROR** - Memory Error

**BMMRES\_BADFRAME** - Invalid Frame Requested

Any other values produce - Unknown Error

### **Return Value:**

One of the following values:

**BMMRES\_ERRORTAKENCARE**

Returned if the user selected Cancel from the dialog or the error was taken care of (silent mode was on).

**BMMRES\_ERRORRETRY**

The user has selected Retry from the dialog.

### **Sample Code:**

```
BMMRES BitmapIO_JPEG::Write(int frame) {  
    //-- If we haven't gone through an OpenOutput(), leave
```

```

if (openMode != BMM_OPEN_W)
    return (ProcessImageIOError(&bi,BMMRES_INTERNALERROR));
/-- Resolve Filename -----
TCHAR filename[MAX_PATH];
if (frame == BMM_SINGLEFRAME) {
    _tcscpy(filename,bi.Name());
} else {
    if (!BMMCreateNumberedFilename(bi.Name(),frame,filename)) {
        return
(ProcessImageIOError(&bi,BMMRES_NUMBEREDFILENAMEERRO
    }
}

```

## G-Buffer Channels Required

### Prototype:

```
virtual DWORD ChannelsRequired()
```

### Remarks:

Implemented by the Plug-In.

These are the channels required for output. By setting this flag, the plug-in can request that 3ds max generate the given channels. Prior to rendering, 3ds max will scan the plug-ins in the chain of events and list all types of channels being requested. The plug-in, at the time of the **Write()** method, will have access to these channels through the channel interface described in **BitmapStorage**.

### Return Value:

See [List of Image Channels](#).

### Default Implementation:

```
{ return BMM_CHAN_NONE; }
```

## Image Information

### Prototype:

```
virtual BMMRES GetImageInfoDlg( HWND hWnd, BitmapInfo
*bi, const TCHAR *filename = NULL )
```

**Remarks:**

Implemented by the Plug-In.

This method will display a dialog with information about the given bitmap (either defined in **bi.Name()/bi.Device()** or explicitly in the filename passed). The default method will retrieve image information using the mandatory **GetImageInfo()** and display a generic information dialog. If an image loader / writer wants to show its own info dialog, perhaps showing an image property not found in the generic dialog, it can implement its own function (and notify the system using the **BMM\_INFODLG** flag in the capabilities flag).

**Parameters:**

**HWND hWnd**

The parent window handle calling the dialog.

**BitmapInfo \*bi**

Defines the name of the bitmap or device (unless specified below).

**const TCHAR \*filename = NULL**

Specifies the filename to use explicitly.

**Return Value:**

The result of the operation. See [Bitmap Error Codes](#).

**Prototype:**

**virtual BMMRES GetImageInfo( BitmapInfo \*bi ) = 0;**

**Remarks:**

Implemented by the Plug-In.

The BitmapIO module implements this method to initialize the **BitmapInfo** instance passed in with information about the image. This information might be obtained from read the image header for example. The BitmapInfo passed contains the name of the image to get the information about.

**Parameters:**

**BitmapInfo \*bi**

A pointer to an instance of the class **BitmapInfo**.

**Return Value:**

If an error occurs, the plug-in should process the error (display a message if

appropriate) and return **BMMRES\_ERRORTAKENCARE**. If everything went OK, the plug-in should return **BMMRES\_SUCCESS**.

**Sample Code:**

```
bi->SetWidth(640);  
bi->SetHeight(480);  
bi->SetType(BMM_TRUE_24);  
bi->SetAspect(1.0f);  
bi->SetGamma(1.0f);  
bi->SetFirstFrame(0);  
bi->SetLastFrame(0);  
return BMM_SUCCESS;
```

**Prototype:**

```
virtual BMMRES GetImageName( BitmapInfo *bi, TCHAR  
*filename)
```

**Remarks:**

Implemented by the Plug-In.

This method is implemented by image file loaders (IFL handlers). It is called to update the filename passed based on the properties of the BitmapInfo passed. See the implementation of this method in **\MAXSDK\SAMPLES\IO\IFL.CPP**.

**Parameters:**

**BitmapInfo \*bi**

Specifies the properties of the IFL sequence.

**TCHAR \*filename**

The filename to update based on the properties of **bi**.

**Return Value:**

See [Bitmap Error Codes](#).

**Prototype:**

```
virtual void EvalMatch(TCHAR *matchString);
```

**Remarks:**

This method is available in release 2.0 and later only.

The bitmap manager caches images in order to speed its process. When a new image is requested, if it's

already loaded, a pointer to it is passed around as opposed to loading an entire new copy of it. It does so by comparing the image name and the frame number requested (in case of multiframe files and/or devices).

This works fine in most cases, however consider the following scenario. The Accom device generates images just as if you were loading Targa files. The comparison explained above works fine. The problem however, is that within the Accom private setup, you can determine where in the Accom to start reading frames. In other words, you may have a sequence of images recorded in the Accom starting at frame 300 for instance. Once in the Accom setup, you define your starting frame at 300 and whenever 3ds max requests a frame, the Accom driver offsets the requested frame number by 300. For example, when 3ds max is rendering its 10th frame and requests a frame from the Accom (for a Map or for a background, etc.), the Accom will see that 3ds max wants frame 10 and the user had setup the starting frame at 300. After computing the offset the Accom driver returns frame 310. Now, if two or more maps are used in a scene, the cache match mechanism explained above fails as it does not take in consideration the "starting frame" parameter of the Accom driver. For it, the name matches (Accom) and the frame number matches (frame 10 in the example above). If two or more maps start at different positions within the Accom, the first one defined will be used as it satisfies the match.

To handle this condition this new method saves extra information (driver private information) about a frame. Basically, part of the match process is now handled by those drivers that implement this method.

If a driver has some private data (handled by its own setup dialogue) that defines anything in how images are returned (in other words, if different images are returned because of those private settings being different), it must define the **BMMIO\_EVALMATCH** return flag and must also implement this method. The flag is necessary to avoid wasting time setting up instances of the driver just to call the (possibly unimplemented) method.

#### **Parameters:**

**TCHAR \*matchString**

The match string. The driver simply builds a string made of its own "per frame" parameters and returns it so 3ds max can compare it with another

instance of the driver.

## Loading

### Prototype:

```
virtual BitmapStorage *Load( BitmapInfo *bi, Bitmap *map,  
    BMMRES *status ) = 0;
```

### Remarks:

Implemented by the Plug-In.

This is the method that is called to actually load and store the image. This method usually creates the storage for the image data, allocates storage space, and puts the image to the storage one scanline at a time. This method also usually sets the **BitmapIO::openMode** flag to **BMM\_OPEN\_R** to indicate the image is loaded and open for reading.

### Parameters:

#### **BitmapInfo \*bi**

Points to an instance of class BitmapInfo. This has the name of the bitmap / device to load.

#### **Bitmap \*map**

This points to the bitmap to be loaded.

#### **BMMRES \*status**

The result of the bitmap load operation. See [Bitmap Error Codes](#).

### Return Value:

The BitmapStorage created to manages this bitmap.

### Sample Code:

See the **Load()** method of  
**\MAXSDK\SAMPLES\IO\JPEG\JPEG.CPP**.

## Output / Write / Close

### Prototype:

```
virtual int OpenOutput(BitmapInfo *bi, Bitmap *map);
```

### Remarks:

Implemented by the Plug-In.

This method opens the image for output and prepare to write to it. This is the time that the plug-in receives the information about what to write, the flags, etc. When the **Write()** method is called the only thing passed is the frame number.

**Parameters:**

**BitmapInfo \*bi**

The image information.

**Bitmap \*map**

Points to the bitmap to save.

**Return Value:**

Returns nonzero if everything was OK; otherwise 0.

**Sample Code:**

See the **OpenOutput()** method of

**\MAXSDK\SAMPLES\IO\JPEG\JPEG.CPP.**

**Prototype:**

**virtual int Write(int frame);**

**Remarks:**

Implemented by the Plug-In.

This method is called to write the image from the **BitmapStorage** to disk.

The data member **bi** contains the relevant information (for example **bi.Name()** contains the name of the file to write).

**Parameters:**

**DWORD frame**

Specifies the frame to write. For single image formats this will be **BMM\_SINGLEFRAME.**

**Return Value:**

Returns nonzero if everything was OK; otherwise 0.

**Sample Code:**

See the **Write()** method of

**\MAXSDK\SAMPLES\IO\JPEG\JPEG.CPP.**

**Prototype:**

```
virtual int Close(int flag);
```

**Remarks:**

Implemented by the Plug-In.

Closes the output file and saves or discards the changes based on the flag passed.

**Parameters:**

```
int flag
```

See [List of Bitmap Close Types](#).

**Return Value:**

Returns nonzero if output was closed successfully; otherwise 0.

**Sample Code:**

See the **Close()** method of

```
\MAXSDK\SAMPLES\IO\WSD\WSD.CPP.
```

**Internal use****Prototype:**

```
void InitOutputGammaTable(BitmapInfo*bi);
```

**Remarks:**

Implemented by the System.

This method is used internally to build the output gamma table.

**Prototype:**

```
virtual PAVIFILE GetPaviFile();
```

**Remarks:**

Implemented by the System.

This method is used internally.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

**inline void SetPrev(BitmapIO \*prev)**

**Remarks:**

Implemented by the System.

This method is used internally.

**Prototype:**

**inline void SetNext(BitmapIO \*next);**

**Remarks:**

Implemented by the System.

This method is used internally.

**Prototype:**

**inline BitmapIO \*Prev();**

**Remarks:**

Implemented by the System.

This method is used internally.

**Prototype:**

**inline BitmapIO \*Next();**

**Remarks:**

Implemented by the System.

This method is used internally.

## Class UtilityObj

See Also: [Class Interface](#), [Class IUtil](#).

class UtilityObj

### Description:

3ds max utility plug-ins are derived from this class. Methods are provided for editing the utilities parameters and responding to changes in the current selection set. An interface pointer is provided for calling the utility methods provided by MAX.

Note: Utility plug-ins are not a direct participant in the geometry pipeline system of 3ds max in the same way modifiers or space warps are. For this reason,

**UtilityObj** plug-ins are not suitable for modifying objects flowing down the pipeline. Use **Modifier** or **WSModifier** plug-ins for this purpose.

Also note: It is possible to create a utility plug-in that uses a modeless dialog box. When 3ds max itself uses modeless dialogs, it disables input to the other open windows such as the Track View, the Materials Editor, etc. In this way, the user cannot perform some action that could disturb the operation of the modeless dialog. For example using Track View, a user could assign a different controller to a node, and a utility plug-in might be accessing keys from the node's previous controller. Since utility plug-ins cannot currently prevent the user from operating these other parts of MAX, developers need to be careful about the use of modeless dialogs.

### Methods:

### Prototype:

```
virtual void BeginEditParams(Interface *ip,IUtil *iu)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called when the utility plug-in may be used in the Utility branch of the command panel. The plug-in may add rollup pages to the command panel in this method for example.

### Parameters:

#### **Interface \*ip**

An interface pointer you may use to call methods of the Interface class.

**IUtil \*iu**

An interface pointer you may use to close the current utility in the command panel.

**Prototype:**

**virtual void EndEditParams(Interface \*ip,IUtil \*iu)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called when the utility plug-in is done being used in the Utility branch of the command panel (for example if the user changes to the Create branch).

**Parameters:****Interface \*ip**

An interface pointer you may use to call methods of the Interface class.

**IUtil \*iu**

An interface pointer you may use to close the current utility in the command panel.

**Prototype:**

**virtual void SelectionSetChanged(Interface \*ip,IUtil \*iu);**

**Remarks:**

Implemented by the Plug-In.

This method is called when the selection set changes. A plug-in may implement this method to respond to this condition.

**Parameters:****Interface \*ip**

An interface pointer you may use to call methods of the Interface class.

**IUtil \*iu**

An interface pointer you may use to close the current utility in the command panel.

**Prototype:**

**virtual void SetStartupParam(TSTR param);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called after **BeginEditParams()** when the user starts the utility from the command line with the option **-U** and passes an argument to the utility.

**Parameters:****TSTR param**

The command line argument is passed here.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void DeleteThis()=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called to delete the utility object allocated by **ClassDesc::Create()**.

For example if the developer has used the **new** operator to allocate memory for their plug-in class they should implement this method as **{ delete this; }** to delete the plug-in object.

In some cases it may be better to use a single static instance of the plug-in class and not allocate and deallocate memory. For example some of the sample utility plug-ins use a single static instance of their plug-in class. This is done so that if the user moves between branches in the command panel (goes into the Create branch and then returns to the Utility branch) all the utility plug-in parameters remain intact. If the memory was allocated and deallocated each time the parameter would be 'forgotten'. The samples that use a single static instance implement this method as **{ }** (NULL). See the sample code in **\MAXSDK\SAMPLES\UTILITIES\COLCLIP.CPP** for an example.

## Class GUP

See Also: [Class Interface](#), [Class DllDir](#), [Class BitmapManager](#), [Class ITreeEnumProc](#), [Class ILoad](#), [Class ISave](#).

```
class GUP : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This is the base class for the creation of Global Utility Plug-Ins.

These plug-ins work as follows: At 3ds max startup time, before 3ds max begins its message loop and after all its subsystems are initialized, the GUP Manager will scan the plug-in directory and load all plug-ins with the **"\*.gup"** extension. One by one, the GUP Manager will call the plug-in's **Start()** method. At that point the plug-in would do its initialization and decide if it wants to remain loaded or if it can be discarded. As an option, it is also possible for a GUP plug-in for make 3ds max fail and abort all together. If a plug-in wishes to remain loaded (after returning a **GUPRESULT\_KEEP** result code from the **Start()** method described below), it should start a new thread for its code as there is no other call from 3ds max.

Unlike other 3ds max plug-ins, GUP's do not provide a user interface. If developers of GUP plug-ins desire to present an interface, they can develop a standard 3ds max utility plug-in to do so. See [Class UtilityObj](#). There is some sample code using this technique available in **\MAXSDK\SAMPLES\GUP\COMSRV\MSCOM.CPP**. This Utility plug-in (**COMSRV.DLU**) accesses the COM/DCOM plug-in and allows the user to "register" or "unregister" the COM interface. See the Advanced Topics section [COM/DCOM Interface](#).

### Plug-In Information:

Class Defined In GUP.H

Super Class ID GUP\_CLASS\_ID

Standard File Name Extension GUP

Extra Include File Needed None

**Methods:**

public:

**Prototype:**

**GUP();**

**Remarks:**

Constructor.

**Prototype:**

**virtual ~GUP();**

**Remarks:**

Destructor.

**Prototype:**

**virtual HINSTANCE MaxInst();**

**Remarks:**

Implemented by the System.

Returns the application instance handle of 3ds max itself.

**Prototype:**

**virtual HWND MaxWnd();**

**Remarks:**

Implemented by the System.

Returns the window handle of 3ds max's main window.

**Prototype:**

**virtual DllDir\* MaxDllDir();**

**Remarks:**

Implemented by the System.

Returns a pointer to an instance of a class which provides access to the DLL Directory. This is a list of every DLL loaded in 3ds max

**Prototype:**

**virtual Interface\* Max();**

**Remarks:**

Implemented by the System.

Returns an interface pointer for calling methods provided by 3ds max.

**Prototype:**

**virtual BitmapManager\* Bmi();**

**Remarks:**

Implemented by the System.

Returns a pointer to the bitmap manager which may be used to manage the use of bitmaps within 3ds max.

**Prototype:**

**virtual int EnumTree(ITreeEnumProc \*proc);**

**Remarks:**

Implemented by the System.

This may be called to enumerate every INode in the scene.

**Parameters:**

**ITreeEnumProc \*proc**

This callback object is called once for each INode in the scene.

**Return Value:**

Nonzero if the process was aborted by the callback (**TREE\_ABORT**); otherwise 0.

**Prototype:**

**virtual bool ExecuteStringScript(TCHAR \*string);**

**Remarks:**

Implemented by the System.

This method will execute the specified MAXScript command. If a developer needs to ask 3ds max to do something and this "something" is not implemented within the COM interface, it is possible to send MAXScript

commands through this method (and **ExecuteFileScript()** below). This method will execute whatever string is specified, for instance **ExecuteStringScript("open \"MyScene.max\")**.

**Parameters:**

**TCHAR \*string**

Points to the MAXScript command to execute.

**Return Value:**

TRUE indicates if the command was successfully sent to MAXScript; FALSE if it was not sent. Note that this does not reflect the success of the embedded command.

**Prototype:**

**virtual bool ExecuteFileScript(TCHAR \*file);**

**Remarks:**

Implemented by the System.

This method will execute the specified MAXScript file.

**Parameters:**

**TCHAR \*file**

The file name for the script file.

**Return Value:**

TRUE indicates if the script was successfully sent to MAXScript; FALSE if it was not sent. Note that this does not reflect the result of the script.

**Prototype:**

**virtual DWORD Start()=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called at boot time. At that point the plug-in should do its initialization and decide if it wants to remain loaded or if it can be discarded. As an option, it is also possible for a GUP plug-in for make 3ds max fail and abort all together. Obviously this should be used with caution and plenty of documentation from the part of the developer of the plug-in. If a plug-in wishes to remain loaded (after returning a **GUPRESULT\_KEEP** result

code), it should start a new thread for its code as there is no other call from 3ds max.

**Return Value:**

One of the following values:

**GUPRESULT\_KEEP**

Return this value to have the plug-in remain loaded.

**GUPRESULT\_NOKEEP**

Return this value to discard.

**GUPRESULT\_ABORT**

Return this value to cause 3ds max to shut down.

**Prototype:**

**virtual void Stop()=0;**

**Remarks:**

Implemented by the Plug-In.

The **Stop()** method is called when 3ds max is going down. The GUP Manager will call this methods for all GUP plug-ins that were maintained in memory right before it discards them. This method is called only for plug-ins that returned **GUPRESULT\_KEEP** for the **Start()** method above.

**Prototype:**

**virtual DWORD Control(DWORD parameter);**

**Remarks:**

Implemented by the Plug-In.

This method is an entry point for external access to GUP plug-in. For instance, Utility plug-ins can invoke their GUP plugin counterpart and have direct access to them.

**Parameters:**

**DWORD parameter**

The meaning of this parameter is defined by the plug-in.

**Return Value:**

The return value meaning is also defined by the plug-in.

**Default Implementation:**

```
{ return 0;}
```

**Prototype:**

```
virtual IOResult Save(ISave *isave);
```

**Remarks:**

This method is called to save any data the plug-in may have into the 3ds max file.

**Parameters:**

**ISave \*isave**

An interface used for saving data.

**Prototype:**

```
virtual IOResult Load(ILoad *iload);
```

**Remarks:**

This method is called to load any data the plug-in may have from the 3ds max file.

**Parameters:**

**ILoad \*iload**

An interface used for loading data.

# Class TrackViewUtility

See Also: [Class ITVUtility](#), [Class Interface](#), [Class Control](#).

class TrackViewUtility

## Description:

This class is available in release 2.0 and later only.

This is the base class for Track View Utility plug-ins. These plug-ins are launched via the 'Track View Utility' icon just to the left of the track view name field in the toolbar. Clicking on this button brings up a dialog of all the track view utilities currently installed in the system. Most utilities will probably be modeless floating dialogs, however modal utilities may be created as well.

The developer will derive their classes from this class. Methods are provided to bracket the beginning and ending of parameter editing, and responding to various changes in Track View (such as key selection, time selection, node selection, etc.). There is also a method to delete this instance of the plug-in class.

Sample code is available in

**\MAXSDK\SAMPLES\UTILITIES\RANDKEYS.CPP, ORTKEYS.CPP and SELKEYS.CPP.**

## Plug-In Information:

Class Defined In	TVUTIL.H
Super Class ID	TRACKVIEW_UTILITY_CLASS_ID
Standard File Name Extension	DLU
Extra Include File Needed	None

## Methods:

### Prototype:

**virtual void DeleteThis()=0;**

### Remarks:

This method is called to delete this instance of the plug-in class. This method should free the memory allocated in **ClassDesc::Create()**. See the Advanced Topics section on [Memory Management](#) for more details.

### Prototype:

**virtual void BeginEditParams(Interface \*ip,ITVUtility \*iu);**

**Remarks:**

This method is called to begin editing of the Track View utility plug-in's parameters.

**Parameters:**

**Interface \*ip**

An interface for calling functions provided by 3ds max.

**ITVUtility \*iu**

An interface for allowing track view utilities to access the Track View they are launched from.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void EndEditParams(Interface \*ip,ITVUtility \*iu);**

**Remarks:**

This method is called when the user has closed the Track View utility or Track View itself.

**Parameters:**

**Interface \*ip**

An interface for calling functions provided by 3ds max.

**ITVUtility \*iu**

An interface for allowing track view utilities to access the Track View they are launched from.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void TrackSelectionChanged();**

**Remarks:**

This method is called when the selection of tracks has changed.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void NodeSelectionChanged();
```

**Remarks:**

This method is called when the selection of nodes has changed.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void KeySelectionChanged();
```

**Remarks:**

This method is called when the selection of keys has changed.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void TimeSelectionChanged();
```

**Remarks:**

This method is called when the amount of time selected changes in Edit Time mode. See **ITVUtility::GetTimeSelection()**.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void MajorModeChanged();
```

**Remarks:**

This method is called if the current mode of Track View changes. These are the modes such as Edit Keys, Edit Time, Edit Ranges, Position Ranges, and Edit Function Curves. See **ITVUtility::GetMajorMode()**.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual void TrackListChanged();
```

**Remarks:**

This method is called when the Track View list is rebuild. This is the list of items that are visible (currently open).

**Default Implementation:**

```
{
```

# Class Renderer

See Also: [Class ReferenceTarget](#), [Class FrameRenderParams](#), [Class RendProgressCallback](#), [Class IRendParams](#), [Class INode](#), [Class ViewParams](#), [Class RendParamDlg](#), [Class RendParams](#), [Class DefaultLight](#).

class Renderer : public ReferenceTarget

## Description:

This is the base class for the creation of plug-in renderers. There are five methods that need to be implemented: **Open()**, **Render()**, **Close()**, **CreateParamDialog()** and **ResetParams()**.

In 3ds max 2.0 and later developers must also implement **ReferenceTarget::Clone()** to support the new Production/Draft renderer capability.

## Methods:

### Prototype:

```
virtual int Open(INode *scene, INode *vnode, ViewParams  
*viewPar, RendParams &rp, HWND hwnd, DefaultLight*  
defaultLights=NULL, int numDefLights=0)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called once per render. It gives the renderer a chance to build up any data structures which it will use over the course of the render.

### Parameters:

#### **INode \*scene**

The root node of the scene to render. Note: If you are rendering in the Materials Editor, you'll instead get a pointer to the INode that is in the sample slot -- not the root node.

#### **INode \*vnode**

The view node. This may be a camera, a light, or NULL.

#### **ViewParams \*viewPar**

View parameters for rendering orthographic or user viewports. This is used if **\*vnode** is NULL.

## **RendParams &rp**

This class contains a set of common renderer parameters.

### **HWND hwnd**

The owner window for messages.

### **DefaultLight\* defaultLights=NULL**

An array of default lights if there are no user created lights in the scene.

### **int numDefLights=0**

Number of lights in **defaultLights** array.

### **Return Value:**

Nonzero for success, zero for failure.

### **Prototype:**

```
virtual int Render(TimeValue t, Bitmap *tobm,  
FrameRendParams &frp, HWND hwnd, RendProgressCallback  
*prog=NULL, ViewParams *viewPar=NULL)=0;
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to render a frame. It should use the camera or view passed to the **Open()** method. This is called once per frame, and the resulting rendered image is written to **tobm**.

### **Parameters:**

#### **TimeValue t**

The frame to render.

#### **Bitmap \*tobm**

This is the target bitmap. The properties of the bitmap define the render width, height and aspect. See [Class Bitmap](#).

#### **FrameRendParams &frp**

A set of time dependent parameters.

#### **HWND hwnd;**

The owner window handle.

#### **RendProgressCallback \*prog=NULL**

A callback used to allow the renderer to update the rendering dialog. The

renderer may call methods of this class.

**ViewParams \*viewPar=NULL**

This parameter is available in release 2.0 and later only.

This parameter allows you to specify a different view transformation on each render call. For example, you can render a given scene at a given time from many different viewpoints, without calling **Render::Open()** for each one.

**Return Value:**

Nonzero for success, zero for failure.

**Prototype:**

**virtual void Close (HWND hwnd)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called once when rendering is done. This is where the renderer frees any data structures being held on to. This includes things such as shadow buffers, render-ready meshes, lists of materials, etc.

**Parameters:**

**HWND hwnd**

The owner window handle.

**Prototype:**

**virtual bool ApplyRenderEffects(TimeValue t, Bitmap \*pBitmap, bool updateDisplay=true);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is called to apply the render effects at the specified time value. It should be called between the **Open()** and **Close()** methods.

This can be used during a multi-pass rendering, in order to apply the render effects to the final, blended bitmap.

**Parameters:**

**TimeValue t**

The time to apply the render effects.

**Bitmap \*pBitmap**

Points to the bitmap.

**bool updateDisplay=true**

Passing true indicates that Bitmap's display should be refreshed by the renderer; false indicates it should not be.

**Return Value:**

Returns true if the effects were successfully applied; otherwise false.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual RendParamDlg *CreateParamDialog(IRendParams  
*ir,BOOL prog=FALSE)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called to create and return a pointer to an instance of the **RendParamDlg** class. The renderer can add rollup page(s) to the renderer configuration dialog using the **IRendParams** interface passed into this method.

**Parameters:****IRendParams \*ir**

An interface that provides methods for use in displaying parameters, for example this class has methods for adding rollup pages.

**BOOL prog=FALSE**

If TRUE then the rollup page should just display the parameters so the user has them for reference while rendering, they should not be editable.

**Return Value:**

A pointer to an instance of the **RendParamDlg** class. This class will be deleted using **RendParamDlg::DeleteThis()**.

**Prototype:**

```
virtual void ResetParams()=0;
```

**Remarks:**

Implemented by the Plug-In.

This method simply sets all the parameters to their default values.

# Class Shader

See Also: [Class BaseShader](#), [Class MacroRecorder](#).

```
class Shader : public BaseShader
```

## Description:

This class is available in release 3.0 and later only.

This is the class that developers use to create Shader plug-ins. Developers must implement the methods of this class to provide data to the 3ds max interactive renderer so it can properly reflect the look of the shader in the viewports. The methods associated with the actual Shader illumination code are from the base class **BaseShader**.

There are various Get and Set methods defined in this class. Plug-in developers provide implementations for the 'Get' methods which are used by the interactive renderer. The implementations of the 'Set' methods are used when switching between shaders types in the Materials Editor. This is used to transfer the corresponding colors between the old Shader and the new one.

Note that some shaders may not have the exact parameters as called for in the methods. In those case an approximate value may be returned from the 'Get' methods. For example, the Strauss Shader doesn't have an Ambient channel. In that case the Diffuse color is taken and divided by 2 and returned as the Ambient color. This gives the interactive renderer something to work with that might not be exact but is somewhat representative.

## Methods:

```
public:
```

## Prototype:

```
virtual void CopyStdParams(Shader* pFrom)=0;
```

## Remarks:

This method copies the standard shader parameters from **pFrom** to this object. Note that plug-ins typically disable the macro recorder during this operation as the Get and Set methods are called. See the sample code for examples.

## Parameters:

```
Shader* pFrom
```

The source parameters.

**Prototype:**

**virtual BOOL GetLockDS()=0;**

**Remarks:**

Returns TRUE if the Diffuse / Specular lock is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetLockAD()=0;**

**Remarks:**

Returns TRUE if the Ambient / Diffuse lock is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetLockADTex()=0;**

**Remarks:**

Returns TRUE if the Ambient / Diffuse Texture lock is on; otherwise FALSE.

**Prototype:**

**virtual Color GetAmbientClr(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Ambient Color.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual Color GetDiffuseClr(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Diffuse Color.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual Color GetSpecularClr(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Specular Color.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual Color GetSelfIllumClr(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Self Illumination Color.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual float GetSelfIllum(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Self Illumination Amount.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual float GetGlossiness(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Glossiness Level.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual float GetSpecularLevel(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Specular Level.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual float GetSoftenLevel(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Soften Level as a float.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual BOOL IsSelfIllumClrOn(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns TRUE if the Self Illumination Color setting is on (checked); FALSE if off.

**Parameters:**

The parameters to this method are not applicable and may safely be ignored.

**Prototype:**

**virtual BOOL IsSelfIllumClrOn()=0;**

**Remarks:**

Returns TRUE if the Self Illumination Color setting is on (checked); FALSE if off.

**Prototype:**

**virtual Color GetAmbientClr(TimeValue t)=0;**

**Remarks:**

Returns the Ambient Color at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the color.

**Prototype:**

**virtual Color GetDiffuseClr(TimeValue t)=0;**

**Remarks:**

Returns the Diffuse Color at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the color.

**Prototype:**

**virtual Color GetSpecularClr(TimeValue t)=0;**

**Remarks:**

Returns the Specular Color at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the color.

**Prototype:**

**virtual float GetGlossiness(TimeValue t)=0;**

**Remarks:**

Returns the Glossiness value at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the value.

**Prototype:**

**virtual float GetSpecularLevel(TimeValue t)=0;**

**Remarks:**

Returns the Specular Level at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the value.

**Prototype:**

**virtual float GetSoftenLevel(TimeValue t)=0;**

**Remarks:**

Returns the Soften Specular Highlights setting at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the value.

**Prototype:**

**virtual float GetSelfIllum(TimeValue t)=0;**

**Remarks:**

Returns the Self Illumination Amount at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the value.

**Prototype:**

**virtual Color GetSelfIllumClr(TimeValue t)=0;**

**Remarks:**

Returns the Self Illumination Color at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the color.

**Prototype:**

**virtual void SetLockDS(BOOL lock)=0;**

**Remarks:**

Sets the state of the Diffuse / Specular lock to on or off.

**Parameters:**

**BOOL lock**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetLockAD(BOOL lock)=0;**

**Remarks:**

Sets the state of the Ambient / Diffuse lock to on or off.

**Parameters:**

**BOOL lock**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetLockADTex(BOOL lock)=0;**

**Remarks:**

Sets the state of the Ambient / Diffuse Texture lock to on or off.

**Parameters:**

**BOOL lock**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetSelfIllum(float v, TimeValue t)=0;**

**Remarks:**

Sets the Self Illumination parameter to the specified value at the time passed.

**Parameters:**

**float v**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetSelfIllumClrOn(BOOL on)=0;**

**Remarks:**

Sets the Self Illumination Color On/Off state.

**Parameters:**

**BOOL on**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetSelfIllumClr(Color c, TimeValue t)=0;**

**Remarks:**

Sets the Self Illumination Color at the specified time.

**Parameters:****Color c**

The color to set.

**TimeValue t**

The time to set the color.

**Prototype:**

**virtual void SetAmbientClr(Color c, TimeValue t)=0;**

**Remarks:**

Sets the Ambient Color at the specified time.

**Parameters:****Color c**

The color to set.

**TimeValue t**

The time to set the color.

**Prototype:**

**virtual void SetDiffuseClr(Color c, TimeValue t)=0;**

**Remarks:**

Sets the Diffuse Color at the specified time.

**Parameters:****Color c**

The color to set.

**TimeValue t**

The time to set the color.

**Prototype:**

**virtual void SetSpecularClr(Color c, TimeValue t)=0;**

**Remarks:**

Sets the Specular Color at the specified time.

**Parameters:**

**Color c**

The color to set.

**TimeValue t**

The time to set the color.

**Prototype:**

**virtual void SetGlossiness(float v, TimeValue t)=0;**

**Remarks:**

Sets the Glossiness parameter to the specified value at the time passed.

**Parameters:**

**float v**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetSpecularLevel(float v, TimeValue t)=0;**

**Remarks:**

Sets the Specular Level parameter to the specified value at the time passed.

**Parameters:**

**float v**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetSoftenLevel(float v, TimeValue t)=0;**

**Remarks:**

Sets the Soften Specular Highlights Level to the specified value at the time passed.

**Parameters:**

**float v**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual float EvalHiliteCurve(float x);**

**Remarks:**

This method is called to evaluate the highlight curve that appears in the Shader user interface.

Note: This gets called from the **DrawHilite()** function which is available to developers in

**\MAXSDK\SAMPLES\MATERIALS\SHADER\SHADERUTIL.CP**

**DrawHilite()** get called from the window proc **HiliteWndProc()** in the same file. This code is available to developers to use in their Shader dialog procs.

**Parameters:**

**float x**

The input value.

**Return Value:**

The output value on the curve. A value of 1.0 represents the top of the curve as it appears in the UI. Values greater than 1.0 are okay and simply appear off the top of the graph.

**Default Implementation:**

```
{ return 0.0f; }
```

**Prototype:**

**virtual float EvalHiliteCurve2(float x, float y, int level = 0);**

**Remarks:**

This is the highlight curve function for the two highlight curves which intersect and appear in the UI, for instance in the Anisotropic shader.

**Parameters:****float x**

The x input value.

**float y**

The y input value.

**int level = 0**

This is used by multi-layer shaders to indicate which layer to draw. The draw highlight curve routines use this when redrawing the graph.

**Return Value:**

The output value of the curve.

**Default Implementation:**

```
{ return 0.0f; }
```

## Class Sampler

See Also: [Class SpecialFX](#), [Class SamplingCallback](#), [Class SFXParamDlg](#), [Class ShadeContext](#), [Class Point3](#), [Class Point2](#), [Class ILoad](#), [Class ISave](#).

```
class Sampler : public SpecialFX
```

### Description:

This class is available in release 3.0 and later only.

This is the base class for the creation of Sampler plug-ins which work with the Standard material. These appear in the Super Sampling rollout in the Sampler dropdown. They have an Enable checkbox and a Quality spinner for their user interface. An optional modal dialog may also be presented.

A Sampler is a plug-in that determines where inside a single pixel the shading and texture samples are computed. For some Samplers this pattern is the same for each pixel, for others a different pattern is chosen for each pixel. After determining the sample locations, the sampler calls back to the renderer to compute the shading values. It then averages the resulting shading values and returns its estimate of the final color.

Some Samplers are adaptive. This means that the Sampler decides on-the-fly how many samples to take to achieve its goal. There are many subtleties to adaptive Samplers and many ways to define the adaptive mechanism. The adaptive mechanism used by the R3 Samplers is very simple: take 4 samples, look for the maximum change in any of the color channels, if it's greater than the threshold, then sample the entire pixel according to the given quality. Threshold is an optional parameter that may, but need not be used by adaptive Samplers.

The transfer of control from 3ds max to the Sampler plug-in is as follows: A Sampler is responsible for the sampling loop. It samples until it is done and computes the sum of its samples upon completion. Once the Sampler's **DoSample()** method is called 3ds max no longer has control. This is how adaptive samplers are handled. The **DoSample()** routine will determine how often and where it samples, then it calls the provided **SamplingCallback::SampleAtOffset()** method to have 3ds max compute the shading value.

### Plug-In Information:

Class Defined In SAMPLER.H

Super Class ID SAMPLER\_CLASS\_ID

Standard File Name Extension DLH

Extra Include File Needed None

### Methods:

public:

### Prototype:

```
virtual void DoSamples(Color& c, Color& t, SamplingCallback*  
cb, ShadeContext* sc, MASK mask=NULL)=0;
```

### Remarks:

This is the method where the Sampler plug-in does its sampling loop. Upon completion it returns the color and transparency back to 3ds max in **c** and **t**.

A sampler samples a range of 0.0 to 1.0. For a pixel sampler this range gets mapped to a single pixel. The sampler doesn't need to be concerned with this however. It just works within the 0.0 to 1.0 space determining where to put the samples. Essentially, this method generates a set of points and calls **SamplingCallback::SampleAtOffset()** for each one. Then it sums up the results of the values returned from **SampleAtOffset()**, divides by the number of samples, and stores the results in **c** and **t**.

### Parameters:

#### **Color& c**

This is the output color.

#### **Color& t**

This is the output transparency.

#### **SamplingCallback\* cb**

This is the callback provided by 3ds max which the sampler uses to actually do the sampling.

#### **ShadeContext\* sc**

The Shade Context which provides information about the pixel being sampled.

#### **MASK mask=NULL**

The 64 bit pixel mask. This mask corresponds to the 8x8 sub-pixel area grid. The actual geometry covers only some portion of these bits. This is essentially an 8x8 raster for the inside of the pixel where bits are set over the polygon

being rendered and bits are off for areas not over the polygon. Developers typically only want to sample where the geometry is and thus when the bits are on. If not the results are very poor visually.

Note: Most polygons are quite small in a typically complex scene being rendered. In other words, most polygons that need to get sampled will only have a small number of these mask bits on since the polygons are very small relative to the pixel. For instance, edge on polygons may project down to only a few bits within the pixel. Consequently it is quite possible that there may be zero samples, i.e. no geometry in the mask. Developers need to check for this zero samples condition. If this is the case then a method of **ShadeContext** called **SurfacePtScreen()** is used. This method returns a point which is guaranteed to be on the fragment no matter how small it is. This point can then be used for at least a single sample.

### **Sample Code:**

The following is a brief analysis of the **DoSamples()** method from the Uniform Sampler of 3ds max. This sampler sub-divides the sample area into a grid and samples the pixel at the center point of each grid unit.

This code is from the file

**\\MAXSDK\\SAMPLES\\RENDER\\SAMPLERS\\STDSAMPLERS.CPI**

The complete code is shown below and then a code fragment analysis follows:

```
void UniformSampler::DoSamples(Color& clr, Color&trans,  
SamplingCallback* cb, ShadeContext* sc, MASK mask )
```

```
{
```

```
int sideSamples = GetSideSamples();
```

```
int numSamples = sideSamples * sideSamples;
```

```
DbgAssert( sideSamples > 0 );
```

```
// we map 0...sideSz into 0..1
```

```
float sideSzInv = 1.0f / float(sideSamples);
```

```
float sampleScale = sideSzInv;
```

```
Point2 sample;
```

```
float nSamples = 0.0f;
```

```
clr.r = clr.g = clr.b = trans.r = trans.g = trans.b = 0;
```

```

// Sampling loop
for( int y = 0; y < sideSamples; ++y ) {
    sample.y = (float(y) + 0.5f) * sideSzInv;

    for( int x = 0; x < sideSamples; ++x ) {
        sample.x = (float(x) + 0.5f) * sideSzInv;

        if ( sampleInMask( sample, mask ) ) {
            Color c, t;
            // NB, returns true for unclipped samples
            if (cb->SampleAtOffset( c, t, sample, sampleScale )) {
                clr += c;
                trans += t;
                nSamples += 1.0f;
            }
        }
    }
}

// Check for 0 samples
if ( nSamples == 0.0f ){
    // use frag center if no other samples
    sample = sc->SurfacePtScreen();
    sample.x = frac(sample.x ); sample.y = frac( sample.y );

    cb->SampleAtOffset( clr, trans, sample, 1.0f );
} else {
    clr /= nSamples;
    trans /= nSamples;
}
}

```

The above code is broken into smaller fragments to look at below:

```

int sideSamples = GetSideSamples();

```

Here the sampler is just getting the number of sides in the sampling grid. This is computed based on the Quality spinner in the user interface. In this sampler this results in a number between 2 and 6 (developers can look at the **UniformSampler::GetSideSamples()** method to see this). Thus the resulting sampling grid is 2x2 or 3x3, up to 6x6. Then the number of samples is computed by multiplying the number of sides times itself.

```
int numSamples = sideSamples * sideSamples;
```

Next the side size inverse is computed to know how big the step size is. This is the amount to step along each time.

The sample scale is how large is the piece that's being sampled. For example, if the grid is 2x2 then each sample is scaled by 1/2

```
float sideSzInv = 1.0f / float(sideSamples);
```

```
float sampleScale = sideSzInv;
```

Next the number of samples, and the color and transparency are initialized to zero:

```
Point2 sample;
```

```
float nSamples = 0.0f;
```

```
clr.r = clr.g = clr.b = trans.r = trans.g = trans.b = 0;
```

Then the sampling loop begins. Here the positions of individual sampling points are computed. Each point is then checked to see if it corresponds to a point in the mask (is over a polygon). (The **sampleInMask** function is defined in

```
\MAXSDK\SAMPLES\RENDER\SAMPLERS\SAMPLERUTIL.CI
```

If it is a point that's over a polygon then **SampleAtOffset()** is called. What **SampleAtOffset()** does is turn the passed 2D sample into a 3D sample and returns a color and transparency. These returned values are summed up over the sampling loop (**clr += c; trans += t;**).

```
// Sampling loop
```

```
for( int y = 0; y < sideSamples; ++y ) {  
sample.y = (float(y) + 0.5f) * sideSzInv;
```

```
for( int x = 0; x < sideSamples; ++x ) {  
sample.x = (float(x) + 0.5f) * sideSzInv;
```

```
if ( sampleInMask( sample, mask ) ) {
```

```

Color c, t;
// NB, returns true for unclipped samples
if (cb->SampleAtOffset( c, t, sample, sampleScale )) {
    clr += c;
    trans += t;
    nSamples += 1.0f;
}
}
}
}

```

At the end of the sampling loop a check is done to see if there were zero samples. This is the case if the geometry is very small relative to the pixel. There are two approaches that one might take when there are zero samples. One is to simply return black. A strict 'jitter-type' sampler might do this since, in fact, no samples were hit. This will result in artifacts to the image however. A better approach is to use the **ShadeContext** method **SurfacePtScreen()** to return a point which is guaranteed to be at the center of the fragment. Then this point is passed to **SampleAtOffset()** so a single sample which is on the fragment is used.

If a single sample point was used, **DoSamples()** is finished. The results are in **clr** and **trans** as returned from **SampleAtOffset()**.

If a number of samples was taken, the Colors **clr** and **trans** are divided by the number of samples (**nSamples**) to get the final colors.

```

// Check for 0 samples
if ( nSamples == 0.0f ){
    // use frag center if no other samples
    sample = sc->SurfacePtScreen();
    sample.x = frac(sample.x ); sample.y = frac( sample.y );

    cb->SampleAtOffset( clr, trans, sample, 1.0f );
} else {
    clr /= nSamples;
    trans /= nSamples;
}

```

**Prototype:****virtual int GetNSamples()=0;****Remarks:**

This methods returns the integer number of samples given the current quality setting. If doing adaptive sampling (where the number of samples may vary) return the maximum number of samples possible.

**Prototype:****virtual float GetQuality()=0;****Remarks:**

Returns the sampling quality in the range of 0.0 to 1.0. Quality means how many samples are taken to compute the shade in a pixel. Higher quality is of course achieved by more samples. Quality 0.0 means "minimal", Quality 1.0 means "best", and Quality 0.5 means "good, the default ". Some samplers do not have adjustable quality (like 3ds max 2.5 Star), in which case the quality spinner is disabled and this method is ignored.

**Prototype:****virtual void SetQuality(float value)=0;****Remarks:**

Sets the sampling quality. This is the one default parameter.

**Parameters:****float value**

Quality is nominal with a range of 0.0 to 1.0.

**Prototype:****virtual int SupportsQualityLevels()=0;****Remarks:**

This method returns 0 on "unchangeable", otherwise the number of quality levels.

**Prototype:****virtual BOOL GetEnable()=0;**

**Remarks:**

Returns TRUE if sampling is enabled; otherwise FALSE.

**Prototype:**

**virtual void SetEnable(BOOL samplingOn)=0;**

**Remarks:**

Sets the Enable Sampler state to on or off.

**Parameters:**

**BOOL samplingOn**

TRUE for on; FALSE for off.

**Prototype:**

**virtual TCHAR\* GetDefaultComment()=0;**

**Remarks:**

Returns a comment string for the Sampler which appears in the Materials Editor user interface.

**Prototype:**

**virtual SamplerParamDlg \*CreateParamDialog(HWND hWndParent);**

**Remarks:**

This method creates and puts up a pop-up modal dialog that allows editing the sampler extended parameters (if any).

**Parameters:**

**HWND hWndParent**

The window handle of the parent.

**Return Value:**

Points to the object to manage the dialog. Note: **typedef SFXParamDlg SamplerParamDlg;**

See [Class SFXParamDlg](#).

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual BOOL SetDlgThing(EffectParamDlg* dlg);
```

**Remarks:**

You should implement this method if you are using the ParamMap2 AUTO\_UI system and the sampler has secondary dialogs that have something other than the incoming effect as their 'thing'. Called once for each secondary dialog for you to install the correct thing. Return TRUE if you process the dialog, FALSE otherwise, in which case the incoming effect will be set into the dialog.

Note: Developers needing more information on this method can see the remarks for [MtlBase::CreateParamDlg\(\)](#) which describes a similar example of this method in use (in that case it's for use by a texture map plug-in).

**Parameters:**

```
EffectParamDlg* dlg
```

Points to the ParamDlg.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
IOResult Save(ISave *isave);
```

**Remarks:**

Implemented by the System.

This method saves the name of the sampler. This should be called at the start of a plug-in's save methods.

**Parameters:**

```
ISave *isave
```

An interface for saving data.

**Prototype:**

```
IOResult Load(ILoad *iload);
```

**Remarks:**

Implemented by the System.

This method loads the name of the sampler. This should be called at the start of a plug-in's load methods.

**Parameters:****ILoad \*iload**

An interface for loading data.

**Optional Adaptive Sampling Methods (with default implementations)****Prototype:**

```
virtual BOOL SupportsAdaptive();
```

**Remarks:**

Returns TRUE if the sampler is adaptive; otherwise FALSE. If this method returns TRUE the Adaptive On checkbox appears along with the Threshold spinner.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual ULONG SupportsStdParams();
```

**Remarks:**

This method determines which of the various optional parameters are displayed. Zero or more of the following values (which may be added together):

**IS\_ADAPTIVE** -- Samples is adaptive in some way.

**ADAPTIVE\_CHECK\_BOX** -- Enable the Adaptive check box.

**ADAPTIVE\_THRESHOLD** -- Enable the adaptive Threshold spinner.

**SUPER\_SAMPLE\_TEX\_CHECK\_BOX** -- Enable the texture Super Sampling check box.

**ADVANCED\_DLG\_BUTTON** -- Enable the Advanced button. This allows an additional popup dialog to be presented to the user. See the

method **ExecuteParamDialog()**.

**OPTIONAL\_PARAM\_0** -- Enable optional spinner 0. See the methods **GetOptionalParamName()**, **GetOptionalParamMax()**, etc.

**OPTIONAL\_PARAM\_1** -- Enable optional spinner 1.

The following option is simply a set of these:

```
R3_ADAPTIVE =  
(IS_ADAPTIVE+ADAPTIVE_CHECK_BOX+ADAPTIVE_THR
```

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual void SetTextureSuperSampleOn(BOOL on);
```

**Remarks:**

This method is called on the Sampler to reflect the change in the 'Supersamp. Tex.' checkbox state. This determines whether to cut down the texture sample size of each sample, or whether to always use 1 pixel texture sample size.

**Parameters:**

**BOOL on**

TRUE for on; FALSE for off.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BOOL GetTextureSuperSampleOn();
```

**Remarks:**

Returns TRUE if Super Sampling is on; otherwise FALSE. See **SetTextureSuperSampleOn()** above.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual void SetAdaptiveOn(BOOL on);**

**Remarks:**

This method is called on the Sampler to reflect the change in the 'Adaptive' checkbox state.

**Parameters:**

**BOOL on**

TRUE for on; FALSE for off.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual BOOL IsAdaptiveOn();**

**Remarks:**

Returns TRUE if Adaptive is on (checked in the user interface); otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual void SetAdaptiveThreshold(float value);**

**Remarks:**

This method is called on the Sampler to reflect the change in the 'Threshold' spinner.

**Parameters:**

**float value**

The value to set. Range 0-1.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual float GetAdaptiveThreshold();**

**Remarks:**

Returns the adaptive threshold setting.

**Default Implementation:**

```
{ return 0.0f; }
```

**Optional Parameter Related Methods****Prototype:**

```
virtual long GetNOOptionalParams();
```

**Remarks:**

Samplers plug-ins support two optional parameter which may be used by the plug-in for its own needs. This methods returns the number of parameters it supports. Note that the max value is 2.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual TCHAR *GetOptionalParamName(long nParam);
```

**Remarks:**

Returns the name of the specified parameter.

**Parameters:**

**long nParam**

The zero based index of the optional parameter: 0 for the first one, 1 for the second.

**Default Implementation:**

```
{ return _T(""); }
```

**Prototype:**

```
virtual float GetOptionalParamMax(long nParam);
```

**Remarks:**

Returns the maximum value of the specified optional parameter.

**Parameters:**

### **long nParam**

The zero based index of the optional parameter: 0 for the first one, 1 for the second.

#### **Default Implementation:**

```
{ return 1.0f; }
```

#### **Prototype:**

```
virtual float GetOptionalParam(long nParam);
```

#### **Remarks:**

Returns the value of the specified optional parameter.

#### **Parameters:**

### **long nParam**

The zero based index of the optional parameter: 0 for the first one, 1 for the second.

#### **Default Implementation:**

```
{ return 0.0f; }
```

#### **Prototype:**

```
virtual void SetOptionalParam(long nParam, float val);
```

#### **Remarks:**

Sets the value of the specified optional parameter.

#### **Parameters:**

### **long nParam**

The zero based index of the optional parameter: 0 for the first one, 1 for the second.

### **float val**

The value to set.

#### **Default Implementation:**

```
{ }
```

#### **Prototype:**

**virtual void ExecuteParamDialog(HWND hWndParent, StdMat2\* mtl);**

**Remarks:**

This method is called to put up a modal dialog which allows editing of the extended parameters. The rest of the operation of 3ds max should be disabled by this modal dialog (which is why you should use **GetMAXHwnd()**). This method is called when the Advanced button is pressed (which is enabled by using the **ADVANCED\_DLG\_BUTTON** flag returned from **SupportsStdParams()**).

**Parameters:**

**HWND hWndParent**

The parent window handle. Use **Interface::GetMAXHwnd()**.

**StdMat2\* mtl**

Points to the owning Standard material.

**Default Implementation:**

**{}**

## Class FilterKernel

See Also: [Class SpecialFX](#), [Class SFXParamDlg](#).

```
class FilterKernel : public SpecialFX
```

### Description:

This class is available in release 3.0 and later only.

This is the plug-in class from which developers sub-class their Anti-aliasing filters. These filters appear in the Render Scene dialog in the Anti-Aliasing section in the Filter dropdown list.

Filter are a very simple plug-in as there are only a few methods that define them. The **KernelFn()** method is the one that does the filtering. The other methods are simply informational.

A filter kernel is nothing but a curve that starts (usually at 0) at some distance R from a center pole. This curve is swept around the center pole forming a volume that is centered at the center of a pixel (0.5,0.5) and extends to cover some of the near neighboring pixels. The height of this filter "hat" gives the weight for each pixel under the filter. To achieve high resolution and good quality this convolution is done at full 8x8 sub-pixel resolution: each subpixel is weighted by the height of the curve above it. This is what the KernelFn() method returns. It is given the distance from the center pole and it returns the weight. This is the only method that computes anything.

Note: When things get rasterized in 3ds max it is done so at a resolution higher than that of the final output raster. 3ds max actually rasterizes the geometry to an 8x8 raster **within each** pixel. This mask is used do hiding and shading operations properly **inside** each pixel. Each of these 64 inside-a-pixel pixels is called a subpixel.

Theoretically, the **KernelFn()** function could get called once for each sub-pixel with the distance of the center of that subpix to the center pole. Of course that would take a great deal of time. Instead 3ds max builds a table at the beginning of a render. This table is slow to build (it requires many calls to **KernelFn()**) but fast to use and gives **exactly** the same answer as doing the computationally intense approach at the sub-pixel level. Thus the **KernelFn()** function can be fairly slow yet the render still happens relatively fast.

### Plug-In Information:

Class Defined In RENDER.H

Super Class ID FILTER\_KERNEL\_CLASS\_ID

Standard File Name Extension DLK

Extra Include File Needed None

**Methods:**

public:

**Prototype:**

**virtual double KernelFn(double x, double y = 0.0)=0;**

**Remarks:**

This is the function that is called to sample the kernel values. This returns the weight of the filtering curve at the specified distance from the center pole of the curve.

**Parameters:**

**double x**

The distance from the center pole of the curve.

**double y = 0.0**

The distance from the center pole of the curve in y (for 2D filters).

**Prototype:**

**virtual bool Is2DKernel()=0;**

**Remarks:**

Most kernels are 1D and hence circularly symmetric about the center pole, but some are 2D like a box or diamond. This method returns true if the filter is 2D and false if 1D.

A 2D kernel uses both parameters of the methods GetKernelSz() and SetKernelSz(). A 1D kernel only uses x; y need not be included in the set. Note that GetKernelSz() always requires both x & y since they are return parameters while a 1D kernel ignores y. Also note that a 2D filter provides a filter function that uses both the x and y parameters.

**Prototype:**

**virtual long GetKernelSupport()=0;**

**Remarks:**

This method returns the kernel 'support'. Support is the integer number of pixels **beyond** the center pixel that are touch in some part by the kernel. Support of 0 is 1x1: the area filter. A support value of 1 is a 3x3 filter, one pixel on all sides of the center. A support of 2 is 5x5. The size of a side of the block is always  $2*Support+1$ . Support confides how many pixels might be touched, but not the exact size of the filter.

**Prototype:**

**virtual long GetKernelSupportY()=0;**

**Remarks:**

For 2D kernels returns the Y support. See **GetKernelSupport()** above.

**Prototype:**

**virtual bool IsVariableSz()=0;**

**Remarks:**

Returns true if the filter is variable size; otherwise false. Size means the distance from the center pole where the filter function becomes essentially 0. In non-variable size filters this width is returned in **GetKernelSz()** and is usually displayed in the greyed out Size box in the user interface. In variable size filters get & set size control the bluriness.

**Prototype:**

**virtual void GetKernelSz(double& x, double& y)=0;**

**Remarks:**

Retrieves the kernel size. A 2D kernel uses both parameters of this method. A 1D kernel only uses x (y is set to 0).

**Parameters:**

**double& x**

The x size is returned here.

**double& y**

The y size is returned here.

**Prototype:****virtual void SetKernelSz(double x, double y = 0.0)=0;****Remarks:**

Stores the kernel size. A 2D kernel stores both parameters of this method. A 1D kernel stores only x.

**Parameters:****double x**

The x value to store.

**double y = 0.0**

The y value to store.

**Prototype:****virtual bool IsNormalized();****Remarks:**

Returning true from this method will disable the built-in normalizer.

Normalized means that if you have some solid color and you filter it, you get the same color out; it is not brighter or darker than the original. With positive only filters this is always possible, but with some negative lobe filters the colors overflow, so they are toned down (produce a slightly darker image, but don't overflow).

The normalizer computes the positive and negative volumes of an arbitrary filter and scales all the filter values by  $1/\text{volume}$  where  $\text{volume} = (\text{posVolume} - \text{abs}(\text{negVolume}))$ . This whole process is turned off and the filter values direct from the plug-in have already been scaled internally when this method returns true.

**Default Implementation:****{ return FALSE; }****Prototype:****virtual bool HasNegativeLobes()=0;****Remarks:**

This method tells the filtering code that it can speed things up potentially by dealing with the positive common only case. Currently this is not taken

advantage of.

**Prototype:**

**virtual TCHAR\* GetDefaultComment()=0;**

**Remarks:**

Returns a pointer to a string which describes the filter. This string is displayed in the static text box in the user interface.

**Prototype:**

**virtual long GetNFilterParams()=0;**

**Remarks:**

There are two optional parameters that may be used by the filter (besides Filter Size). This method returns the number used. If two parameters are used both hidden spinners appear in the Anti-aliasing section of the Render Scene dialog. If only one parameter is used just the top spinner appears. If this method returns nonzero then the methods below are used to supply the names for the parameter(s) and to provide and receive the values.

**Prototype:**

**virtual TCHAR \*GetFilterParamName(long nParam)=0;**

**Remarks:**

Returns a pointer to the string containing the name of the specified parameter.

**Parameters:**

**long nParam**

The index of the parameter (0 or 1).

**Prototype:**

**virtual double GetFilterParam(long nParam)=0;**

**Remarks:**

Returns the specified parameter value.

**Parameters:**

**long nParam**

The index of the parameter (0 or 1).

**Prototype:**

**virtual void SetFilterParam(long nParam, double val)=0;**

**Remarks:**

Stores the value passed for the specified parameter.

**Parameters:**

**long nParam**

The index of the parameter (0 or 1).

**double val**

The value to set.

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

Saves the filter name. This should be called at the start of a plug-in's save method.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

Loads the filter name. This should be called at the start of a plug-in's load method.

## Class ShadowType

### Description:

**This class is only available in release 5 or later.**

The user of GetAreaShadowType() is a linear or area light.

The usage is:

```
void AreaShadowLightObjDesc::createShadowGenerator(
    AreaLight*  light,
    bool        forceShadowBuf
)
{
    ShadowType* shad = light->ActiveShadowType();
    IAreaShadowType* area = shad->GetAreaShadowType();

    // If we aren't forcing Shadow Map and the shadow generator
    // supports area shadows, then create the area shadow generator.
    // The flags are the same as for CreateShdowGenerator
    if (!forceShadowBuf && area != NULL) {
        _areaShadGen = area->CreateAreaShadowGenerator(light, this,
            SHAD_2SIDED)
    } else {
        _shadGen = shad->CreateShadowGenerator(light, this,
SHAD_2SIDED);
    }
}
```

Sampling the area shadows is a little tricky to allow for some optimizaton.

This is an example of the code needed in AreaShadowLightObjDesc::Illuminate.

The variable, sampler, should be local to allow multithreading.

```
AreaShadowSampler* sampler = _areaShadGen->InitializeSampler(  
    alloca(_areaShadGen->GetSamplerSize()));
```

Once the sampler has been initialized, you can calculate the visibility between any point on the light and the point being shaded by using:

```
float atten = sampler->(sc, pointOnLight, shadedNormal,  
lightColor);
```

The value of pointOnLight depends on the type of light we are sampling. If the light is parallel, then pointOnLight needs to be in the local light coordinates. If the light is not parallel, then pointOnLight needs to be in camera coordinates.

## Class ShadowGenerator

See Also: [Class ShadowType](#), [Class ShadBufRenderer](#), [Class RendContext](#), [Class RenderGlobalContext](#), [Class ShadeContext](#), [Class Matrix3](#), [Class Point3](#), [Class Color](#).

class ShadowGenerator

### Description:

This class is available in release 3.0 and later only.

This class is used by a Shadow Type plug-in to generate the shadows. It only exists during a render, with one per instance of the light. Methods of this class perform the shadow buffer creation and sampling.

The ShadowGenerator API allows for two methods of sampling: A generator can use either a "generic" sampling method:

**float Sample(ShadeContext &sc, Point3 &norm, Color& color);**

Or, if it the generator is to work with Volumetric lights, it must use the following sampling shadow-map style interface:

**float Sample(ShadeContext &sc, float x, float y, float z, float xslope, float yslope);**

**BOOL QuickSample(int x, int y, float z);**

**FiltSample(int x, int y, float z, int level);**

**LineSample(int x1, int y1, float z1, int x2, int y2, float z2);**

To indicate that the latter interface is used, the method

**ShadowType::SupportStdMapInterface()** must return TRUE;

**The following functions are not part of this class but are available for use:**

**Function:**

**ShadowType \*NewDefaultShadowMapType();**

**Remarks:**

This global function returns a new default shadow-map shadow generator.

**Function:**

**ShadowType \*NewDefaultRayShadowType();**

**Remarks:**

This global function returns a new default ray-trace shadow generator.

**Methods:**

public:

**Prototype:**

**virtual int Update(TimeValue t, const RendContext& rendCntxt, RenderGlobalContext \*rgc, Matrix3& lightToWorld, float aspect, float param, float clipDist = DONT\_CLIP)=0;**

**Remarks:**

This is called on every frame to create a new shadow buffer for that frame. For example, the objects in the scene will have moved to different position, etc., so a new shadow buffer will need to be set up. See [Class ShadBufRenderer](#) for a helper class used for generating shadow map buffers.

**Parameters:**

**TimeValue t**

The time for the update.

**const RendContext& rendCntxt**

The render context -- this is used for the progress bar.

**RenderGlobalContext \*rgc**

This is used to get an instance list.

**Matrix3& lightToWorld**

The light to world space transformation matrix. This is not necessarily the same as that of the light.

**float aspect**

This is the aspect ratio for non-square buffers. The aspect gives the height/width ratio of the shadow rectangle. The shadow buffer bitmap is always the same number of pixels wide as it is high, but it can be mapped into a non-square rectangle.

**float param**

This is the field-of-view of the light in radians for perspective projections or the width in world coordinates for parallel projections.

### **float clipDist = DONTCLIP**

This parameter specifies the far clipping distance for the light. This is used when the far distance attenuation is turned on, and can result in much more efficient shadow buffer creation. If you have a small scene in the middle of a large complex scene, and the small scene is lit by, for instance, a shadow-casting omni, if you don't use far attenuation the omni has to take into account the entire large scene in its shadow map. Using far attenuation will clip all this outside stuff. Also omnies free up any of their 6 shadow buffer that end up being empty, so this can save memory usage.

#### **Return Value:**

Nonzero on success; otherwise zero.

#### **Prototype:**

```
virtual int UpdateViewDepParams(const Matrix3&  
worldToCam)=0;
```

#### **Remarks:**

If things such as automatic cubic maps or mirror are used, the rendering is done from several different points of view. This method is called to allow the view matrix to be computed and cached so it won't have to be computed over and over again. The shadow buffer caches the matrix that does the transformation from the current view coordinates into its coordinates.

#### **Parameters:**

```
const Matrix3& worldToCam
```

This is the direction the view is looking from. Object coordinates are relative to this 'camera'. This is not always a 'camera', it is just world to whatever view is needed, for example from a mirror.

#### **Return Value:**

Nonzero on success; otherwise zero.

#### **Prototype:**

```
virtual void FreeBuffer()=0;
```

#### **Remarks:**

This method is used to delete the memory associated with the buffer.

**Prototype:**

```
virtual void DeleteThis()=0;
```

**Remarks:**

Call this to destroy the ShadowGenerator.

**Prototype:**

```
virtual float Sample(ShadeContext &sc, Point3 &norm, Color&  
color);
```

**Remarks:**

Generic shadow sampling function. Implement this when **ShadowType::SupportStdMapInterface()** returns FALSE.

This is the Sample method used for ray traced shadows, for example. It takes the color that would illuminate the surface if there were no shadows, and returns a modified value. The shade context provides the point on the surface (**sc.P()**) and **norm** is the normal to the surface.

**Parameters:**

**ShadeContext &sc**

The shade context provides the point on the surface (**sc.P()**).

**Point3 &norm**

This is the normal to the surface.

**Color& color**

The input color.

**Return Value:**

It returns an attenuation, where 1.0 indicates it is not in shadow, and 0.0 indicates it is in shadow.

**Default Implementation:**

```
{ return 1.0f; }
```

**Prototype:**

```
virtual float Sample(ShadeContext &sc, float x, float y, float z,  
float xslope, float yslope);
```

**Remarks:**

Implement this method when **ShadowType::SupportStdMapInterface()** returns TRUE. This interface allows illuminated atmospheric.

This method is called to determine how much the point (x, y, z) is in shadow. It returns an attenuation, where 1.0 indicates it is not in shadow, and 0.0 indicates it is in shadow, and potentially a small negative number. A small negative number should be returned when the sample falls outside of the buffer (this is needed in order to fix a problem occurring with Omni Lights when using shadow maps). All shadow generators that implement this function need to do this. The value itself isn't important, as long as it is negative and very small (for instance (-float(1.0e-30))).

**Parameters:**

**ShadeContext &sc**

The shade context.

**float x**

The x coordinate of the point to check. This point is normalized into shadow buffer space. For example if the shadow buffer was 256x256 a point at the center would be 128, 128.

**float y**

The y coordinate of the point to check. This point is normalized into shadow buffer space.

**float z**

The z coordinate of the point to check. This is the distance perpendicular to the light where 0.0 is right at the light.

**float xslope**

This indicates the slope of the surface relative to the shadow buffer in x.

**float yslope**

This indicates the slope of the surface relative to the shadow buffer in y.

**Default Implementation:**

```
{ return 1.0f; }
```

**Prototype:**

```
virtual BOOL QuickSample(int x, int y, float z);
```

**Remarks:**

Implement this method when **ShadowType::SupportStdMapInterface()** returns TRUE. This interface allows illuminated atmospheric.

This method determines if the given point is in a shadow. It samples a single pixel in the shadow map.

**Parameters:**

**int x**

The x coordinate of the point to check. This point is normalized into shadow buffer space. For example if the shadow buffer was 256x256 a point at the center would be 128, 128.

**int y**

The y coordinate of the point to check. This point is normalized into shadow buffer space.

**float z**

The z coordinate of the point to check. This is the distance perpendicular to the light where 0.0 is right at the light.

**Return Value:**

TRUE if the point is in shadow; otherwise FALSE.

**Default Implementation:**

```
{ return 1; }
```

**Prototype:**

```
virtual float FiltSample(int x, int y, float z, int level);
```

**Remarks:**

Implement this method when **ShadowType::SupportStdMapInterface()** returns TRUE. This interface allows illuminated atmospheric.

This method is called to determine how much the point (x, y, z) is in shadow. It returns an attenuation, where 1.0 indicates it is not in shadow, and 0.0 indicates it is in shadow. The method **QuickSample()** above looks at a single pixel in the shadow buffer. This method looks at either 4 or 8 pixels (based on the level parameter) to compute the result. The center pixel is given the highest weighting, while the other pixels are given lesser weightings. However this method is still fairly quick, since it doesn't base the weighting on the location within the pixel. This is in contrast to the **Sample()** method above, where the blending of the adjacent pixels is weighted by the position within

the sub-pixel.

**Parameters:**

**int x**

The x coordinate of the point to check. This point is normalized into shadow buffer space. For example if the shadow buffer was 256x256 a point at the center would be 128, 128.

**int y**

The y coordinate of the point to check. This point is normalized into shadow buffer space.

**float z**

The z coordinate of the point to check. This is the distance perpendicular to the light where 0.0 is right at the light.

**int level**

This may be 0 or 1. If 0, four neighboring pixels are blended in. If 1, eight neighboring pixels are blended in.

**Return Value:**

A value in the range 0.0 to 1.0.

**Default Implementation:**

```
{ return 1.0f; }
```

**Prototype:**

```
virtual float LineSample(int x1, int y1, float z1, int x2, int y2, float z2);
```

**Remarks:**

Implement this method when **ShadowType::SupportStdMapInterface()** returns TRUE. This interface allows illuminated atmospherics.

This method is called to sample the shadow map along a line segment. It uses a line between x1, y1 and x2, y2. The z values are interpolated between z1 and z2 and compared to the z value in the shadow map for that pixel.

**Parameters:**

**int x1**

The start x coordinate of the line. This point is normalized into shadow buffer space. For example if the shadow buffer was 256x256 a point at the center

would be 128, 128.

**int y1**

The start y coordinate of the line. This point is normalized into shadow buffer space.

**float z1**

The start z coordinate of the line. This is the distance perpendicular to the light where 0.0 is right at the light.

**int x2**

The end x coordinate of the line. This point is normalized into shadow buffer space.

**int y2**

The end y coordinate of the line. This point is normalized into shadow buffer space.

**float z2**

The end z coordinate of the line. This is the distance perpendicular to the light where 0.0 is right at the light.

**Return Value:**

A value in the range 0.0 to 1.0 which represents how much of the ray was inside the light and how much was outside the light.

**Default Implementation:**

```
{ return 1.0f; }
```

## Class ShadBufRenderer

See Also: [Class ShadowGenerator](#), [Class RendContext](#), [Class RenderGlobalContext](#), [Class ObjLightDesc](#), [Class Matrix3](#).

```
class ShadBufRenderer
```

### Description:

This class is available in release 3.0 and later only.

This class is used to generate a Shadow Buffer which may be used to determine if a point is in shadow or not. The 3ds max shadow maps use this object internally, for example.

There is a global function that creates one of these **ShadBufRenderer** objects. With one of these developers can call its **Render()** method to generate (render) a Shadow Buffer.

The rendered shadow buffer stores a Z distance at every point in the buffer. This can then be used to determine if something is in shadow. To check a certain point you simply see if the Z value is behind the one in the buffer. That is, a shadow buffer tells one, from the point of view of a light, how far it is to the first object for each pixel in the buffer. If the Z point of the thing being shadowed is farther than (behind) the corresponding Z value in the buffer then the thing is in shadow. If it's closer than it is not in shadow.

The main **Render()** method is typically called from the **Update()** method of class **ShadowGenerator** which is called on every frame to create a new shadow buffer.

To use this class you basically do the following:

Allocate an array of floating point values, one float for each point in the shadow buffer:

```
buffer = new float[shadsize*shadsize];
```

Then create a default Shadow Buffer Renderer using the global function provided:

```
ShadBufRenderer *sbr = NewDefaultShadBufRenderer();
```

Then you setup all the parameters for the view, etc prior to calling the **Render()** method to render the buffer. (These parameters are passed in to the **ShadowGenerator::Update()** method).

```
int nRendered = sbr->Render(rc, RGC, buffer, parallel, shadsize,
```

**param, aspect, clipDist, ltDesc, worldToLight);**

You can check the return value to determine if any objects were intersected by the shadow volume. If none were, the shadow buffer can be freed.

```
if (nRendered==0) {  
    delete [] buffer;  
    sbr->DeleteThis();  
    buffer = NULL;  
    return 1;  
}
```

All methods of this class are implemented by the system.

**The following global function is not a member of this class but is available for use:**

**Function:**

```
ShadBufRenderer *NewDefaultShadBufRenderer();
```

**Remarks:**

This global function creates and returns a pointer to a new default shadow buffer renderer.

**Methods:**

public:

**Prototype:**

```
virtual int Render(RendContext &rc, RenderGlobalContext  
*RGC, float *buf, BOOL parallel, int shadsize, float param, float  
aspect, float clipDist, ObjLightDesc *ltDesc, Matrix3  
worldToLight)=0;
```

**Remarks:**

Compute a shadow Z buffer for the current scene from the viewpoint of the light. NOTE: The computed shadow buffer has positive Z values as you go away from the light, which is the reverse of the 3ds max coordinate system.

**Parameters:**

```
RendContext &rc
```

The RendContext which is used for the progress bar API.

**RenderGlobalContext \*RGC**

Points to the RenerGlobalContext which is used to retireve information about the global rendering enviornment (to get an instance list).

**float \*buf**

This is the buffer to render to. This is a pre-allocated array of floats (**shadsize\*shadsize**).

**BOOL parallel**

The projection type. TRUE if parallel projection; FALSE if perspective projection.

**int shadsize**

The size of the buffer (shadsize by shadsize pixels).

**float param**

The view parameter. For a perspective this is the:field-of-view (in radians). For a parallel view this is the width in world coordinates.

**float aspect**

This is the aspect ratio of the buffer projection.

**float clipDist**

The clipping distance. This tells the shadow buffer renderer to not consider objects farther than this distance from light.

**ObjLightDesc \*ltDesc**

This is the descriptor for light that was passed in to CreateShadowGenerator().

**Matrix3 worldToLight**

The world to light transformation matrix for the light.

**Return Value:**

Returns the number of objects that the shadow volume intersected. If this value is 0, the shadow buffer can be freed to save memory.

**Prototype:**

**virtual float Furthest()=0;**

**Remarks:**

After a render, this method returns the farthest Z in the shadow buffer.

**Prototype:**

**virtual float Closest()=0;**

**Remarks:**

After a render, this method returns the closest Z in the shadow buffer.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Deletes this **ShadowBufRenderer** object.

## Class SoundObj

See Also: [Class ReferenceTarget](#), [Class Animatable](#).

```
class SoundObj : public ReferenceTarget
```

### Description:

This is the base class for the creation of sound plug-ins. The 3ds max user may choose a sound plug-in using the File / Preferences... Animation Tab / Sound Plug-In option.

There is always one sound object in the scene. A sound object's primary purpose is to provide a sound track for the scene. The sound object also serves as a clock that controls timing when an animation is played. This ensure the animation is synched to the sound object. This class has methods to start and stop the sound playing, play a specified range of the sound, and toggle the sound on and off.

A sound plug-in can participate in Track View by implementing the methods of Animatable such as **PaintTrack()**. See the Advanced Topics section [Track View](#) for details.

Sound Object plug-ins use a Super Class ID of **SOUNDOBJ\_CLASS\_ID**.

### Methods:

#### Prototype:

```
virtual BOOL Play(TimeValue tStart,TimeValue t0,TimeValue t1,TimeValue frameStep)=0;
```

#### Remarks:

Implemented by the Plug-In.

When the system calls this method the plug-in should loop the playing of sound from time **t0** to **t1** beginning at time **tStart**. It should continue to loop until **Stop()** is called.

#### Parameters:

**TimeValue tStart**

The time to start playing the sound.

**TimeValue t0**

The loop begin range.

**TimeValue t1**

The loop end range.

**TimeValue frameStep**

The frame increment.

**Return Value:**

TRUE if the sound was played; FALSE otherwise.

**Prototype:**

**virtual void Scrub(TimeValue t0,TimeValue t1)=0;**

**Remarks:**

Implemented by the Plug-In.

Implementation of this method is optional. The plug-in should play the amount of sound between time **t0** and **t1**. The sound should only be played once.

**Parameters:**

**TimeValue t0**

The start time for playback.

**TimeValue t1**

The end time for playback.

**Prototype:**

**virtual TimeValue Stop()=0;**

**Remarks:**

Implemented by the Plug-In.

This stops the sound from playing.

**Return Value:**

The time at which the sound was stopped.

**Prototype:**

**virtual TimeValue GetTime()=0;**

**Remarks:**

Implemented by the Plug-In.

This returns the current time as managed by the **SoundObj**.

**Return Value:**

The current time.

**Prototype:**

**virtual BOOL Playing()=0;**

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the sound is playing; otherwise FALSE.

**Prototype:**

**virtual void SaveSound(PAVIFILE pfile,TimeValue t0,TimeValue t1)=0;**

**Remarks:**

Implemented by the Plug-In.

This saves the sound between the specified times to the specified file.

**Parameters:**

**PAVIFILE pfile**

The file to save the sound track to.

**TimeValue t0**

The start of the time range to save.

**TimeValue t1**

The end of the time range to save.

**Prototype:**

**virtual void SetMute(BOOL mute)=0;**

**Remarks:**

Implemented by the Plug-In.

Sets the sound to mute or toggles it back on. This will be called if the Active checkbox is toggled for example.

**Parameters:**

**BOOL mute**

Specifies if the sound should be muted. TRUE indicates the sound should be

muted; FALSE indicates the sound should be enabled.

**Prototype:**

**virtual BOOL IsMute()=0;**

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the sound is muted; otherwise FALSE.

**The following function is not a method of class SoundObj but is used by the system internally:**

**Function:**

**SoundObj \*NewDefaultSoundObj();**

**Remarks:**

Implemented by the System.

Returns a new default sound object. This is the standard one provided by 3ds max.

# Class ColPick

See Also: [Class ColorPicker](#), [Class HSVCallback](#), [Class Class\\_ID](#), [Class IPoint2](#), [DWORD--COLORREF Color Format..](#)

class ColPick : public InterfaceServer

## Description:

This class is available in release 3.0 and later only.

This is the base class for the creation of plug-in color selectors. The list of available color pickers appear in the 3ds max user interface in the General page of the Preferences dialog. The chosen picker will be called whenever a user clicks on a 3ds max color swatch control.

## Plug-In Information:

Class Defined In HSV.H

Super Class ID COLPICK\_CLASS\_ID

Standard File Name Extension DLU

Extra Include File Needed HSV.H

## Functions:

These global functions are not part of this class but are used internally by 3ds max to plug in the current color picker. Developers should not need to access these.

## Function:

**ColPick \*SetCurColPick(ColPick \*colpick);**

## Remarks:

This function is available in release 3.0 and later only.

This global function is used internally to establish the current color picker used.

## Parameters:

**ColPick \*colpick**

Points to the color picker to use.

## Return Value:

A pointer to the current color picker.

**Function:**

**ColPick \*GetCurColPick();**

**Remarks:**

This function is available in release 3.0 and later only.

Returns a pointer to the current color picker.

**Methods:**

public:

**Prototype:**

**virtual INT\_PTR ModalColorPicker(HWND hwndOwner,  
DWORD \*lpc, IPoint2 \*spos, HSVCallback \*callBack, TCHAR  
\*name)=0;**

**Remarks:**

Implemented by the Plug-In

This method is called to bring up the modal color picker.

**Parameters:**

**HWND hwndOwner**

The owning window handle

**DWORD \*lpc**

A pointer to the color to be edited. See [DWORD COLORREF Format](#).

**IPoint2 \*spos**

The starting position of the dialog. This is set to ending position on return.

**HSVCallback \*callBack**

This callback is called whenever color changes.

**TCHAR \*name**

The name of color being edited

**Return Value:**

TRUE if the user pressed OK; FALSE on cancel.

**Prototype:**

**virtual ColorPicker \*CreateColorPicker(HWND  
hwndOwner, DWORD initColor, IPoint2\* spos, HSVCallback  
\*pcallback, TCHAR \*name, BOOL isObjectColor=FALSE)=0;**

**Remarks:**

Implemented by the Plug-In

This method is called to create and return a ColorPicker object for the modeless color picker.

**Parameters:****HWND hwndOwner**

The owning window handle.

**DWORD initColor**

The initial value of the color. See [DWORD COLORREF Format](#).

**IPoint2\* spos**

The starting position of dialog.

**HSVCallback \*pcallback**

This callback is called whenever color changes.

**TCHAR \*name**

The name of color being edited.

**BOOL isObjectColor=FALSE**

This indicates the color picker is being used for the object color in the command panel, and the color picker then displays the Add Color button

**Prototype:**

```
virtual const TCHAR *ClassName()=0;
```

**Remarks:**

Implemented by the Plug-In

Returns the name of the class. This name appears in the drop down list of color picker choices.

**Prototype:**

```
virtual Class_ID ClassID()=0;
```

**Remarks:**

Implemented by the Plug-In

Returns the unique ClassID of this plug-in. The Class\_ID for the default color picker is **Class\_ID(DEFAULT\_COLPICK\_CLASS\_ID,0)**.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Implemented by the Plug-In

This method is called to delete this instance of the plug-in class.

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0)=0;**

**Remarks:**

Implemented by the Plug-In

This method is used for future expansion and is currently not used.

## **Class FrontEndController**

This class and its former purpose has been replaced by the [CUIFrameMgr](#).

## Class MCInputDevice

See Also: [Class MCDeviceBinding](#), [Class IMCapManager](#), [Time](#), [Character Strings](#).

```
class MCInputDevice
```

### Description:

This class is available in release 2.0 and later only.

This is the base class for an input device plug-in. All methods of this class are implemented by the plug-in.

In terms of the motion capture system, the basic item that is plug-able is the motion capture device. This is something like a mouse, joystick, or midi device. Developers implement two classes, this one, **MCInputDevice**, and **MCDeviceBinding**. There is usually only one instance of **MCInputDevice**. This is like the virtual mouse, or the joystick. This represents the actual device. An instance of the device binding represents an instance where a motion capture controller has been bound to a device, i.e. the user has picked the device and assigned it to a parameter. Thus there may be many instances of the device binding. The device binding is part of the reference hierarchy. The device itself doesn't usually have any parameters for the user to adjust -- these are rather part of the device binding.

Some simple sample code for the mouse motion capture device is available in **\MAXSDK\SAMPLES\MOCAP\MCDEVICE.CPP**.

### Methods:

#### Prototype:

```
virtual TSTR DeviceName()=0;
```

#### Remarks:

Returns the name for the input device.

#### Prototype:

```
virtual MCDeviceBinding *CreateBinding()=0;
```

#### Remarks:

The motion capture utility creates a list of all the MCInputDevices in the system. When the user wants to pick one it will call this method. It returns a

new instance of the **MCDeviceBinding** class.

**Prototype:**

**virtual void UtilityStarted(IMCapManager \*im);**

**Remarks:**

This method is called when the user enters the utility.

**Parameters:**

**IMCapManager \*im**

This is an interface into the motion capture manager.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void UtilityStopped(IMCapManager \*im);**

**Remarks:**

This method is called when the user leaves the utility.

**Parameters:**

**IMCapManager \*im**

This is an interface into the motion capture manager.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void Cycle(UINT tick);**

**Remarks:**

This method is called when the user is in 'Record' (capture) mode or 'Test' mode. It is called once per millisecond. For instance the joystick device uses this method. To understand this method consider the following example:

With MIDI you don't call a function to see if a key has been pressed or not -- rather it is a message based system where you're notified if something happens. In contrast to this is the joystick. If the user moves the joystick the program is not notified. Rather a developer must poll the joystick to get its

current position. During motion capture one could poll the joystick at every frame to get its current position. However this approach leads to jittering (aliasing). The problem is that, on average, the joystick is providing a smooth series of values, but instantaneously, the values jump around a bit. So, the joystick motion capture plug-in implements this method to stores the values returned at every millisecond. Then later, when needing to sample the joystick at a certain time, the stored table of values can be averaged and this provides a level of smoothing.

**Parameters:**

**UINT tick**

The time of this call in milliseconds.

**Default Implementation:**

```
{
```

## Class ViewFile

class ViewFile

### Description:

This class allows a developer to replace the file viewer used by 3ds max (This is the "View File" option in 3ds max's File menu). By creating a DLL from this class, and replacing the standard 3ds max ViewFile. DLL the system will always use the developer defined version. Note: To execute this plug-in, put the DLL in the same directory as the 3DSMAX.EXE executable.

The following two functions are called by the system to create and delete the instance of this class that handles the file viewing.

### **void \*ViewFileCreate();**

This function is implemented by the plug-in to create a new instance of this class. For example:

```
void *ViewFileCreate() {  
    return new ViewFile;  
}
```

### **void ViewFileDestroy(ViewFile \*v);**

This function is implemented by the plug-in to delete the instance of this class created above. For example:

```
void ViewFileDestroy(ViewFile *v) {  
    if (v)  
        delete v;  
}
```

### Methods:

### Prototype:

```
void View(HWND hWnd)
```

### Remarks:

Implemented by the Plug-In.

This method is called by the system to bring up the file viewer.

### Parameters:

```
HWND hWnd
```

The parent window handle.

See Also: `\MAXSDK\SAMPLES\VIEWFILE\VIEWFILE.CPP` for an example of the standard 3ds max file viewer.

## Interface Class Overview

See Also: [Links to Interface Classes](#), [Class Interface](#), [Class ViewExp](#), [Class INode](#).

3ds max provides a large number of functions for plug-ins to use. There are two approaches plug-ins may use to call these functions in MAX:

- 1 Plug-Ins can directly call functions in **CORE.DLL**
- 2 They may call methods on an interface pointer.

The primary purpose of **CORE.DLL** is that it contains implementations for some of the methods of the base classes that plug-ins derive their classes from. Plug-Ins generally don't call these methods directly, but instead inherit them as part of their derived class. **CORE.DLL** also has other miscellaneous functions that it exports. Plug-ins link to **CORE.LIB** so they can simply call these methods like any other function.

Unlike **CORE.DLL**, **3DSMAX.EXE** does not export any methods. Instead, a plug-in calls functions in 3ds max through a pointer to an interface class. An interface class is simply a class with no data members and all pure virtual methods. These classes are essentially just a table of function pointers.

Some of these interfaces are general. For example, the class **Interface** provides methods to do things like redraw the viewports, work with standard 3ds max dialog boxes, and work with MAX's coordinate systems. Other interfaces have specific contexts. For instance, the **ViewExp** interface class provides an interface to a viewport. Methods of this interface refer to a specific viewport.

Most interfaces have a specific life time outside of which they must not be used. For instance, when a modifier becomes active in the modify branch of the command panel, its **BeginEditParams()** method is called and an **IObjParams** interface is passed in. This interface pointer is defined to be valid until (and including) the modifier's **EndEditParams()** method is called. If a plug-in were to hang on to the pointer outside of this interval and then call one of its methods, the result would be undefined. Another example is the **INode** interface pointer, which when passed in to **Object::Display()**, is valid for only the duration of the function call.

For a list of all the interfaces into 3ds max see the section [Interface Classes](#).

Note: Developer will often see the storage class attributes **CoreExport** and

**DllExport** in the source code. These attributes are simply Microsoft-specific extensions to the C++ language used to enable 3ds max to export functions, data, and objects from a DLL so that plug-ins may call them.

# Class Point2

See Also: [Class IPoint2](#).

class Point2

## Description:

This class describes a 2D point using float x and y coordinates. Methods are provided to add and subtract points, multiply and divide by scalars, normalize and compute the dot product of two Point2s. All methods are implemented by the system.

## Data Members:

public:

**float x,y;**

The x and y components of the point.

**static const Point2 Origin;**

This data member is available in release 3.0 and later only.

This is equivalent to **Point2(0.0f, 0.0f);**

**static const Point2 XAxis;**

This data member is available in release 3.0 and later only.

This is equivalent to **Point2(1.0f, 0.0f);**

**static const Point2 YAxis;**

This data member is available in release 3.0 and later only.

This is equivalent to **Point2(0.0f, 1.0f);**

## Methods:

Constructors

## Prototype:

**Point2()**

## Remarks:

Constructor. No initialization is performed by this constructor.

## Prototype:

## **Point2(float X, float Y)**

### **Remarks:**

Constructor. Data members are initialized to X and Y.

### **Prototype:**

## **Point2(double X, double Y)**

### **Remarks:**

Constructor. Data members are initialized to X and Y cast as floats.

### **Prototype:**

## **Point2(const Point2& a)**

### **Remarks:**

Constructor. Data members are initialized to a.x and a.y.

### **Prototype:**

## **Point2(float af[2])**

### **Remarks:**

Constructor. Data members are initialized as  $x = af[0]$  and  $y = af[1]$ .

### **Prototype:**

## **Point2& Set(float X, float Y);**

### **Remarks:**

This method is available in release 3.0 and later only.

Sets the x and y coordinate to the values passed and returns a reference to this Point2.

### **Parameters:**

#### **float X**

The new x value.

#### **float Y**

The new y value.

### **Return Value:**

A reference to this **Point2**.

**Prototype:**

**float DotProd(const Point2&) const**

**Remarks:**

Returns the dot product of two Point2's. This is the sum of both x values multiplied together and both y values multiplied together.

**Prototype:**

**float Length() const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the length of the point. This is **sqrt(v.x\*v.x+v.y\*v.y)**;

**Prototype:**

**int MaxComponent() const;**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns the component with the maximum absolute value.

**Return Value:**

0 for X, 1 for Y, 2 for Z.

**Prototype:**

**int MinComponent() const;**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns the component with the minimum absolute value.

**Return Value:**

0 for X, 1 for Y, 2 for Z.

**Prototype:**

**Point2 Normalize() const;**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns a normalized version of this Point2. This method is more accurate than **\*this/Length()** (internal computations are done in double precision).

**Prototype:**

```
int Equals(const Point2& p, float epsilon = 1E-6f);
```

**Remarks:**

This method is available in release 3.0 and later only.

Compares this Point2 and the specified one to see if the x and y values are within plus or minus the specified tolerance.

**Parameters:**

**const Point2& p**

The point to compare.

**float epsilon = 1E-6f**

The tolerance to use in the comparison.

**Return Value:**

Nonzero if the points are 'equal'; otherwise zero.

**Prototype:**

```
Point2& Unify();
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to unify (or normalize) this Point2 (in place) and return the result. Internal computations are done in double precision.

**Prototype:**

```
float LengthUnify();
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to unify (or normalize) this Point2 (in place) and return the previous length. Internal computations are done in double precision.

**Operators:**

**Prototype:**

**float& operator[](int i)  
const float& operator[](int i) const**

**Remarks:**

Allows access to x, y using the subscript operator.

**Return Value:**

A value for **i** of 0 will return x, 1 will return y.

**Prototype:**

**operator float\*()**

**Remarks:**

Returns the address of the Point2.x

**Prototype:**

**Point2 operator-() const**

**Remarks:**

Unary -. Negates both x and y.

**Return Value:**

A Point2 with -x, -y.

**Prototype:**

**Point2 operator+() const**

**Remarks:**

Unary +. Returns the point unaltered.

**Return Value:**

Returns the Point2 unaltered.

**Prototype:**

**Point2& operator-=(const Point2&)**

**Remarks:**

Subtracts a Point2 from this Point2.

**Return Value:**

A Point2 that is the difference between two Point2s.

**Prototype:**

**Point2& operator+=(const Point2&)**

**Remarks:**

Adds a Point2 to this Point2.

**Return Value:**

A Point2 that is the sum of two Point2's.

**Prototype:**

**Point2& operator\*=(float)**

**Remarks:**

Multiplies this Point2 by a floating point value.

**Return Value:**

A Point2 multiplied by a float.

**Prototype:**

**Point2& operator/=(float)**

**Remarks:**

Divides this Point2 by a floating point value.

**Return Value:**

A Point2 divided by a float.

**Prototype:**

**Point2 operator-(const Point2&) const**

**Remarks:**

Subtracts a Point2 from a Point2.

**Return Value:**

A Point2 that is the difference between two Point2's.

**Prototype:**

**Point2 operator+(const Point2&) const**

**Remarks:**

Adds a Point2 to a Point2.

**Return Value:**

The sum of two Point2's.

**Prototype:**

**float operator\*(const Point2&) const**

**Remarks:**

Returns the dot product of two Point2's. This is the sum of both x values multiplied together and both y values multiplied together.

**Prototype:**

**int operator==(const Point2& p) const**

**Remarks:**

Equality operator. Compares two Point2's.

**Return Value:**

Nonzero if the Point2's are equal; otherwise 0.

The following functions are not methods of class Point2 but are available for use:

**Prototype:**

**Point2 operator\*(float, const Point2&)**  
**Point2 operator\*(const Point2&, float)**

**Remarks:**

Each returns a Point2 multiplied by a scalar.

**Prototype:**

**Point2 operator/(const Point2&, float)**

**Remarks:**

Returns a Point2 whose x and y members are divided by a scalar.

**Prototype:**

**ostream &operator<<(ostream&, const Point2&)**

**Remarks:**

Formats the Point2 for output as in:

**(x, y)**

**Function:**

**float Length(const Point2& v)**

**Remarks:**

Returns the length of the Point2, ie:

**sqrt(v.x\*v.x+v.y\*v.y);**

**Function:**

**int MaxComponent(const Point2&)**

**Remarks:**

Returns the component with the maximum absolute value. 0=x, 1=y.

**Function:**

**int MinComponent(const Point2&)**

**Remarks:**

Returns the component with the minimum absolute value. 0=x, 1=y.

**Function:**

**Point2 Normalize(const Point2&)**

**Remarks:**

Returns a unit vector. This is a Point2 with each component divided by the point **Length()**.

# Class Point3

See Also: [Class IPoint3](#), [Class DPoint3](#), [Class Matrix3](#).

class Point3

## Description:

This class describes a 3D point using float x, y and z coordinates. Methods are provided to add and subtract points, multiply and divide by scalars, and element by element multiply and divide two points.

This class is also frequently used to simply store three floating point values that may not represent a point. For example, a color value where x=red, y=green, and z=blue. For color, the range of values is 0.0 to 1.0, where 0 is 0 and 1.0 is 255. All methods are implemented by the system.

Note: In 3ds max, all vectors are assumed to be row vectors. Under this assumption, multiplication of a vector with a matrix can be written either way (Matrix\*Vector or Vector\*Matrix), for ease of use, and the result is the same -- the (row) vector transformed by the matrix.

## Data Members:

public:

**float x, y, z;**

The x, y and z components of the point.

**static const Point3 Origin;**

This data member is available in release 3.0 and later only.

This is equivalent to **Point3(0.0f, 0.0f, 0.0f);**

**static const Point3 XAxis;**

This data member is available in release 3.0 and later only.

This is equivalent to **Point3(1.0f, 0.0f, 0.0f);**

**static const Point3 YAxis;**

This data member is available in release 3.0 and later only.

This is equivalent to **Point3(0.0f, 1.0f, 0.0f);**

**static const Point3 ZAxis;**

This data member is available in release 3.0 and later only.

This is equivalent to **Point3(0.0f, 0.0f, 1.0f);**

**Methods:**

**Prototype:**

**Point3()**

**Remarks:**

Constructor. No initialization is performed.

**Prototype:**

**Point3(float X, float Y, float Z)**

**Remarks:**

Constructor. x, y, and z are initialized to the values specified.

**Prototype:**

**Point3(double X, double Y, double Z)**

**Remarks:**

Constructor. x, y, and z are initialized to the specified values (cast as floats).

**Prototype:**

**Point3(int X, int Y, int Z)**

**Remarks:**

Constructor. x, y, and z are initialized to the specified values (cast as floats).

**Prototype:**

**Point3(const Point3& a)**

**Remarks:**

Constructor. x, y, and z are initialized to the specified Point3.

**Prototype:**

**Point3(float af[3])**

**Remarks:**

Constructor. x, y, and z are initialized to af[0], af[1], and af[2] respectively.

**Operators:**

**Prototype:**

```
float& operator[](int i)
const float& operator[](int i) const
```

**Remarks:**

Allows access to x, y and z using the subscript operator.

**Return Value:**

An value for **i** of 0 will return x, 1 will return y, 2 will return z.

**Prototype:**

```
operator float*()
```

**Remarks:**

Conversion function. Returns the address of the Point3.x

**Prototype:**

```
Point3 operator-() const
```

**Remarks:**

Unary - operator. Negates x, y and z.

**Prototype:**

```
Point3 operator+() const
```

**Remarks:**

Unary +. Returns the Point3.

**Prototype:**

```
inline Point3& operator--=(const Point3&);
```

**Remarks:**

Subtracts a Point3 from this Point3.

**Prototype:**

```
inline Point3& operator+=(const Point3&);
```

**Remarks:**

Adds a Point3 to this Point3.

**Prototype:**

**inline Point3& operator\*=(float);**

**Remarks:**

Multiplies this Point3 by a floating point value.

**Prototype:**

**inline Point3& operator/=(float);**

**Remarks:**

Divides this Point3 by a floating point value.

**Prototype:**

**inline Point3& operator\*=(const Point3&);**

**Remarks:**

Element-by-element multiplication of two Point3s:  
(**x\*x, y\*y, z\*z**).

**Prototype:**

**int operator==(const Point3& p) const**

**Remarks:**

Equality operator. Test for equality between two Point3's.

**Return Value:**

Nonzero if the Point3's are equal; otherwise 0.

**Prototype:**

**inline Point3 operator-(const Point3&) const;**

**Remarks:**

Subtracts a Point3 from a Point3.

**Prototype:**

**inline Point3 operator+(const Point3&) const;**

**Remarks:**

Adds a Point3 to a Point3.

**Prototype:**

**inline Point3 operator/(const Point3&) const;**

**Remarks:**

Divides a Point3 by a Point3 element by element.

**Prototype:**

**inline Point3 operator\*(const Point3&) const;**

**Remarks:**

Multiplies a Point3 by a Point3 element by element.  
(**x\*x, y\*y, z\*z**).

**Prototype:**

**Point3 operator^(const Point3&) const;**

**Remarks:**

The cross product of two Point3's (vectors).

**Return Value:**

The cross product of two Point3's.

**The following functions are not methods of Point3 but are available for use:**

**Prototype:**

**inline float Length(const Point3& v)**

**Remarks:**

Returns the 'Length' of the point (vector). This is:  
**sqrt(v.x\*v.x+v.y\*v.y+v.z\*v.z)**

**Prototype:**

**inline float FLength(const Point3& v)**

**Remarks:**

Returns the 'Length' of the point (vector) using a faster assembly language implementation for square root. This is:

**Sqrt(v.x\*v.x+v.y\*v.y+v.z\*v.z)**

**Prototype:**

**inline float LengthSquared(const Point3& v)**

**Remarks:**

The 'Length' squared of the point. This is **v.x\*v.x+v.y\*v.y+v.z\*v.z**.

**Prototype:**

**int MaxComponent(const Point3&);**

**Remarks:**

Returns the component with the maximum absolute value. 0=x, 1=y, 2=z.

**Prototype:**

**int MinComponent(const Point3&);**

**Remarks:**

Returns the component with the minimum absolute value. 0=x, 1=y, 2=z.

**Prototype:**

**Point3 Normalize(const Point3&);**

**Remarks:**

Returns a normalized unit vector. This is a Point3 with each component divided by the point **Length()**.

**Prototype:**

**Point3 FNormalize(const Point3&);**

**Remarks:**

Returns a normalized unit vector using faster assembly language code than that used by **Normalize()**. This is a Point3 with each component divided by

the point **Length()**.

**Prototype:**

**inline Point3 operator\*(float f, const Point3& a)**

**Remarks:**

Returns a Point3 that is the specified Point3 multiplied by the specified float.

**Prototype:**

**inline Point3 operator\*(const Point3& a, float f)**

**Remarks:**

Returns a Point3 that is the specified Point3 multiplied by the specified float.

**Prototype:**

**inline Point3 operator/(const Point3& a, float f)**

**Remarks:**

Returns a Point3 that is the specified Point3 divided by the specified float.

**Prototype:**

**inline Point3 operator+(const Point3& a, float f)**

**Remarks:**

Returns a Point3 that is the specified Point3 with the specified floating point valued added to each component x, y, and z.

**Prototype:**

**inline float DotProd(const Point3& a, const Point3& b);**

**Remarks:**

Returns the dot product of two Point3s. This is the sum of each of the components multiplied together, element by element

**$a.x*b.x+a.y*b.y+a.z*b.z$**

The dot product has the property of equaling the product of the magnitude (length) of the two vector times the cosine of the angle between them.

**Prototype:**

**Point3 CrossProd(const Point3& a, const Point3& b);**

**Remarks:**

This returns the cross product of the specified Point3's (vectors). The cross product of two vectors is a third vector, perpendicular to the plane formed by the two vectors.

**Prototype:**

**ULONG CompressNormal(Point3 p);**

**Remarks:**

This function will compress a normal vector from 12 bytes to 4 bytes. The vector has to be  $\leq 1.0$  in length.

**Prototype:**

**Point3 DeCompressNormal(ULONG n);**

**Remarks:**

This function may be used to decompress a surface normal from the G-Buffer (ie the **BMM\_CHAN\_NORMAL** channel). The Point3 returned is normalized. The decompressed vector has absolute error  $<.001$  in each component.

## Class Matrix3

See Also: [Class Point3](#), [Matrix Representations of 3D Transformations](#), [Class Quat](#), [Class AngAxis](#), [Structure AffineParts](#), [Class BigMatrix](#).

class Matrix3

### Description:

This class implements a 4x3 3D transformation matrix object. Methods are provided to zero the matrix, set it to the identity, compute its inverse, apply incremental translation, rotation and scaling, and build new X, Y and Z rotation matrices. Operators are provided for matrix addition, subtraction, and multiplication. All methods are implemented by the system.

Note: In 3ds max, all vectors are assumed to be row vectors. Under this assumption, multiplication of a vector with a matrix can be written either way (Matrix\*Vector or Vector\*Matrix), for ease of use, and the result is the same -- the (row) vector transformed by the matrix.

### Data Members:

private:

**float m[4][3];**

Matrix storage.

**DWORD flags;**

Matrix Identity Flags.

**POS\_IDENT**

Indicates the translation row of the matrix is the identity.

**ROT\_IDENT**

Indicates the rotation elements of the matrix are the identity.

**SCL\_IDENT**

Indicates the scale elements of the matrix are the identity.

**MAT\_IDENT**

Indicates the matrix is the identity matrix. This is equivalent to **(POS\_IDENT|ROT\_IDENT|SCL\_IDENT)**.

public:

**static const Matrix3 Identity;**

This data member is available in release 3.0 and later only.

An instance of an identity matrix.

**Methods:**

**Prototype:**

**Matrix3()**

**Remarks:**

Constructor. Note that no initialization is done. Use **Zero()** or **Identity()**, or the constructors below.

**Prototype:**

**Matrix3(BOOL init)**

**Remarks:**

Constructor. If TRUE is passed to the method the matrix is set to the identity.

**Parameters:**

**BOOL init**

Specifies if the Matrix3 should be initialized to the identity.

**Prototype:**

**Matrix3(float (\*fp)[3]);**

**Remarks:**

Constructor. The matrix is initialized to **fp**.

**Parameters:**

**float (\*fp)[3]**

Specifies the initial values for the matrix.

**Prototype:**

**Matrix3(const Point3& U, const Point3& V, const Point3& N,  
const Point3& T);**

**Remarks:**

This method is available in release 3.0 and later only.

Constructor. Initializes the matrix with the row data passed and validates the matrix flags.

**Parameters:**

**const Point3& U**

The data for row 0.

**const Point3& V**

The data for row 1.

**const Point3& N**

The data for row 2.

**const Point3& T**

The data for row 3.

**Prototype:**

**void SetNotIdent()**

**Remarks:**

This clears the **MAT\_IDENT** flag to indicate the matrix is not the identity. If any changes are made to components directly via **GetAddr()**, this method must be called.

**Prototype:**

**Remarks:**

**void SetIdentFlags(DWORD f)**

**Remarks:**

This sets the specified identity flag(s).

**Parameters:**

**DWORD f**

Specifies the identity flag bit(s) to set. See Matrix Identity Flags above.

**Prototype:**

**DWORD GetIdentFlags() const**

**Remarks:**

Returns the identity flags.

**Prototype:**

## **void ClearIdentFlag(DWORD f)**

### **Remarks:**

Clears the specified identity flag(s). See Matrix Identity Flags above.

### **Parameters:**

#### **DWORD f**

Specifies the identity flag bit(s) to clear.

### **Prototype:**

#### **BOOL IsIdentity() const**

### **Remarks:**

Returns TRUE if the matrix is the identity matrix (based on the flags); otherwise FALSE.

### **Prototype:**

#### **void ValidateFlags();**

### **Remarks:**

This method is available in release 2.0 and later only.

This method may be used to recompute the \*\_IDENT flags for this matrix.

For instance, if you call a method, such as

**INode::GetObjTMAfterWSM()**, and it returns a matrix, you cannot use the **IsIdentity()** method to check if the matrix is indeed the identity. This is because the flags that method checks are not initialized by the INode method. What you can do however is call this method first. This will validate the flags in the matrix so they accurately reflect the properties of the matrix. If after calling this method, and then calling **IsIdentity()**, the proper result would be returned.

### **Prototype:**

#### **MRow\* GetAddr()**

### **Remarks:**

Returns the address of this Matrix3.

The Matrix3 class keeps flags indicating identity for rotation, scale, position, and the matrix as a whole, and thus the direct access via the [] operator is

restricted to prevent developers from modifying the matrix without updating the flags. This method, `GetAddr()`, still lets you get at the matrix itself and then you can use the `[]` operator on the result. Note: If you change the matrix via this pointer, you **MUST** clear the proper IDENT flags!

Also Note: **`typedef float MRow[3];`**

**Return Value:**

The address of the `Matrix3`.

**Prototype:**

**`const MRow* GetAddr() const;`**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the address of this `Matrix3`.

The `Matrix3` class keeps flags indicating identity for rotation, scale, position, and the matrix as a whole, and thus the direct access via the `[]` operator is restricted to prevent developers from modifying the matrix without updating the flags. This method, `GetAddr()`, still lets you get at the matrix itself and then you can use the `[]` operator on the result. Note: If you change the matrix via this pointer, you **MUST** clear the proper IDENT flags!

Also Note: **`typedef float MRow[3];`**

**Prototype:**

**`void IdentityMatrix();`**

**Remarks:**

Set this matrix to the Identity Matrix.

**Prototype:**

**`void Zero();`**

**Remarks:**

This method sets all elements of the matrix to 0.0f

**Prototype:**

**`Matrix3& Set(const Point3& U, const Point3& V, const Point3&`**

**N, const Point3& T);**

**Remarks:**

This method is available in release 3.0 and later only.

Initializes the matrix with the row data passed and validates the matrix flags.

**Parameters:**

**const Point3& U**

The data for row 0.

**const Point3& V**

The data for row 1.

**const Point3& N**

The data for row 2.

**const Point3& T**

The data for row 3.

**Return Value:**

A reference to this matrix.

**Prototype:**

**int Equals(const Matrix3& M, float epsilon = 1E-6f) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Compares the elements of this matrix and the one specified element by element for equality within the specified tolerance **epsilon**. Returns nonzero if they are 'equal'; otherwise zero.

**Parameters:**

**const Matrix3& M**

The matrix to compare against.

**float epsilon = 1E-6f**

The tolerance for comparison. If the values in the matrix are within this value (+ **epsilon** or - **epsilon**) they are considered equal.

**Prototype:**

**Point3 GetRow(int i) const**

**Remarks:**

Returns the specified row of this matrix.

**Parameters:**

**int i**

Specifies the row to retrieve.

**Prototype:**

**void SetRow(int i, Point3 p);**

**Remarks:**

Sets the specified row of this matrix to the specified values.

**Parameters:**

**int i**

Specifies the row to set.

**Point3 p**

The values to set.

**Prototype:**

**Point4 GetColumn(int i) const;**

**Remarks:**

Returns the 'i-th' column of this matrix.

**Parameters:**

**int i**

Specifies the column to get (0-2).

**Prototype:**

**void SetColumn(int i, Point4 col);**

**Remarks:**

Sets the 'i-th' column of this matrix to the specified values.

**Parameters:**

**int i**

Specifies the column to set (0-2).

### **Point4 col**

The values to set.

#### **Prototype:**

**Point3 GetColumn3(int i) const;**

#### **Remarks:**

Returns the upper three entries in the specified column.

#### **Parameters:**

**int i**

Specifies the partial column to get (0-2).

#### **Prototype:**

**void NoTrans();**

#### **Remarks:**

This method zeros the translation portion of this matrix.

#### **Prototype:**

**void NoRot();**

#### **Remarks:**

This method zeros the rotation portion of this matrix.

#### **Prototype:**

**void NoScale();**

#### **Remarks:**

The method zeros the scale portion of this matrix without orthogonalization. If the matrix was sheared (skewed) then the method is not able to remove scale component completely. Use Orthogonalize() method first, and then NoScale() to remove scale component entirely. Read SCL\_IDENT flag to check if NoScale() method was enough to make the matrix to be orthogonal (with perpendicular axes of unit length).Prototype:

**void Orthogonalize();**

**Remarks:**

This is an "unbiased" orthogonalization of this matrix. The algorithm seems to take a maximum of 4 iterations to converge. An orthogonal matrix has an axis system where each axis is 90 degrees from the others (it's not skewed).

**Prototype:**

**void SetTrans(const Point3 p)**

**Remarks:**

Sets the translation row of this matrix to the specified values. The **POS\_IDENT** flag is cleared.

**Parameters:**

**const Point3 p**

Specifies the values for the translation row.

**Prototype:**

**void SetTrans(int i, float v)**

**Remarks:**

Sets the specified component of the translation row of this matrix to the specified value. The **POS\_IDENT** flag is cleared.

**Parameters:**

**int i**

Specifies the component of the translation row of this matrix to set.

**float v**

The value to set.

**Prototype:**

**Point3 GetTrans() const**

**Remarks:**

Returns the translation row of this matrix.

**Return Value:**

The translation row of this matrix.

**Prototype:**

**void SetScale(const Point3 p);**

**Remarks:**

This method is not currently implemented.

**Prototype:**

**void Translate(const Point3& p);**

**Remarks:**

Apply an incremental translation transformation to this matrix. This is equivalent to multiplying on the RIGHT by the transform.

**Parameters:**

**const Point3& p**

Specifies the translation.

**Prototype:**

**void RotateX(float angle);**

**Remarks:**

Apply an incremental X rotation transformation to this matrix. This is equivalent to multiplying on the RIGHT by the transform.

**Parameters:**

**float angle**

Specifies the X rotation in radians.

**Prototype:**

**void RotateY(float angle);**

**Remarks:**

Apply an incremental Y rotation transformation to this matrix. This is equivalent to multiplying on the RIGHT by the transform.

**Parameters:**

**float angle**

Specifies the Y rotation in radians.

**Prototype:**

**void RotateZ(float angle);**

**Remarks:**

Apply an incremental Z rotation transformation to this matrix. This is equivalent to multiplying on the RIGHT by the transform.

**Parameters:**

**float angle**

Specifies the Z rotation in radians.

**Prototype:**

**void Scale(const Point3& s, BOOL trans = FALSE);**

**Remarks:**

Apply an incremental scaling transformation to this matrix. This is equivalent to multiplying on the RIGHT by the transform.

**Parameters:**

**const Point3& s**

The scale values.

**BOOL trans = FALSE**

If set to **TRUE**, the translation component is scaled. If **trans = FALSE** the translation component is unaffected. When 3ds max was originally written there was a bug in the code for this method where the translation portion of the matrix was not being scaled. This meant that when a matrix was scaled the bottom row was not scaled. Thus it would always scale about the local origin of the object, but it would scale the world axes. When this bug was discovered, dependencies existed in the code upon this bug. Thus it could not simply be fixed because it would break the existing code that depended upon it working the incorrect way. To correct this the **trans** parameter was added. If this is set to **TRUE**, the translation component will be scaled correctly. The existing plug-ins don't use this parameter, it defaults to **FALSE**, and the code behaves the old way.

**Prototype:**

**void PreTranslate(const Point3& p);**

**Remarks:**

Apply an incremental translation transformation to this matrix. This is equivalent to multiplying on the LEFT by the transform.

**Parameters:**

**const Point3& p**

Specifies the translation distance.

**Prototype:**

**void PreRotateX(float angle);**

**Remarks:**

Apply an incremental X rotation transformation to this matrix. This is equivalent to multiplying on the LEFT by the transform.

**Parameters:**

**float angle**

Specifies the X rotation in radians.

**Prototype:**

**void PreRotateY(float angle);**

**Remarks:**

Apply an incremental Y rotation transformation to this matrix. This is equivalent to multiplying on the LEFT by the transform.

**Parameters:**

**float angle**

Specifies the Y rotation in radians.

**Prototype:**

**void PreRotateZ(float angle);**

**Remarks:**

Apply an incremental Z rotation transformation to this matrix. This is equivalent to multiplying on the LEFT by the transform.

**Parameters:**

**float angle**

Specifies the Z rotation in radians.

**Prototype:**

**void PreScale(const Point3& s, BOOL trans = FALSE);**

**Remarks:**

Apply an incremental scaling transformation to this matrix. This is equivalent to multiplying on the LEFT by the transform.

**Parameters:**

**const Point3& s**

The scale values.

**BOOL trans = FALSE**

If trans = FALSE the translation component is unaffected.

**Prototype:**

**void SetTranslate(const Point3& p);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets this matrix to the identity and the translation components to the specified values.

**Parameters:**

**const Point3& p**

The translation values to store.

**Prototype:**

**void SetRotateX(float angle);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets this matrix to the identity and the rotation components to the specified X rotation.

**Parameters:**

**float angle**

The angle for X rotation (in radians).

**Prototype:**

**void SetRotateY(float angle);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets this matrix to the identity and the rotation components to the specified Y rotation.

**Parameters:**

**float angle**

The angle for Y rotation (in radians).

**Prototype:**

**void SetRotateZ(float angle);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets this matrix to the identity and the rotation components to the specified Z rotation.

**Parameters:**

**float angle**

The angle for Z rotation (in radians).

**Prototype:**

**void SetRotate(const Quat& q);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the rotation components of the matrix as specified by the quaternion. The translation and scale components will match the identity matrix.

**Parameters:**

**const Quat& q**

Specifies the rotation to use for the matrix.

**Prototype:**

**void SetRotate(const AngAxis& aa);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the rotation components of the matrix as specified by the AngAxis. The translation and scale components will match the identity matrix.

**Parameters:**

**const AngAxis& aa**

Specifies the rotation to use for the matrix.

**Prototype:**

**void SetRotate(float yaw, float pitch, float roll);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the rotation components of this matrix using yaw, pitch and roll angles.

There are many different conventions for specifying a rotation by means of three Euler angles. This function uses the convention of rotating around the world Z axis, then the X axis, then the Y axis; the three arguments are given in the order Y, X, Z.

This one is equivalent to:

M.IdentityMatrix();

M.RotateZ(roll);

M.RotateX(pitch);

M.RotateY(yaw);

--Which presupposes Y is vertical, X is sideways, Z is forward

**Parameters:**

**float yaw**

The yaw angle in radians.

**float pitch**

The pitch angle in radians.

**float roll**

The roll angle in radians.

**Prototype:**

**void SetAngleAxis();**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the rotation portion of the matrix to the rotation specified by the angle and axis and sets the translation portion to zeros.

**Parameters:**

**const Point3& axis**

The axis of rotation.

**float angle**

The angle of rotation about the axis in radians.

**Prototype:**

**void SetScale(const Point3& s);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the scale components of this matrix to the specified values. The other components to this matrix will match the identity.

**Parameters:**

**const Point3& s**

The scale factors for the matrix.

**Prototype:**

**void SetFromToUp(const Point3& from, const Point3& to, const Point3& up);**

**Remarks:**

This method is available in release 3.0 and later only.

This creates a matrix describing a viewpoint which is at the 'from' location, looking toward the 'to' location; the viewpoint is tilted so that the 'up' vector points to the top of the view.

**Parameters:**

**const Point3& from**

This specifies the viewpoint source location.

**const Point3& to**

This vector specifies the direction of view.

**const Point3& up**

This vector points to the top of the view.

**Prototype:**

**void Invert();**

**Remarks:**

This method is available in release 3.0 and later only.

This method inverts this matrix. An inverted matrix, when multiplied by the original, yields the identity.

**Prototype:**

**BOOL Parity() const;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the 'parity' of the matrix. If one axis of the matrix is scaled negatively this switches the 'parity'. However if you scale two axes it will flip it back.

Three times switches it again.

When rendering a mesh, if you scale something along one axis, it turns 'inside out'. That is the direction when the normals are reversed. This method may be used to detect that case and then reverse the normals. The 3ds max renderer does this -- if this method returns TRUE it flips all the normals so it won't turn inside out.

**Sample Code:**

```
BOOL Matrix3::Parity() const {  
    if (flags&SCL_IDENT) return FALSE;  
    Point3 cp = CrossProd(GetRow(0),GetRow(1));  
    if (DotProd(cp,GetRow(2)) < 0.0) return TRUE;  
    return FALSE;  
}
```

**Prototype:**

**Point3 PointTransform(const Point3& p) const;**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the specified point transformed by this matrix.

**Parameters:**

**const Point3& p**

The point to transform by this matrix.

**Prototype:**

**Point3 VectorTransform(const Point3& p) const;**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the specified vector transformed by this matrix.

**Parameters:**

**const Point3& p**

The vector to transform by this matrix.

**Prototype:**

**void TransformPoints(Point3 \*array, int n, int stride =  
sizeof(Point3));**

**Remarks:**

This method is available in release 3.0 and later only.  
Transforms the specified list of points with this matrix.

**Parameters:**

**Point3 \*array**

The array of points to transform with this matrix.

**int n**

The number of points in the array.

**int stride = sizeof(Point3)**

The size of the increment used when moving to the next point. If you wish to transform an array of data objects which contain x, y, and z coordinates in order (such as a **Point4**, or a structure containing a **Point3** as a member) you can specify a 'stride' value (for instance **sizeof(data\_object)**).

**Prototype:**

```
void TransformPoints(const Point3 *array, Point3 *to, int n, int stride = sizeof(Point3), int strideTo = sizeof(Point3));
```

**Remarks:**

This method is available in release 3.0 and later only.

Transforms the specified list of points with this matrix and stores the resulting transformed points in the storage passed.

**Parameters:**

**const Point3 \*array**

The array of points to transform (the source).

**Point3 \*to**

The array to store the transformed points (the destination).

**int n**

The number of points in the source array.

**int stride = sizeof(Point3)**

**int strideTo = sizeof(Point3)**

The size increment used when moving to the next storage location.

**Prototype:**

```
void TransformVectors(Point3 *array, int n, int stride = sizeof(Point3));
```

**Remarks:**

This method is available in release 3.0 and later only.

Transforms the specified list of vectors with this matrix.

**Parameters:**

**Point3 \*array**

The array of vectors to transform with this matrix.

**int n**

The number of vectors in the array.

**int stride = sizeof(Point3)**

The size of the increment used when moving to the next vector. If you wish to transform an array of data objects which contain x, y, and z coordinates in order (such as a **Point4**, or a structure containing a **Point3** as a member) you

can specify a 'stride' value (for instance **sizeof(data\_object)**).

**Prototype:**

```
void TransformVectors(const Point3 *array, Point3 *to, int n, int stride = sizeof(Point3), int strideTo = sizeof(Point3));
```

**Remarks:**

This method is available in release 3.0 and later only.

Transforms the specified list of vectors with this matrix and stores the resulting transformed vectors in the storage passed.

**Parameters:**

**const Point3 \*array**

The array of vectors to transform (the source).

**Point3 \*to**

The array to store the transformed vectors (the destination).

**int n**

The number of vectors in the source array.

**int stride = sizeof(Point3)**

The size increment used when moving to the next source location.

**int strideTo = sizeof(Point3)**

The size increment used when moving to the next storage location.

**Prototype:**

```
void GetYawPitchRoll(float *yaw, float *pitch, float *roll);
```

**Remarks:**

This method is available in release 3.0 and later only.

Retrieves the yaw, pitch and roll angles represented by the rotation in this matrix.

**Parameters:**

**float \*yaw**

The yaw rotation angle is stored here (in radians).

**float \*pitch**

The pitch rotation angle is stored here (in radians).

**float \*roll**

The roll rotation angle is stored here (in radians).

**Operators:**

**Prototype:**

**int operator==(const Matrix3& M) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Compares the elements of this matrix and the one specified element by element for exact equality. Returns nonzero if they are equal; otherwise zero.

**Parameters:**

**const Matrix3& M**

The matrix to compare against.

**Prototype:**

**Matrix3& operator-=(const Matrix3& M);**

**Remarks:**

Subtracts a **Matrix3** from this **Matrix3**.

**Prototype:**

**Matrix3& operator+=(const Matrix3& M);**

**Remarks:**

Adds a **Matrix3** to this **Matrix3**.

**Prototype:**

**Matrix3& operator\*=(const Matrix3& M);**

**Remarks:**

Multiplies this Matrix3 by the specified Matrix3 (**\*this = (\*this)\*M;**).

**Prototype:**

**Matrix3 operator\*(const Matrix3&) const;**

**Remarks:**

Perform matrix multiplication.

**Prototype:**

**Matrix3 operator+(const Matrix3&) const;**

**Remarks:**

Perform matrix addition.

**Prototype:**

**Matrix3 operator-(const Matrix3&) const;**

**Remarks:**

Perform matrix subtraction.

**Prototype:**

**const Point3& operator[](int i) const;**

**Remarks:**

Returns a reference to the 'i-th' Point3 of the matrix.

The following functions are not methods of **Matrix3** but are available for use:

**Function:**

**Matrix3 RotateXMatrix(float angle);**

**Remarks:**

Builds a new matrix for use as a X rotation transformation.

**Parameters:**

**float angle**

Specifies the angle of rotation in radians.

**Return Value:**

A new X rotation **Matrix3**.

**Function:**

**Matrix3 RotateYMatrix(float angle);**

**Remarks:**

Builds a new matrix for use as a Y rotation transformation.

**Parameters:**

**float angle**

Specifies the angle of rotation in radians.

**Return Value:**

A new Y rotation **Matrix3**.

**Function:**

**Matrix3 RotateZMatrix(float angle);**

**Remarks:**

Builds a new matrix for use as a Z rotation transformation.

**Parameters:**

**float angle**

Specifies the angle of rotation in radians.

**Return Value:**

A new Z rotation **Matrix3**.

**Function:****Matrix3 TransMatrix(const Point3& p);****Remarks:**

Builds a new matrix for use as a translation transformation.

**Parameters:****const Point3& p**

Specifies the translation values.

**Return Value:**

A new translation **Matrix3**.

**Function:****Matrix3 ScaleMatrix(const Point3& s);****Remarks:**

Builds a new matrix for use as a scale transformation.

**Parameters:****const Point3& p**

Specifies the scale values.

**Return Value:**

A new scale **Matrix3**.

**Function:****Matrix3 RotateYPRMatrix(float Yaw, float Pitch, float Roll);****Remarks:**

Builds a new matrix for use as a rotation transformation by specifying yaw, pitch and roll angles.

This definition will depend on what our coordinate system is. This one is equivalent to:

**M.IdentityMatrix();****M.RotateZ(roll);****M.RotateX(pitch);****M.RotateY(yaw);**

Which presupposes Y is vertical, X is sideways, Z is forward

**Parameters:**

**float Yaw**

Specifies the yaw angle in radians.

**float Pitch**

Specifies the pitch angle in radians.

**float Roll**

Specifies the roll angle in radians.

**Return Value:**

A new rotation **Matrix3**.

**Function:**

**Matrix3 RotAngleAxisMatrix(Point3& axis, float angle);**

**Remarks:**

Builds a new matrix for use as a rotation transformation by specifying an angle and axis.

**Parameters:**

**Point3& axis**

Specifies the axis of rotation. Note that this angle is expected to be normalized.

**float angle**

Specifies the angle of rotation. Note: The direction of the angle in this method is opposite of that in **AngAxisFromQ()**.

**Return Value:**

A new rotation **Matrix3**.

**Function:**

**Matrix3 Inverse(const Matrix3& M);**

**Remarks:**

Return the inverse of the matrix

**Parameters:**

**const Matrix3& M**

The matrix to compute the inverse of.

**Function:**

**Point3 VectorTransform(const Matrix3& M, const Point3& V);**

**Remarks:**

Transform the vector (Point3) with the specified matrix.

**Parameters:**

**const Matrix3& A**

The matrix to transform the vector with.

**const Point3& V**

The vector to transform.

**Return Value:**

The transformed vector (as a Point3).

**Function:**

**Matrix3 XFormMat(const Matrix3& xm, const Matrix3& m);**

**Remarks:**

This method is used to build a matrix that constructs a transformation in a particular space. For example, say you have a rotation you want to apply, but you want to perform the rotation in another coordinate system. To do this, you typically transform into the space of the coordinate system, then apply the transformation, and then transform out of that coordinate system. This method constructs a matrix that does just this. It transforms matrix **m** so it is applied in the space of matrix **xm**. It returns a **Matrix3** that is **xm\*m\*Inverse(xm)**.

**Parameters:**

**const Matrix3& xm**

Specifies the coordinate system you want to work in.

**const Matrix3& m**

Specifies the transformation matrix.

**Return Value:**

Returns a **Matrix3** that is **xm\*m\*Inverse(xm)**.

**Function:**

**Point3 operator\*(const Matrix3& A, const Point3& V);**  
**Point3 operator\*(const Point3& V, const Matrix3& A);**

**Remarks:**

These transform a **Point3** with a **Matrix3**. These two versions of transforming a point with a matrix do the same thing, regardless of the order of operands (linear algebra rules notwithstanding).

**Parameters:**

**const Point3& V**

The point to transform.

**const Matrix3& A**

The matrix to transform the point with.

**Return Value:**

The transformed **Point3**.

## Class Quat

See Also: [Class Point3](#), [Class Matrix3](#), [Class AngAxis](#).

class Quat

### Description:

This class provides a compact representation for orientation in three space and provides methods to perform Quaternion algebra.

Quaternions provide an alternative representation for orientation in three-space. To reduce computing, you can substitute quaternion multiplication for rotation-matrix composition.

A quaternion is made up of four terms: a real scalar part which specifies the amount of rotation and an imaginary vector part which defines the axis of rotation. If the quaternion is normalized, the scalar term equals the cosine of half the angle of rotation, the vector term is the axis of rotation, and the magnitude of the vector term equals the sine of half the angle of rotation.

Interpolation between two key frame orientations is much easier using quaternions and produces smooth and natural motion. Unlike Euler angles, no numerical integration is necessary; quaternions provide an analytic result (no approximations).

The rotation convention in the 3ds max API is the left-hand-rule. Note that this is different from the right-hand-rule used in the 3ds max user interface.

For additional information see: Quaternion operations:

From "Quaternion Calculus and Fast Animation",  
by Ken Shoemake, in notes for SIGGRAPH 1987 Course # 10,  
"Computer Animation: 3-D Motion Specification and Control".

All methods of this class are implemented by the system.

### Data Members:

public:

**float x,y,z,w;**

The **x**, **y**, **z** values make up the vector portion. **w** is the angle of rotation about the vector (see remarks above for details).

### Methods:

**Prototype:**

**Quat()**

**Remarks:**

Constructor. No initialization is performed.

**Prototype:**

**Quat(float X, float Y, float Z, float W)**

**Remarks:**

Constructor. The data members are initialized to the values passed.

**Prototype:**

**Quat(double X, double Y, double Z, double W)**

**Remarks:**

Constructor. The data members are initialized to the values passed (cast as floats).

**Prototype:**

**Quat(const Quat& a)**

**Remarks:**

Constructor. The data members are initialized to the Quat passed.

**Prototype:**

**Quat(float af[4])**

**Remarks:**

Constructor. The data members are initialized to the values passed.  
**x = af[0]; y = af[1]; z = af[2]; w = af[3];**

**Prototype:**

**Quat(const Matrix3& mat);**

**Remarks:**

Constructor. Convert the specified 3x3 rotation matrix to a unit quaternion.

**Prototype:**

**Quat(const AngAxis& aa);**

**Remarks:**

Constructor. The Quat is initialized to the AngAxis passed.

**Prototype:**

**Quat(const Point3& V, float W);**

**Remarks:**

This method is available in release 3.0 and later only.

Constructor. The quaternion is initialized from the vector **V** and angle **W** passed. The quaternion is then normalized.

**Prototype:**

**Quat Inverse() const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the inverse of this quaternion (1/q).

**Prototype:**

**Quat Conjugate() const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the conjugate of a quaternion.

**Prototype:**

**Quat LogN() const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the natural logarithm of a UNIT quaternion.

**Prototype:**

## **Quat Exp() const;**

### **Remarks:**

This method is available in release 3.0 and later only.  
Returns the exponentiate quaternion (where  $\mathbf{q.w}=0$ ).

### **Operators:**

### **Prototype:**

**float& operator[](int i)**

### **Remarks:**

Array access operator. Valid **i** values: 0=x, 1=y, 2=z, 3=w.

### **Prototype:**

**const float& operator[](int i) const**

### **Remarks:**

Array access operator. Valid **i** values: 0=x, 1=y, 2=z, 3=w.

### **Prototype:**

**operator float\*()**

### **Remarks:**

Returns the address of the Quaternion.

Unary operators

### **Prototype:**

**Quat operator-() const**

### **Remarks:**

Unary negation. Returns **Quat(-x,-y,-z,-w)**.

### **Prototype:**

**Quat operator+() const**

### **Remarks:**

Unary +. Returns the Quat unaltered.

## Assignment operators

### **Prototype:**

**Quat& operator-=(const Quat&);**

### **Remarks:**

This operator is the same as the /= operator.

### **Prototype:**

**Quat& operator+=(const Quat&);**

### **Remarks:**

This operator is the same as the \*= operator..

### **Prototype:**

**Quat& operator\*=(const Quat&);**

### **Remarks:**

Multiplies this quaternion by a quaternion.

### **Prototype:**

**Quat& operator\*=(float);**

### **Remarks:**

Multiplies this quaternion by a floating point value.

### **Prototype:**

**Quat& operator/=(float);**

### **Remarks:**

Divides this quaternion by a floating point value.

### **Prototype:**

**int operator==(const Quat& a) const;**

### **Remarks:**

Returns nonzero if the quaternions are equal; otherwise 0.

**Prototype:**

**void Identity()**

**Remarks:**

Sets this quaternion to the identity quaternion (**x=y=z=0.0; w=1.0**).

**Prototype:**

**int IsIdentity() const;**

**Remarks:**

Returns nonzero if the quaternion is the identity; otherwise 0.

**Prototype:**

**void Normalize();**

**Remarks:**

Normalizes this quaternion, dividing each term by a scale factor such that the resulting sum of the squares of all parts equals unity.

**Prototype:**

**Quat& MakeClosest(const Quat& qto);**

**Remarks:**

Modifies **q** so it is on same side of hypersphere as **qto**.

**Prototype:**

**void MakeMatrix(Matrix3 &mat, bool b=FALSE) const;**

**Remarks:**

Converts the quaternion to a 3x3 rotation matrix. The quaternion need not be unit magnitude.

**Parameters:**

**Matrix3 &mat**

The matrix.

**BOOL b=FALSE**

This parameter is available in release 4.0 and later only.

When this argument is set to false (or omitted), each function performs as it

did before version 4.0. When the boolean is TRUE, the matrix is made with its terms transposed. When this transposition is specified, **EulerToQuat()** and **QuatToEuler()** are consistent with one another. (In 3ds max 3, they have opposite handedness).

**Prototype:**

**Quat operator+(const Quat&) const;**

**Remarks:**

This operator is the same as the \* operator.

**Prototype:**

**Quat operator-(const Quat&) const;**

**Remarks:**

This operator is the same as the / operator.

**Prototype:**

**Quat operator\*(const Quat&) const;**

**Remarks:**

Returns the product of two quaternions.

**Prototype:**

**Quat operator/(const Quat&) const;**

**Remarks:**

Returns the ratio of two quaternions: This creates a result quaternion  $r = p/q$ , such that  $q*r = p$ . (Order of multiplication is important)

The following operators and functions are not part of class Quat but are available for use.

**Prototype:**

**Quat operator\*(float, const Quat&);**

**Remarks:**

Multiplies the quaternion by a scalar.

**Prototype:**

**Quat operator\*(const Quat&, float);**

**Remarks:**

Multiplies the quaternion by a scalar.

**Prototype:**

**Quat operator/(const Quat&, float);**

**Remarks:**

Divides the quaternion by a scalar.

**Prototype:**

**Quat Inverse(const Quat& q);**

**Remarks:**

Returns the inverse of the quaternion ( $1/\mathbf{q}$ ).

**Prototype:**

**Quat Conjugate(const Quat& q);**

**Remarks:**

Returns the conjugate of a quaternion.

**Prototype:**

**Quat LogN(const Quat& q);**

**Remarks:**

Returns the natural logarithm of UNIT quaternion.

**Prototype:**

**Quat Exp(const Quat& q);**

**Remarks:**

Exponentiate quaternion (where  $\mathbf{q}.\mathbf{w}==0$ ).

**Prototype:**

**Quat Slerp(const Quat& p, const Quat& q, float t);**

**Remarks:**

Spherical linear interpolation of UNIT quaternions.

As t goes from 0 to 1, qt goes from p to q.

$\text{slerp}(p,q,t) = (p*\sin((1-t)*\omega) + q*\sin(t*\omega)) / \sin(\omega)$

**Prototype:**

**Quat LnDif(const Quat& p, const Quat& q);**

**Remarks:**

Computes the "log difference" of two quaternions,  $\mathbf{p}$  and  $\mathbf{q}$ , as  $\ln(\mathbf{q}\text{inv}(\mathbf{p})*\mathbf{q})$ .

**Prototype:**

**Quat QCompA(const Quat& qprev,const Quat& q, const Quat& qnext);**

**Remarks:**

Compute a, the term used in Boehm-type interpolation.

$\mathbf{a}[n] = \mathbf{q}[n]*\mathbf{q}\text{exp}(-(1/4)*(\ln(\mathbf{q}\text{inv}(\mathbf{q}[n])*q[n+1]) + \ln(\mathbf{q}\text{inv}(\mathbf{q}[n])*q[n-1] )))$

**Prototype:**

**Quat Squad(const Quat& p, const Quat& a, const Quat &b, const Quat& q, float t);**

**Remarks:**

$\text{Squad}(p,a,b,q; t) = \text{Slerp}(\text{Slerp}(p,q;t), \text{Slerp}(a,b;t); 2(1-t)t)$ .

**Prototype:**

**Quat qorthog(const Quat& p, const Point3& axis);**

**Remarks:**

Rotate **p** by 90 degrees (quaternion space metric) about the specified **axis**.

**Prototype:**

**Quat squadrev(float angle,const Point3& axis,const Quat& p,  
const Quat& a,const Quat& b,const Quat& q,float t);**

**Remarks:**

Quaternion interpolation for angles > 2PI.

**Parameters:**

**float angle**

Angle of rotation

**const Point3& axis**

The axis of rotation

**const Quat& p**

Start quaternion

**const Quat& a**

Start tangent quaternion

**const Quat& b**

End tangent quaternion

**const Quat& q**

End quaternion

**float t**

Parameter, in range [0.0,1.0]

**Prototype:**

**void RotateMatrix(Matrix3& mat, const Quat& q);**

**Remarks:**

Converts the quaternion to a matrix and multiples it by the specified matrix.  
The result is returned in **mat**.

**Prototype:**

**void PreRotateMatrix(Matrix3& mat, const Quat& q);**

**Remarks:**

Converts the quaternion to a matrix and multiplies it on the left by the specified matrix. . The result is returned in **mat**.

**Prototype:**

**Quat QFromAngAxis(float ang, const Point3& axis);**

**Remarks:**

Converts the [angle,axis] representation to the equivalent quaternion.

**Prototype:**

**void AngAxisFromQ(const Quat& q, float \*ang, Point3& axis);**

**Remarks:**

Converts the quaternion to the equivalent [angle,axis] representation.

**Prototype:**

**float QangAxis(const Quat& p, const Quat& q, Point3& axis);**

**Remarks:**

Compute the [angle,axis] corresponding to the rotation from **p** to **q**. Returns angle, sets axis.

**Prototype:**

**void DecomposeMatrix(const Matrix3& mat, Point3& p,  
Quat& q, Point3& s);**

**Remarks:**

Decomposes a matrix into a rotation, scale, and translation (to be applied in that order). This only will work correctly for scaling which was applied in the rotated axis system. For more general decomposition see the function **decomp\_affine()**. See [Structure AffineParts](#).

**Prototype:**

**Quat TransformQuat(const Matrix3 &m, const Quat&q );**

**Remarks:**

Returns the transformation of the specified quaternion by the specified matrix.

**Prototype:**

**inline Quat IdentQuat()**

**Remarks:**

Returns the identity quaternion (**Quat(0.0,0.0,0.0,1.0)**).

**Function:**

**void QuatToEuler(Quat &q, float \*ang);**

**Remarks:**

Converts the quaternion to Euler angles. When converting a quaternion to Euler angles using this method, the correct order of application of the resulting three rotations is X, then Y, then Z. The angles are returned as **ang[0]=x, ang[1]=y, ang[2]=z**.

**Function:**

**void EulerToQuat(float \*ang, Quat &q, int order);**

**Remarks:**

Converts Euler angles to a quaternion. The angles are specified as **ang[0]=x, ang[1]=y, ang[2]=z**. This method is implemented as:

```
void EulerToQuat(float *ang, Quat &q, int order)  
{  
  Matrix3 mat(1);  
  for (int i=0; i<3; i++) {  
    switch (orderings[order][i]) {  
      case 0: mat.RotateX(ang[i]); break;  
      case 1: mat.RotateY(ang[i]); break;  
      case 2: mat.RotateZ(ang[i]); break;  
    }  
  }
```

```
    q = Quat(mat);  
}
```

**Prototype:**

```
ostream &operator<<(ostream&, const Quat&);
```

**Remarks:**

Output on an ostream.

## Class Color

See Also: [Class AColor](#), [Structure BMM\\_Color\\_24](#), [Structure BMM\\_Color\\_32](#), [Structure BMM\\_Color\\_48](#), [Structure BMM\\_Color\\_64](#), [Structure BMM\\_Color\\_fl](#), [Structure RealPixel](#).

class Color

### Description:

This class represents color as three floating point values: r, g, and b. All methods of this class are implemented by the system.

### Data Members:

public:

**float r,g,b;**

These values are in the range 0.0 to 1.0.

### Methods:

### Prototype:

**Color()**

### Remarks:

Constructor. The resulting object should be initialized with one of the initialization methods.

### Prototype:

**Color(float R, float G, float B)**

### Remarks:

Constructor. Initializes the Color to the RGB color values passed.

### Parameters:

**float R**

Specifies the red component of the color.

**float G**

Specifies the green component of the color.

**float B**

Specifies the blue component of the color.

**Prototype:**

**Color(double R, double G, double B)**

**Remarks:**

Constructor. Initializes the Color to the RGB color values passed.

**Parameters:**

**double R**

Specifies the red component of the color.

**double G**

Specifies the green component of the color.

**double B**

Specifies the blue component of the color.

**Prototype:**

**Color(int R, int G, int B)**

**Remarks:**

Constructor. Initializes the Color to the RGB color values passed.

**Parameters:**

**int R**

Specifies the red component of the color.

**int G**

Specifies the green component of the color.

**int B**

Specifies the blue component of the color.

**Prototype:**

**Color(const Color& a)**

**Remarks:**

Constructor. Initializes the Color to the Color passed.

**Parameters:**

**Color& a**

Specifies the initial color.

**Prototype:**

**Color(DWORD rgb)**

**Remarks:**

Constructor. Initializes the color to the Windows RGB value.

**Parameters:**

**DWORD rgb**

Specifies the initial color via a Windows RGB value.

**Prototype:**

**Color(Point3 p)**

**Remarks:**

Constructor. Initializes the Color to the value of the Point3 passed.

**Parameters:**

**Point3 p**

Specifies the color. r=x, g=y, b=z.

**Prototype:**

**Color(float af[3])**

**Remarks:**

Constructor. Initializes the color to the value passed.

**Parameters:**

**float af[3]**

Specifies the color. r=af[0], g=af[1], b=af[2].

**Prototype:**

**Color(const BMM\_Color\_24& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this Color from the 24 bit color value passed.

**Parameters:**

**const BMM\_Color\_24& c**

The 24 bit color to initialize from.

**Prototype:**

**Color(const BMM\_Color\_32& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this Color from the 32 bit color value passed.

**Parameters:**

**const BMM\_Color\_32& c**

The 32 bit color to initialize from.

**Prototype:**

**Color(const BMM\_Color\_48& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this Color from the 48 bit color value passed.

**Parameters:**

**const BMM\_Color\_48& c**

The 48 bit color to initialize from.

**Prototype:**

**Color(const BMM\_Color\_64& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this Color from the 64 bit color value passed.

**Parameters:**

**const BMM\_Color\_64& c**

The 64 bit color to initialize from.

**Prototype:**

**Color(const BMM\_Color\_fl& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this Color from the floating point color passed.

**Parameters:**

**const BMM\_Color\_fl& c**

The floating point color to initialize from. No conversion or scaling is done.

**Prototype:**

**Color(RealPixel rp);**

**Remarks:**

Constructor. Initializes the color to the RealPixel structure passed.

**Parameters:**

**RealPixel rp**

Specifies the RealPixel format to convert.

**Prototype:**

**void Black()**

**Remarks:**

Sets the Color to black.  $r = g = b = 0.0f$

**Prototype:**

**void White()**

**Remarks:**

Sets the Color to white.  $r = g = b = 1.0f$

**Prototype:**

**void ClampMax()**

**Remarks:**

Makes all the components of the Color  $\leq 1.0$

**Prototype:**

## **void ClampMin()**

### **Remarks:**

Makes all the components of the Color  $\geq 0.0$

### **Prototype:**

**void ClampMinMax()**

### **Remarks:**

Makes all the components fall in the range  $[0,1]$

### **Prototype:**

**int MaxComponent(const Color&)**

### **Remarks:**

Returns the index of the component with the maximum absolute value.

### **Parameters:**

**const Color&**

The color to check.

### **Return Value:**

The index of the component with the maximum absolute value. r=0, g=1, b=2.

### **Prototype:**

**int MinComponent(const Color&)**

### **Remarks:**

Returns the index of the component with the minimum absolute value

### **Parameters:**

**const Color&**

The color to check.

### **Return Value:**

The index of the component with the minimum absolute value. r=0, g=1, b=2.

### **Prototype:**

**float MaxVal(const Color&)**

**Remarks:**

Returns the value of the component with the maximum absolute value.

**Parameters:**

**const Color&**

The color to check.

**Return Value:**

The value of the component with the maximum absolute value.

**Prototype:**

**float MinVal(const Color&)**

**Remarks:**

The value of the component with the minimum absolute value.

**Parameters:**

**const Color&**

The color to check.

**Return Value:****Prototype:**

**inline float Length(const Color& v)**

**Remarks:**

Returns the 'length' of the color, i.e.

```
return (float)sqrt(v.r*v.r+v.g*v.g+v.b*v.b);
```

**Parameters:**

**const Color& v**

The color to return the length of.

**Return Value:**

The length of the color.

**Operators:****Prototype:**

**float& operator[](int i)**

**Remarks:**

Access operator.

**Parameters:**

**int i**

The index of the component to return.

**Return Value:**

0=r, 1=g, 2=b.

**Prototype:**

**const float& operator[](int i) const**

**Remarks:**

Access operator.

**Parameters:**

**int i**

The index of the component to return.

**Return Value:**

0=r, 1=g, 2=b.

**Prototype:**

**operator float\*()**

**Remarks:**

Returns a pointer to the red component of the color. This may be used to treat the Color as an array of three floats.

**Prototype:**

**operator DWORD()**

**Remarks:**

Convert the Color to a Windows RGB color. See [COLORREF](#).

**Prototype:**

**operator Point3()**

**Remarks:**

Convert the Color to a Point3. x=r, y=g, z=b.

**Prototype:**

**operator RealPixel()**

**Remarks:**

Convert the Color to the RealPixel format.

**Prototype:**

**operator BMM\_Color\_24();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this Color to the BMM\_Color\_24 format.

**Prototype:**

**operator BMM\_Color\_32();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this Color to the BMM\_Color\_32 format.

**Prototype:**

**operator BMM\_Color\_48();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this Color to the BMM\_Color\_48 format.

**Prototype:**

**operator BMM\_Color\_64();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this Color to the BMM\_Color\_64 format.

**Prototype:**

**operator BMM\_Color\_fl();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this Color to the BMM\_Color\_fl format.

**Prototype:**

**Color operator-() const**

**Remarks:**

Unary - operator.

**Return Value:**

The Color with the components negated, i.e.

**{ return(Color(-r,-g,-b)); }**

**Prototype:**

**Color operator+() const**

**Remarks:**

Unary + operator.

**Return Value:**

The Color itself.

## Assignment operators.

**Prototype:**

**inline Color& operator-=(const Color&);**

**Remarks:**

Subtracts a Color from this Color.

**Return Value:**

A Color that is the difference between two Colors.

**Prototype:**

**inline Color& operator+=(const Color&);**

**Remarks:**

Adds a Color to this Color.

**Return Value:**

A Color that is the sum of two Colors.

**Prototype:**

**inline Color& operator\*=(float);**

**Remarks:**

Multiplies the components of this Color by a float.

**Return Value:**

A Color multiplied by a float.

**Prototype:**

**inline Color& operator/=(float);**

**Remarks:**

Divides the components of a Color by a float.

**Return Value:**

A Color divided by a float.

**Prototype:**

**inline Color& operator\*=(const Color&);**

**Remarks:**

Performs element-by-element multiplying between two Colors.

**Return Value:**

A Color element-by-element multiplied by another Color.

**Prototype:**

**int operator==(const Color& p) const**

**Remarks:**

Test for equality between two Colors.

**Return Value:**

Nonzero if the Colors are equal; otherwise 0.

**Prototype:**

**int operator!=(const Color& p) const**

**Remarks:**

Tests for inequality between two Colors.

**Return Value:**

Nonzero if the Colors are not equal; otherwise 0.

**Prototype:**

**inline Color operator-(const Color&) const;**

**Remarks:**

Subtracts a Color from a Color.

**Return Value:**

A Color that is the difference between two Colors.

**Prototype:**

**inline Color operator+(const Color&) const;**

**Remarks:**

Adds a Color to a Color.

**Return Value:**

A Color that is the difference between two Colors.

**Prototype:**

**inline Color operator/(const Color&) const;**

**Remarks:**

Divides a Color by a Color.

**Return Value:**

A Color divided by a Color. r/r, g/g, b/b.

**Prototype:**

**inline Color operator\*(const Color&) const;**

**Remarks:**

Multiplies a Color by a Color.

**Return Value:**

A Color multiplied by a Color. r\*r, g\*g, b\*b.

## Class WSMModifier

See Also: [Class Modifier](#), [SimpleWSMMod](#).

class WSMModifier : public Modifier

### Description:

This is a base class for creating world space modifiers. It simply provides a default implementation of **SuperClassID()**.

World Space Modifier plug-ins use a Super Class ID of **WSM\_CLASS\_ID**.

### Methods:

### Prototype:

**SClass\_ID SuperClassID();**

### Remarks:

Implemented by the System.

Returns the Super Class ID of this plug-in type: **WSM\_CLASS\_ID**.

## Class IKeyControl

See Also: [Class Animatable](#), [Class IKey](#), [Class Control](#), [List of SuperClass IDs](#), [Keyframe and Procedural Controller Data Access](#).

class IKeyControl

### Description:

This is an interface into the TCB, Linear, and Bezier keyframe controllers. It allows a developer to add, delete, retrieve and store the keys of the controller. This is for controllers that have made their keys accessible using this interface. 3ds max has done this for its keyframe controllers. It is up to other third party developers to decide if they wish to make their keys available through this interface as well. See below for more details.

It is up to the developer to make sure that the **IKey\*** points to a key of the appropriate derived class based on the **ClassID()** of the controller. For the details of using these APIs see the Advanced Topics section [Keyframe and Procedural Cotroller Data Access](#).

All methods of this class are implemented by the system.

To get a pointer to this interface given a pointer to a controller, use the following macro (defined in **ANIMTBL.H** ). Using this macro, given any Animatable, it is easy to ask for the control interface.

```
#define GetKeyControlInterface(anim)  
((IKeyControl*)anim->GetInterface(I_KEYCONTROL))
```

A plug-in developer may use this macro as follows:

```
IKeyControl *ikc = GetKeyControlInterface(anim);
```

This return value will either be NULL or a pointer to a valid controller interface. Here is an example of getting the controller interface from a node in the scene. First get the position controller from the node (see [Class INode](#)) and then get the controller interface.

```
Control *c;  
c = node->GetTMController()->GetPositionController();  
IKeyControl *ikeys = GetKeyControlInterface(c);
```

With this controller interface you can use its methods to get information about the keys.

```
int num = ikeys->GetNumKeys();
```

Developers should note that the values that are retrieved from this class may differ from the values that appear in Key Info in the 3ds max user interface. For instance, the Intan and Outtan values are multiplied by the global function **GetFrameRate()** when displayed. Additionally, the sign of angles (+ or -) may be reversed from the what is found in the UI. For example, the following shows the values shown in Key Info versus the values retrieved from **GetKey()**:

Motion branch Key Info:

**Key#1**

**X: -1.0**

**Y: 0.0**

**Z: 0.0**

**Ang: 0.0**

**Key#2**

**X: 0.0**

**Y: 1.0**

**Z: 0.0**

**Ang: 90.0**

**Key#3**

**X: 0.0**

**Y: 0.0**

**Z: 1.0**

**Ang: 90.0**

```
ITCBRotKey key;
```

```
ikc->GetKey(i, &key);
```

**Key#1**

**X: 1.0**

**Y: 0.0**

**Z: 0.0**

**Ang: 0.0**

**Key#2**  
**X: 0.0**  
**Y: -1.0**  
**Z: 0.0**  
**Ang: 1.57**

**Key#3**  
**X: 0.0**  
**Y: 0.0**  
**Z: -1.0**  
**Ang: 1.57**

Sample code in the SDK that makes use of this interface is the 3D Studio Export plug-in. See `\MAXSDK\SAMPLES\IMPEXP\3DSEXP.CPP`.

**The following classes are available for keyframe data storage:**

**Tension/Continuity/Bias:**

[Class ITCBKey](#)

[Class ITCBFloatKey](#)

[Class ITCBPoint3Key](#),

[Class ITCBRotKey](#)

[Class ITCBScaleKey](#)

**Bezier:**

[Class IBezFloatKey](#)

[Class IBezPoint3Key](#)

[Class IBezQuatKey](#)

[Class IBezScaleKey](#)

**Linear:**

[Class ILinFloatKey](#)

[Class ILinPoint3Key](#)

[Class ILinRotKey](#)

[Class ILinScaleKey](#)

Note: Developers creating controller plug-ins may wish to make their keys accessible to others through this interface. The way 3ds max does this is by deriving the controllers from this class (**IKeyControl**) in addition to class **Control**. So, multiple inheritance is used, and 3ds max then implements the methods of this class to provide the interface.

Below is the code from 3ds max implementation of **Animatable::GetInterface()** (as part of a template). Note the cast of the **this** pointer to **IKeyControl**.

```
INTERP_CONT_TEMPLATE
void*
InterpControl<INTERP_CONT_PARAMS>::GetInterface(ULC
id)
{
if (id==I_KEYCONTROL) {
return (IKeyControl*)this;
} else {
return Control::GetInterface(id);
}
}
```

**Methods:****Prototype:**

```
virtual int GetNumKeys()=0;
```

**Remarks:**

Returns the total number of keys.

**Prototype:**

```
virtual void SetNumKeys(int n)=0;
```

**Remarks:**

Sets the number of keys allocated. This may add blank keys or delete existing keys. It is more efficient to set a large number of keys using this method and then calling **SetKey()** to store the values rather than calling **AppendKey()** over and over for each key.

**Parameters:****int n**

The new number of keys.

**Prototype:****virtual void GetKey(int i,IKey \*key)=0;****Remarks:**

Retrieves the 'i-th' key and stores the result in key.

**Parameters:****int i**

The index of the key to retrieve.

**IKey \*key**

Storage for the key data.

**Prototype:****virtual void SetKey(int i,IKey \*key)=0;****Remarks:**

Sets the 'i-th' key. The 'i-th' key must exist.

**Parameters:****int i**

The index of the key to set.

**IKey \*key**

Pointer to the key data.

**Prototype:****virtual int AppendKey(IKey \*key)=0;****Remarks:**

This method will append a new key onto the end of the key list. Note that the key list will ultimately be sorted by time.

**Parameters:**

**IKey \*key**

Pointer to the key data to append.

**Return Value:**

The key's index.

**Prototype:**

**virtual void SortKeys()=0;**

**Remarks:**

This method should be called if any changes are made that would require the keys to be sorted. The keys are stored in order by TimeValue.

**Prototype:**

**virtual DWORD &GetTrackFlags()=0;**

**Remarks:**

Retrieves the track flags.

**Return Value:**

One or more of the following values:

**TFLAG\_CURVESEL**

Determines if the curve is selected in the track view in the function curve editor.

**TFLAG\_RANGE\_UNLOCKED**

Determines if the range is locked to the first key and the last key. If a user goes into Position Ranges mode and moves the range bar, the range becomes unlocked.

**TFLAG\_LOOPEDIN**

This is set if the in out of range type is set to loop.

**TFLAG\_LOOPEDOUT**

This is set if the out of range type is set to loop.

**TFLAG\_COLOR**

Set for Bezier Point3 controllers that are color controllers.

## **TFLAG\_HSV**

Set for color controls that interpolate in HSV rather than RGB.

# Class Object

See Also: [Class BaseObject](#), [Class Deformer](#), [Class Interval](#), [Class GraphicsWindow](#), [Template Class Tab](#), [Geometry Pipeline System](#).

```
class Object: public BaseObject , public IXTCAccess
```

## Description:

The object class is the base class for all objects. An object is one of two things: A procedural object or a derived object. Derived objects are part of the system and may not be created by plug-ins. They are containers for modifiers.

Procedural objects can be many different things such as cameras, lights, helper objects, geometric objects, etc. Methods of this class are responsible for things such as allowing the object to be deformed (changing its points), retrieving a deformed bounding box, converting the object between different types (to a mesh or patch for example), texture mapping the object (if appropriate) and interacting with the system regarding mapping. There are other methods involved in validity intervals for the object and its channels, and a method used to return the sub-object selection state of the object.

## Method Groups:

The hyperlinks below jump to the start of groups of related methods within the class:

[Deformable Object Methods](#)

[NURBS Weight Related Methods](#)

[Type Conversion Methods](#)

[Face and Vertex Count Calculations](#)

[Collapse Methods](#)

[Mapping Methods](#)

[Bounding Box Method](#)

[Object Name, Properties, Display, and IntersectRay Methods](#)

[Validity Interval Method](#)

[Modifier Stack Branching](#)

[Particle System Methods](#)

[Parametric Surface Access](#)

[Shapes Within Objects](#)

[Object Integrity Checking](#)

[Sub-Object Selection](#)

[Data Flow Evaluation](#)  
[Extension Channel Access](#)  
[Viewport Rectangle Enlargement](#)

## Methods:

## Deformable Object Methods

### Prototype:

```
virtual int IsDeformable()
```

### Remarks:

Implemented by the Plug-In.

Indicates whether this object is deformable. A deformable object is simply an object with points that can be modified. Deformable objects must implement the generic deformable object methods (**NumPoints()**, **GetPoint(i)**, **SetPoint(i)**, **Deform()**).

A deformable object is simply an object with points that can be modified. These points can be stored in any form the object wants. They are accessed through a virtual array interface with methods to get and set the 'i-th' point. If an object has tangents for instance, it would convert them to and from points as necessary. For example, a simple Bezier spline object that stored its control handles relative to the knot would convert them to be absolute when **GetPoint()** was called with 'i' specifying one of the control points. When the control point is later set, the object can convert it back to be relative to its knot. At this point it could also apply any constraints that it may have, such as maintaining a degree of continuity. The idea is that the entity calling **GetPoint(i)** and **SetPoint(i)** doesn't care what the point represents. It will simply apply some function to the point.

### Return Value:

Return nonzero if the object is deformable and implements the generic deformable object methods; otherwise 0.

### Default Implementation:

```
{ return 0; }
```

Deformable object methods. These only need to be implemented if the object

returns TRUE from the **IsDeformable()** method.

**Prototype:**

**virtual int NumPoints()**

**Remarks:**

Implemented by the Plug-In.

The points of a deformable object are accessed through a virtual array interface. This method specifies the number of points in the object. The meaning of 'points' is defined by the object. A TriObject uses the vertices as the points for example.

**Default Implementation:**

**{ return 0;}**

**Return Value:**

The number of points in the object.

**Prototype:**

**virtual Point3 GetPoint(int i)**

**Remarks:**

Implemented by the Plug-In.

The points of a deformable object are accessed through a virtual array interface. This method returns the 'i-th' point of the object.

Note: If your plug-in is a modifier and you want to operate on the selected points of the object you are modifying, you can't tell which points are selected unless you know the type of object. If it is a generic deformable object there is no way of knowing since the way the object handles selection is up to it.

Therefore, if you want to operate on selected points of a generic deformable object, use a Deformer.

**Parameters:**

**int i**

Specifies which point should be returned.

**Default Implementation:**

**{ return Point3(0,0,0); }**

**Return Value:**

The 'i-th' point of the object.

**Prototype:**

**virtual void SetPoint(int i, const Point3& p)**

**Remarks:**

Implemented by the Plug-In.

The points of a deformable object are accessed through a virtual array interface. This method stores the 'i-th' point of the object.

**Parameters:**

**int i**

The index of the point to store.

**const Point3& p**

The point to store.

**Prototype:**

**virtual BOOL IsPointSelected(int i);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the 'i-th' point is selected; otherwise FALSE.

**Parameters:**

**int i**

The zero based index of the point to check.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual float PointSelection(int i);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a floating point weighted point selection if the object supports it. The default implementation just returns 1.0f if selected and 0.0f if not.

**Parameters:****int i**

The zero based index of the point to check.

**Default Implementation:**

```
{ return IsPointSelected(i) ? 1.0f : 0.0f; }
```

**Prototype:**

```
virtual void PointsWereChanged();
```

**Remarks:**

Implemented by the Plug-In.

Informs the object that its points have been deformed, so it can invalidate its cache. A developer who uses the **GetPoint()** / **SetPoint()** approach to modifying an object will call **PointsWereChanged()** to invalidate the object's cache. For example, if a modifier calls **SetPoint()**, when it is finished it should call this method so the object can invalidate and/or update its bounding box and any other data it might cache.

**Prototype:**

```
virtual void Deform(Deformer *defProc, int useSel=0);
```

**Remarks:**

Implemented by the Plug-In.

This is the method used to deform the object with a deformer. The developer should loop through the object's points calling the **defProc** for each point (or each selected point if **useSel** is nonzero).

The **Deform()** method is mostly a convenience. Modifiers can implement a 'Deformer' callback object which is passed to the **Deform()** method. The object then iterates through its points calling their deformer's callback for each point. The only difference between using the **Deform()** method as opposed to iterating through the points is that the **Deform()** method should respect sub-object selection. For example, the TriObject's implementation of **Deform()** iterates through its vertices, if the TriObject's selection level is set to vertex then it only calls the Deformer's callback for vertices that are selected. This way modifiers can be written that can be applied only to selection sets without

any specific code to check selected points. The default implementation of this method just iterates through all points using **GetPoint(i)** and **SetPoint(i)**. If an object supports sub-object selection sets then it should override this method.

**Parameters:**

**Deformer \*defProc**

A pointer to an instance of the Deformer class. This is the callback object that actually performs the deformation.

**int useSel=0**

A flag to indicate if the object should use the selected points only. If nonzero the selected points are used; otherwise all the points of the object are used.

**Default Implementation:**

```
void Object::Deform(Deformer *defProc,int useSel) {
    int nv = NumPoints();
    for (int i=0; i<nv; i++)
        SetPoint(i,defProc->Map(i,GetPoint(i)));
    PointsWereChanged();
}
```

**Sample Code:**

This code shows the **TriObject** implementation of this method. Note how it looks at the **useSel** parameter to only call the selected points if required.

```
void TriObject::Deform(Deformer *defProc,int useSel) {
    int nv = NumPoints();
    int i;
    if ( useSel ) {
        BitArray sel = mesh.VertexTempSel();
        float *vssel = mesh.getVSelectionWeights ();
        if (vssel) {
            for (i=0; i<nv; i++) {
                if(sel[i]) {
                    SetPoint(i,defProc->Map(i,GetPoint(i)));
                    continue;
                }
                if (vssel[i]==0) continue;
            }
        }
    }
}
```

```

    Point3 & A = GetPoint(i);
    Point3 dir = defProc->Map(i,A) - A;
    SetPoint(i,A+vssel[i]*dir);
}
} else {
    for (i=0; i<nv; i++) if (sel[i]) SetPoint(i,defProc-
>Map(i,GetPoint(i)));
}
} else {
    for (i=0; i<nv; i++)
        SetPoint(i,defProc->Map(i,GetPoint(i)));
}
PointsWereChanged();
}

```

## NURBS Relational Weights

### Prototype:

```
virtual BOOL HasWeights();
```

### Remarks:

This method is available in release 2.0 and later only.

Returns TRUE if the object has weights for its points that can be set; otherwise FALSE.

### Default Implementation:

```
{ return FALSE; }
```

### Prototype:

```
virtual double GetWeight(int i);
```

### Remarks:

This method is available in release 2.0 and later only.

Returns the weight of the specified point of the object.

### Parameters:

```
int i
```

The point to return the weight of.

**Default Implementation:**

```
{ return 1.0; }
```

**Prototype:**

```
virtual void SetWeight(int i, const double w);
```

**Remarks:**

This method is available in release 2.0 and later only.

Sets the weight of the specified point.

**Parameters:**

**int i**

The point whose weight to set.

**const double w**

The value to set.

**Default Implementation:**

```
{ }
```

## **Bounding Box Method**

**Prototype:**

```
virtual void GetDeformBBox(TimeValue t, Box3& box, Matrix3  
*tm=NULL, BOOL useSel=FALSE );
```

**Remarks:**

Implemented by the Plug-In.

This method computes the bounding box in the objects local coordinates or the optional space defined by **tm**.

Note: If you are looking for a precise bounding box, use this method and pass in the node's object TM (**INode::GetObjectTM()**) as the matrix.

**Parameters:**

**TimeValue t**

The time to compute the box.

**Box3& box**

A reference to a box the result is stored in.

**Matrix3 \*tm=NULL**

This is an alternate coordinate system used to compute the box. If the **tm** is not NULL this matrix should be used in the computation of the result.

**BOOL useSel=FALSE**

If TRUE, the bounding box of selected sub-elements should be computed; otherwise the entire object should be used.

## Type Conversion Methods

### Prototype:

**virtual int CanConvertToType(Class\_ID obtype)**

### Remarks:

Implemented by the Plug-In.

Indicates whether the object can be converted to the specified type. If the object returns nonzero to indicate it can be converted to the specified type, it must handle converting to and returning an object of that type from

**ConvertToType()**.

Also see [Class ObjectConverter](#) for additional details on converting objects between types.

### Parameters:

**Class\_ID obtype**

The Class\_ID of the type of object to convert to. See [Class Class\\_ID](#), [List of Class\\_IDs](#).

### Return Value:

Nonzero if the object can be converted to the specified type; otherwise 0.

### Default Implementation:

```
{ return 0; }
```

### Prototype:

**virtual Object\* ConvertToType(TimeValue t, Class\_ID obtype)**

### Remarks:

Implemented by the Plug-In.

This method converts this object to the type specified and returns a pointer to it. Note that if `ConvertToType()` returns a new object it should be a completely different object with no ties (pointers or references) to the original.

Also see [Class ObjectConverter](#) for additional details on converting objects between types.

The following is an issue that developers of world space modifiers need to be aware of if the world space modifier specifies anything but generic deformable objects as its input type. In other words, if a world space modifier, in its implementation of **Modifier::InputType()**, doesn't specifically return **defObjectClassID** then the following issue regarding the 3ds max pipeline needs to be considered. Developers of other plug-ins that don't meet this condition don't need to be concerned with this issue.

World space modifiers that work on anything other than generic deformable objects are responsible for transforming the points of the object they modify into world space using the ObjectState TM. To understand why this is necessary, consider how 3ds max applies the node transformation to the object flowing down the pipeline.

In the geometry pipeline architecture, the node in the scene has its transformation applied to the object in the pipeline at the transition between the last object space modifier and the first world space modifier. The node transformation is what places the object in the scene -- thus this is what puts the object in world space. The system does this by transforming the points of the object in the pipeline by the node transformation. This is only possible however for deformable objects. Deformable objects are those that support the **Object::IsDeformable()**, **NumPoints()**, **GetPoint()** and **SetPoint()** methods. These deformable objects can be deformed by the system using these methods, and thus the system can modify the points to put them in world space itself.

If a world space modifier does not specify that it works on deformable objects, the system is unable to transform the points of the object into world space. What it does instead is apply the transformation to the ObjectState TM. In this case, a world space modifier is responsible for transforming the points of the object into world space itself, and then setting the ObjectState TM to the identity. There is an example of this in the sample code for the Bomb space warp. The Bomb operates on **TriObjects** and implements **InputType()** as `{ return Class_ID(TRIOBJ_CLASS_ID,0); }`. Since

it doesn't specifically return **defObjectClassID**, it is thus responsible for transforming the points of the object into world space itself. It does this in its implementation of **ModifyObject()** as follows:

```
if (os->GetTM()) {  
    Matrix3 tm = *(os->GetTM());  
    for (int i=0; i<triOb->mesh.getNumVerts(); i++) {  
        triOb->mesh.verts[i] = triOb->mesh.verts[i] * tm;  
    }  
    os->obj->UpdateValidity(GEOM_CHAN_NUM,os->tmValid());  
    os->SetTM(NULL,FOREVER);  
}
```

As the code above shows, the Bomb checks if the ObjectState TM is non-NULL. If it is, the points of the object are still not in world space and thus must be transformed. It does this by looping through the points of the **TriObject** and multiplying each point by the ObjectState TM. When it is done, it sets the ObjectState TM to NULL to indicate the points are now in world space. This ensure that any later WSMs will not transform the points with this matrix again.

For the Bomb world space modifier this is not a problem since it specifies in its implementation of **ChannelsChanged()** that it will operate on the geometry channel (**PART\_GEOM**). Certain world space modifiers may not normally specify **PART\_GEOM** in their implementation of **ChannelsChanged()**. Consider the camera mapping world space modifier. Its function is to apply mapping coordinates to the object it is applied to. Thus it would normally only specify **PART\_TEXMAP** for **ChannelsChanged()**. However, since it operates directly on **TriObjects**, just like the Bomb, the system cannot transform the points into world space, and therefore the camera mapping modifier must do so in its implementation of **ModifyObject()**. But since it is actually altering the points of the object by putting them into world space it **is** altering the geometry channel. Therefore, it should really specify **PART\_GEOM | PART\_TEXMAP** in its implementation of **ChannelsChanged()**. If it didn't do this, but went ahead and modified the points of the object anyway, it would be transforming not copies of the points, but the original points stored back in an earlier cache or even the base object.

This is the issue developers need to be aware of. To state this in simple terms then: Any world space modifier that needs to put the points of the object into world space (since it doesn't implement **InputType()** as **defObjectClassID**) needs to specify **PART\_GEOM** in its implementation of **ChannelsChanged()**.

**Parameters:**

**TimeValue t**

The time at which to convert.

**Class\_ID obtype**

The Class\_ID of the type of object to convert to. See [Class Class\\_ID](#), [List of Class IDs](#).

**Return Value:**

A pointer to an object of type **obtype**.

**Default Implementation:**

```
{ return NULL; }
```

**Sample Code:**

The following code shows how a **TriObject** can be retrieved from a node. Note on the code that if you call **ConvertToType()** on an object and it returns a pointer other than itself, you are responsible for deleting that object.

```
// Retrieve the TriObject from the node
int deleteIt;
TriObject *triObject = GetTriObjectFromNode(ip->GetSelNode(0),
    deleteIt);
// Use the TriObject if available
if (!triObject) return;
// ...
// Delete it when done...
if (deleteIt) triObject->DeleteMe();

// Return a pointer to a TriObject given an INode or return NULL
// if the node cannot be converted to a TriObject
TriObject *Utility::GetTriObjectFromNode(INode *node, int
&deleteIt) {
    deleteIt = FALSE;
```

```

Object *obj = node->EvalWorldState(0).obj;
if (obj->CanConvertToType(Class_ID(TRIOBJ_CLASS_ID, 0))) {
    TriObject *tri = (TriObject *) obj->ConvertToType(0,
        Class_ID(TRIOBJ_CLASS_ID, 0));
    // Note that the TriObject should only be deleted
    // if the pointer to it is not equal to the object
    // pointer that called ConvertToType()
    if (obj != tri) deleteIt = TRUE;
    return tri;
}
else {
    return NULL;
}
}

```

## Face and Vertex Count Calculations

### Prototype:

```

virtual BOOL PolygonCount(TimeValue t, int& numFaces, int&
numVerts);

```

### Remarks:

This method is available in release 3.0 and later only.

Retrieves the number of faces and vertices of the polyginal mesh representation of this object. If this method returns FALSE then this functionality is not supported.

Note: Plug-In developers should use the global function

**GetPolygonCount(Object\*, int&, int&)** to retrieve the number of vertices and faces in an arbitrary object.

### Parameters:

#### **TimeValue t**

The time at which to compute the number of faces and vertices.

#### **int& numFaces**

The number of faces is returned here.

#### **int& numVerts**

The number of vertices is returned here.

**Return Value:**

TRUE if the method is fully implemented; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

## Collapse Stack Methods

**Prototype:**

```
virtual Class_ID PreferredCollapseType();
```

**Remarks:**

This method is available in release 2.0 and later only.

This method allows objects to specify the class that is the best class to convert to when the user collapses the stack. The main base classes have default implementations. For example, `GeomObject` specifies `TriObjects` as its preferred collapse type and `shapes` specify `splines` as their preferred collapse type

**Default Implementation:**

```
{return Class_ID(0,0);}
```

**Return Value:**

The `Class_ID` of the preferred object type. See [List of Class IDs](#).

**Prototype:**

```
virtual void GetCollapseTypes(Tab<Class_ID>  
&clist, Tab<TSTR*> &nlist);
```

**Remarks:**

This method is available in release 2.0 and later only.

When the user clicks on the Edit Stack button in the modify branch a list of 'Convert To:' types is presented. The user may click on one of these choices to collapse the object into one of these types (for instance, an Editable Mesh or an Editable NURBS object). This method returns a list of `Class_IDs` and descriptive strings that specify the allowable types of objects that this object may be collapsed into.

Note: Most plug-ins call the base class method in **Object** in their implementation of this method. The base class implementation provided by

**Object** checks if the object can convert to both an editable mesh and an editable spline. If it can, these are added to the allowable types.

**Parameters:**

**Tab<Class\_ID> &clist**

The table of allowable Class\_IDs.

**Tab<TSTR\*> &nlist**

The table of pointers to strings that correspond to the table of Class\_IDs above.

**Sample Code:**

```
void SphereObject::GetCollapseTypes(Tab<Class_ID>
&clist, Tab<TSTR*> &nlist)
{
    Object::GetCollapseTypes(clist, nlist);
    Class_ID id = EDITABLE_SURF_CLASS_ID;
    TSTR *name = new TSTR(GetString(IDS_SM_NURBS_SURFACE));
    clist.Append(1, &id);
    nlist.Append(1, &name);
}
```

**Prototype:**

**virtual Object \*CollapseObject();**

**Remarks:**

This method is available in release 4.0 and later only.

This method is called on the world space cache object when the stack gets collapsed, that lets the pipeline object decide, if it wants to return a different object than itself. The default implementation simply returns this. A PolyObject e.g. can create and return an EditablePolyObject in this method, so that the collapsed object has a UI. I only implemented this method for PolyObject, but this can potentially be implemented that way for all pipeline objects, that currently pass up the editable version.

It is important, that all places, that collapse the stack are calling this method after evaluating the stack.

It also is important, that the editable version implements this method to simply return this, otherwise you'll get a non-editable object when you collapse an

editable polyobject.

**Return Value:**

A pointer to the resulting object.

**Default Implementation:**

```
{ return this;}
```

## Mapping Methods

**Prototype:**

```
virtual int IsMappable()
```

**Remarks:**

This method lets you know if the **ApplyUVWMap()** method is available for this object. This is used by things like the UVW mapping modifier, so that it can determine which objects can have their mapping modified. Returns nonzero if the object is mappable; otherwise zero.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual int NumMapChannels();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the maximum number of channels supported by this type of object.

TriObjects for instance return **MAX\_MESHMAPS** which is currently set to 100.

**Default Implementation:**

```
{ return IsMappable(); }
```

**Prototype:**

```
virtual int NumMapsUsed();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of maps currently used by this object. This is at least 1+ (highest channel in use). This is used so a plug-in that does something to all map channels doesn't always have to do it to every channel up to **MAX\_MESHMAPS** but rather only to this value.

**Default Implementation:**

```
{ return NumMapChannels(); }
```

**Prototype:**

```
void ApplyUVWMap(int type,float utile, float vtile, float wtile,int  
uflip, int vflip, int wflip, int cap,const Matrix3 &tm, int  
channel=1);
```

**Remarks:**

This method may be called to map the object with UVW mapping coordinates. If the object returns nonzero from **IsMappable()** then this method should be implemented.

**Parameters:**

**int type**

The mapping type. One of the following values:

**MAP\_PLANAR**

**MAP\_CYLINDRICAL**

**MAP\_SPHERICAL**

**MAP\_BALL**

**MAP\_BOX**

**float utile**

Number of tiles in the U direction.

**float vtile**

Number of tiles in the V direction.

**float wtile**

Number of tiles in the W direction.

**int uflip**

If nonzero the U values are mirrored.

**int vflip**

If nonzero the V values are mirrored.

**int wflip**

If nonzero the W values are mirrored.

**int cap**

This is used with **MAP\_CYLINDRICAL**. If nonzero, then any face normal that is pointing more vertically than horizontally will be mapped using planar coordinates.

**const Matrix3 &tm**

This defines the mapping space. As each point is mapped, it is multiplied by this matrix, and then it is mapped.

**int channel=1**

This parameter is available in release 2.0 and later only.

This indicates which channel the mapping is applied to. See [List of Mapping Channel Index Values](#).

## Object Name, Properties, Display, and IntersectRay Methods

**Prototype:**

```
virtual int IsRenderable()=0;
```

**Remarks:**

Implemented by the Plug-In.

Indicates whether the object may be rendered. Some objects such as construction grids and helpers should not be rendered and can return zero.

**Return Value:**

Nonzero if the object may be rendered; otherwise 0.

**Prototype:**

```
virtual int IsConstObject()
```

**Remarks:**

Implemented by the Plug-In.

This is called to determine if this is a construction object or not.

**Return Value:**

Nonzero if the object is a construction object; otherwise 0.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual void InitNodeName(TSTR& s)=0;
```

**Remarks:**

Implemented by the Plug-In.

This is the default name of the node when it is created.

**Parameters:**

**TSTR& s**

The default name of the node is stored here.

**Prototype:**

```
virtual int UsesWireColor()
```

**Remarks:**

Implemented by the Plug-In.

This method determines if the object color is used for display.

**Return Value:**

TRUE if the object color is used for display; otherwise FALSE.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual int DoOwnSelectHilite()
```

**Remarks:**

Implemented by the Plug-In.

If an object wants to draw itself in the 3D viewports in its selected state in some custom manner this method should return nonzero. If this item returns nonzero, the **BaseObject::Display()** method should respect the selected state of the object when it draws itself. If this method returns zero the system will use its standard method of showing the object as selected.

**Default Implementation:**

```
{ return 0; }
```

**Return Value:**

Nonzero if the object will draw itself in the selected state; otherwise 0. If nonzero, the plug-in developer is responsible for displaying the object in the selected state as part of its **Display()** method.

**Prototype:**

```
virtual int IntersectRay(TimeValue t, Ray& r, float& at, Point3& norm)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to compute the intersection point and surface normal at this intersection point of the ray passed and the object.

**Parameters:****TimeValue t**

The time to compute the intersection.

**Ray& r**

Ray to intersect. See [Class Ray](#).

**float& at**

The point of intersection.

**Point3& norm**

Surface normal at the point of intersection.

**Default Implementation:**

```
{return FALSE;}
```

**Return Value:**

Nonzero if a point of intersection was found; otherwise 0.

See Also: The [Mesh class](#) implementation of this method.

**Prototype:**

```
virtual BOOL NormalAlignVector(TimeValue t,Point3 &pt, Point3 &norm);
```

**Remarks:**

This method is available in release 2.0 and later only.

Objects that don't support the **IntersectRay()** method (such as helper objects) can implement this method to provide a default vector for use with the normal align command in 3ds max.

**Parameters:**

**TimeValue t**

The time to compute the normal align vector.

**Point3 &pt**

The point of intersection.

**Point3 &norm**

The normal at the point of intersection.

**Return Value:**

TRUE if this method is implemented to return the normal align vector; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

## Validity Interval Methods

**Prototype:**

```
virtual Interval ObjectValidity(TimeValue t)
```

**Remarks:**

Implemented by the Plug-In.

This method returns the validity interval of the object as a whole at the specified time.

**Parameters:**

**TimeValue t**

The time to compute the validity interval.

**Default Implementation:**

```
{ return FOREVER; }
```

**Return Value:**

The validity interval of the object.

**Prototype:**

```
void UpdateValidity(int nchan, Interval v);
```

**Remarks:**

Implemented by the Plug-In.

When a modifier is applied to an object, it needs to include its own validity interval in the interval of the object. To do this, a modifier calls the **UpdateValidity()** method of an object. This method intersects interval **v** to the **nchan** channel validity of the object.

**Parameters:****int nchan**

The validity interval of the modifier is intersected with this channel of the object. See [List of Object Channel Numbers](#).

**Interval v**

The interval to intersect.

## Shapes Within Objects

Shape viewports can reference shapes contained within objects, so the system needs to be able to access the shapes within an object. The following four methods provide this access. These methods are used by the loft object. Since loft objects are made up of shapes, this gives the system the ability to query the object to find out if it is a shape container. Most objects don't contain shapes so they can just use the default implementations.

**Prototype:**

```
virtual int NumberOfContainedShapes();
```

**Remarks:**

Implemented by the Plug-In.

Returns the number of shapes contained inside this object. A shape container may return zero if it doesn't currently have any shapes.

**Return Value:**

The number of shapes. A return value of -1 indicates this is not a container.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

**virtual ShapeObject \*GetContainedShape(TimeValue t, int index)**

**Remarks:**

Implemented by the Plug-In.

This method returns the **ShapeObject** specified by the index passed at the time specified. See [Class ShapeObject](#).

**Parameters:**

**TimeValue t**

The time to return the shape.

**int index**

The index of the shape.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

**virtual void GetContainedShapeMatrix(TimeValue t, int index, Matrix3 &mat)**

**Remarks:**

Implemented by the Plug-In.

Returns the matrix associated with the shape whose index is passed. This matrix contains the offset within the object used to align the shape viewport to the shape.

**Parameters:**

**TimeValue t**

The time to return the matrix.

**int index**

The index of the shape whose matrix to return.

**Matrix3 &mat**

The matrix is returned here.

**Default Implementation:**

```
{ }
```

**Prototype:****virtual BitArray ContainedShapeSelectionArray()****Remarks:**

Implemented by the Plug-In.

This is used by the lofter. The lofter can have several shapes selected, and the bit array returned here will have a bit set for each selected shape. See [Class BitArray](#).

**Return Value:****Default Implementation:****{ return BitArray(); }**

## Object Integrity Checking

**Prototype:****virtual BOOL CheckObjectIntegrity()****Remarks:**

Implemented by the Plug-In.

This method is used for debugging only. The TriObject implements this method by making sure its face's vert indices are all valid.

**Return Value:**

TRUE if valid; otherwise FALSE.

**Default Implementation:****{return TRUE;}**

## Sub-Object Selection

**Prototype:****virtual DWORD GetSubselState()****Remarks:**

Implemented by the Plug-In.

For objects that have sub selection levels, this method returns the current selection level of the object. For example, a TriObject has the following selection levels: object, vertex, face, edge. Other object types may have

different selection levels. The only standard is that a value of 0 indicates object level.

**Default Implementation:**

```
{return 0;}
```

**Return Value:**

The current selection level of the object.

## **Data Flow Evaluation Methods**

Most plug-in procedural objects do not need to be concerned with the following methods associated with locks, channels and shallow copies. The only type of plug-ins that needs to be concerned with these methods are objects that actually flow down the pipeline. Most procedural plug-ins don't go down the pipeline, instead they convert themselves to a TriObject or PatchObject, and these goes down the pipeline. It is these TriObjects or PatchObject that deal with these methods. However plug-in objects that actually flow down the pipeline will use these methods. For more information see the Advanced Topics section on the [Geometry Pipeline System](#).

**Prototype:**

```
virtual ObjectState Eval(TimeValue t)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called to evaluate the object and return the result as an ObjectState. When the system has a pointer to an object it doesn't know if it's a procedural object or a derived object. So it calls **Eval()** on it and gets back an ObjectState. A derived object managed by the system may have to call **Eval()** on its input for example. A plug-in (like a procedural object) typically just returns itself.

A plug-in that does not just return itself is the Morph Object (`\MAXSDK\SAMPLES\OBJECTS\MORPHOBJ.CPP`). This object uses a morph controller to compute a new object and fill in an ObjectState which it returns.

**Parameters:**

**TimeValue t**

Specifies the time to evaluate the object.

**Return Value:**

The result of evaluating the object as an ObjectState.

**Sample Code:**

Typically this method is implemented as follows:

```
{ return ObjectState(this); }
```

**Prototype:**

```
void LockObject()
```

**Remarks:**

Implemented by the System.

This method locks the object as a whole. The object defaults to not modifiable.

**Prototype:**

```
void UnlockObject()
```

**Remarks:**

Implemented by the System.

This method unlocks the object as a whole.

**Prototype:**

```
int IsObjectLocked()
```

**Remarks:**

Implemented by the System.

Returns nonzero if the object is locked; otherwise 0.

**Prototype:**

```
void LockChannels(ChannelMask channels)
```

**Remarks:**

Implemented by the System.

Locks the specified channels of the object.

**Parameters:**

## **ChannelMask channels**

The channels to lock.

### **Prototype:**

**void UnlockChannels(ChannelMask channels)**

### **Remarks:**

Implemented by the System.

Unlocks the specified channel(s) of the object.

### **Parameters:**

**ChannelMask channels**

Specifies the channels to unlock.

### **Prototype:**

**ChannelMask GetChannelLocks()**

### **Remarks:**

Implemented by the System.

Returns the locked status of the channels.

### **Return Value:**

The channels of the object that are locked.

### **Prototype:**

**void SetChannelLocks(ChannelMask channels)**

### **Remarks:**

Implemented by the System.

Sets the locked status of the object's channels.

### **Parameters:**

**ChannelMask channels**

The channel to set to locked.

### **Prototype:**

**ChannelMask GetChannelLocks(ChannelMask m)**

**Remarks:**

Implemented by the System.  
Returns the locked status of the channels.

**Parameters:****ChannelMask m**

Not used.

**Return Value:**

The channels of the object that are locked.

**Prototype:**

```
void CopyChannelLocks(Object *obj, ChannelMask  
needChannels);
```

**Remarks:**

Implemented by the System.  
Copies the specified channels from the object passed.

**Parameters:****Object \*obj**

The source object.

**ChannelMask needChannels**

Indicates the channels to copy.

**Prototype:**

```
virtual Interval ChannelValidity(TimeValue t, int nchan);
```

**Remarks:**

Implemented by the Plug-In.  
Retrieve the current validity interval for the **nchan** channel of the object.  
Note that most procedural objects won't implement this method since they don't have individual channels. Developers wanting to get the validity interval for a procedural object should use **Object::ObjectValidity()** instead.

**Parameters:****TimeValue t**

The time to retrieve the validity interval of the channel.

**int nchan**

Specifies the channel to return the validity interval of. See [List of Object Channel Numbers](#).

**Return Value:**

The validity interval of the specified channel.

**Default Implementation:**

**return FOREVER;**

**Prototype:**

**virtual void SetChannelValidity(int nchan, Interval v);**

**Remarks:**

Implemented by the Plug-In.

Sets the validity interval of the specified channel.

**Parameters:****int nchan**

Specifies the channel. See [List of Object Channel Numbers](#).

**Interval v**

The validity interval for the channel.

**Prototype:**

**virtual void InvalidateChannels(ChannelMask channels)**

**Remarks:**

Implemented by the Plug-In.

This method invalidates the intervals for the given channel mask. This just sets the validity intervals to empty (calling **SetEmpty()** on the interval) .

**Parameters:****ChannelMask channels**

Specifies the channels to invalidate.

**Prototype:**

**void ReadyChannelsForMod(ChannelMask channels);**

**Remarks:**

Implemented by the System.

If the requested channels are locked, this method will replace their data with a copy and unlock them, otherwise it leaves them alone.

**Parameters:****ChannelMask channels**

The channels to ready for modification.

**Prototype:**

**virtual Object \*MakeShallowCopy(ChannelMask channels)**

**Remarks:**

Implemented by the Plug-In.

This method must make a copy of its "shell" and then shallow copy (see below) only the specified channels. It must also copy the validity intervals of the copied channels, and invalidate the other intervals.

**Parameters:****ChannelMask channels**

The channels to copy.

**Return Value:**

A pointer to the shallow copy of the object.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

**virtual void ShallowCopy(Object\* fromOb, ChannelMask channels)**

**Remarks:**

Implemented by the Plug-In.

This method copies the specified channels from the **fromOb** to **this** and copies the validity intervals.

A plug-in needs to copy the specified channels from the specified object **fromOb** to itself by just copying pointers (not actually copying the data). No

new memory is typically allocated, this method is just copying the pointers.

**Parameters:**

**Object\* fromOb**

Object to copy the channels from.

**ChannelMask channels**

Channels to copy.

**Prototype:**

**virtual void NewAndCopyChannels(ChannelMask channels)**

**Remarks:**

Implemented by the Plug-In.

This method replaces the locked channels with newly allocated copies. It will only be called if the channel is locked.

**Parameters:**

**ChannelMask channels**

The channels to be allocate and copy.

**Prototype:**

**virtual void FreeChannels(ChannelMask channels)**

**Remarks:**

Implemented by the Plug-In.

This method deletes the memory associated with the specified channels and set the intervals associated with the channels to invalid (empty).

**Parameters:**

**ChannelMask channels**

Specifies the channels to free.

**Prototype:**

**Interval GetNoEvalInterval()**

**Remarks:**

This method is used internally.

**Prototype:****void SetNoEvalInterval(Interval iv);****Remarks:**

This method is used internally.

**Prototype:****virtual void ReduceCaches(TimeValue t);****Remarks:**

Implemented by the Plug-In.

This method give the object the chance to reduce its caches.

**Parameters:****TimeValue t**

The time to discard any caches the object has.

**Modifier Stack Branching Methods.****Prototype:****virtual int NumPipeBranches(bool selected = true)****Remarks:**

Implemented by the Plug-In.

This method returns the number of pipeline branches combined by the object.

This is not the total number of branches, but rather the number that are active.

For example in the boolean object, if the user does not have any operands selected, this methods would return zero. If they have one selected it would return one.

**Parameters:****bool selected = true**

This parameter is available in release 4.0 and later only.

This parameter must be supported by all compound objects. In case the selected parameter is true the object should only return the number of pipebranches, that are currently selected in the UI (this is the way it worked in R3 and before. In case this parameter is false, the object has to return the number of all branches, no matter if they are selected or not

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual Object *GetPipeBranch(int i, bool selected = true)
```

**Remarks:**

Implemented by the Plug-In.

Retrieves sub-object branches from an object that supports branching. Certain objects combine a series of input objects (pipelines) into a single object. These objects act as a multiplexer allowing the user to decide which branch(s) they want to see the history for.

It is up to the object how they want to let the user choose. For example the object may use sub-object selection to allow the user to pick a set of objects for which the common history will be displayed.

When the history changes for any reason, the object should send a notification (**REFMSG\_BRANCHED\_HISTORY\_CHANGED**) via **NotifyDependents()**.

**Parameters:**

**int i**

The branch index.

**bool selected = true**

This parameter is available in release 4.0 and later only.

This parameter must be supported by all compound objects. In case the selected parameter is true the object should only return the number of pipebranches, that are currently selected in the UI (this is the way it worked in R3 and before. In case this parameter is false, the object has to return the number of all branches, no matter if they are selected or not

**Return Value:**

The 'i-th' sub-object branch.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

**virtual INode \*GetBranchINode(TimeValue t, INode \*node, int i, bool selected = true)**

**Remarks:**

Implemented by the Plug-In.

When an object has sub-object branches, it is likely that the sub-objects are transformed relative to the object. This method gives the object a chance to modify the node's transformation so that operations (like edit modifiers) will work correctly when editing the history of the sub object branch. An object can implement this method by returning a pointer to a new

**INodeTransformed** that is based on the node passed into this method. See [Class INodeTransformed](#).

**Parameters:**

**TimeValue t**

The time to get the INode.

**INode \*node**

The original INode pointer.

**int i**

The branch index.

**bool selected = true**

This parameter is available in release 4.0 and later only.

This parameter must be supported by all compound objects. In case the selected parameter is true the object should only return the number of pipebranches, that are currently selected in the UI (this is the way it worked in R3 and before. In case this parameter is false, the object has to return the number of all branches, no matter if they are selected or not

**Return Value:**

A pointer to an INode. This can be the original passed in (the default implementation does this) or a new **INodeTransformed**.

**Default Implementation:**

```
{return node;}
```

## Particle System Methods

**Prototype:**

## **virtual BOOL CanCacheObject()**

### **Remarks:**

Implemented by the Plug-In.

This method determines if this object can have channels cached. Particle objects flow up the pipeline without making shallow copies of themselves and therefore cannot be cached. Objects other than particle system can just use the default implementation.

### **Return Value:**

TRUE if the object can be cached; otherwise FALSE.

### **Default Implementation:**

```
{return TRUE;}
```

### **Prototype:**

```
virtual void WSStateInvalidate()
```

### **Remarks:**

Implemented by the Plug-In.

This is called by a node when the node's world space state has become invalid. Normally an object does not (and should not) be concerned with this, but in certain cases like particle systems an object is effectively a world space object and needs to be notified.

### **Default Implementation:**

```
{}
```

### **Prototype:**

```
virtual BOOL IsWorldSpaceObject()
```

### **Remarks:**

Implemented by the Plug-In.

Returns TRUE if the object is a world space object; otherwise FALSE. World space objects (particles for example) can not be instanced because they exist in world space not object space. Objects other than particle system can just use the default implementation.

### **Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual Object *FindObject();
```

**Remarks:**

Implemented by the Plug-In.

This method moved from **IDerivedObject** where it was in release 1.x.

It is called to return a pointer to the base object (an object that is not a derived object). This method is overridden by DerivedObjects to search down the pipeline for the base object. The default implementation just returns **this**. This function is still implemented by derived objects and WSM's to search down the pipeline. This allows you to just call it on a nodes ObjectRef without checking for type.

**Default Implementation:**

```
{ return this; }
```

## **Parametric Surface Access**

**Prototype:**

```
virtual BOOL IsParamSurface();
```

**Remarks:**

Implemented by the Plug-In.

This method is available in release 2.0 and later only.

There are several methods used to access a parametric position on the surface of the object. If this method returns TRUE then

**Object::GetSurfacePoint()** will be called to return a point on the surface that corresponds to the **u** and **v** parameters passed to it. If this method returns FALSE then it is assumed the object does not support returning a point on the surface based on parameteric values. For sample code see

**\MAXSDK\SAMPLES\OBJECTS\SPHERE.CPP**. If the object has several parametric surfaces then a second version of **GetSurfacePoint()** with an integer which specifies which surface will be called.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual int NumSurfaces(TimeValue t);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of parametric surfaces within the object. Prior to release 3.0 only one parametric object could be accessed.

**Parameters:**

**TimeValue t**

The time at which to check.

**Default Implementation:**

**{return 1;}**

**Prototype:**

**virtual Point3 GetSurfacePoint(TimeValue t, float u, float v,Interval &iv);**

**Remarks:**

Implemented by the Plug-In.

This method is available in release 2.0 and later only.

This method needs to be implemented if **Object::IsParamSurface()** returns TRUE. This method is used to retrieve a point on the surface of the object based on two parameters of the surface, **u** and **v**.

Note: This method assumes there is a single parametric surface. If there is more than 1 (**NumSurfaces()** returns > 1, use the **GetSurface()** method below which allows for multiple surfaces.

**Parameters:**

**TimeValue t**

The time to retrieve the point.

**float u**

The parameter along the horizontal axis of the surface.

**float v**

The parameter along the vertical axis of the surface.

**Interval &iv**

This interval is updated based on the interval of the surface parameter.

**Default Implementation:**

```
{return Point3(0,0,0);}
```

**Prototype:**

```
virtual Point3 GetSurfacePoint(TimeValue t, int surface, float u,  
float v,Interval &iv);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to retrieve a point on the specified surface of the object based on two parameters of the surface, **u** and **v**.

**Parameters:**

**TimeValue t**

The time to retrieve the point.

**int surface**

The zero based index of the surface. This number is  $\geq 0$  and  $< \text{NumSurfaces}()$ .

**float u**

The parameter along the horizontal axis of the surface.

**float v**

The parameter along the vertical axis of the surface.

**Interval &iv**

This interval is updated based on the interval of the surface parameter.

**Default Implementation:**

```
{return Point3(0,0,0);}
```

**Prototype:**

```
virtual void SurfaceClosed(TimeValue t, int surface, BOOL  
&uClosed, BOOL &vClosed);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method allows the object to return flags that indicate whether the parametric surface is closed in the U and V dimensions. Set the appropriate closure variables to TRUE if the surface is closed in that direction, FALSE if it is not. A torus, for example, is closed in both directions.

**Parameters:**

**TimeValue t**

The time to check the surface.

**int surface**

The zero based index of the surface. This number is  $\geq 0$  and  $< \text{NumSurfaces}()$ .

**BOOL &uClosed**

Set to TRUE if the surface is closed in U; otherwise to FALSE.

**BOOL &vClosed**

Set to TRUE if the surface is closed in V; otherwise to FALSE.

**Default Implementation:**

**{uClosed = vClosed = TRUE;}**

## **Viewport Rectangle Enlargement**

**Prototype:**

**virtual void MaybeEnlargeViewportRect(GraphicsWindow \*gw, Rect &rect);**

**Remarks:**

This method is available in release 3.0 and later only.

This method allows the object to enlarge its viewport rectangle, if it wants to. The system will call this method for all objects when calculating the viewport rectangle; the object can enlarge the rectangle if desired. This is used by the Editable Spline code to allow extra room for vertex serial numbers, which can extend outside the normal bounding rectangle.

**Parameters:**

**GraphicsWindow \*gw**

Points to the GraphicsWindow associated with the viewport.

**Rect &rect**

The enlarged rectangle is returned here.

**Default Implementation:**

```
{
```

**Sample Code:**

```
void SplineShape::MaybeEnlargeViewportRect(GraphicsWindow
*gw, Rect &rect) {
    if(!showVertNumbers)
        return;
    TCHAR dummy[256];
    SIZE size;
    int maxverts = -1;
    for(int i = 0; i < shape.splineCount; ++i) {
        int verts = shape.splines[i]->KnotCount();
        if(verts > maxverts)
            maxverts = verts;
    }
    sprintf(dummy,"%d",maxverts);
    gw->getTextExtents(dummy, &size);
    rect.SetW(rect.w() + size.cx);
    rect.SetY(rect.y() - size.cy);
    rect.SetH(rect.h() + size.cy);
}
```

**Prototype:**

```
virtual BOOL GetExtendedProperties(TimeValue t, TSTR
&prop1Label, TSTR &prop1Data, TSTR &prop2Label, TSTR
&prop2Data);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method allows an object to return extended Properties fields. It is called when the Object Properties dialog is being prepared. If you don't want to

display any extended properties, simply return FALSE.

To display extended property fields, place the field label in the appropriate label string and the display value in a formatted string. Two fields are supplied, each with a label and a data string; if only using one, make the second label field and data field blank (""). Return TRUE to indicate you have filled in the fields. The properties dialog will display your returned values.

**Parameters:**

**TimeValue t**

The time at which the strings are requested.

**TSTR &prop1Label**

The string for the property 1 label.

**TSTR &prop1Data**

The formatted data string to appear as property 1.

**TSTR &prop2Label**

The string for the property 2 label.

**TSTR &prop2Data**

The formatted data string to appear as property 2.

**Return Value:**

TRUE if this method is implemented and the fields are filled in; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**ExtensionChannel Access :**

**Prototype:**

```
void AddXTCObject(XTCObject *pObj, int priority = 0, int  
branchID = -1);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method adds an extension object into the pipeline.

Implemented by the System.

**Parameters:**

**XTCObject \*pObj**

Points to the extension object to add.

**int priority = 0**

The priority of the object. The methods (XTCObject::Display(), PreChanChangedNotify() etc) of higher priority XTCObjects will be called before those of lower priority XTCObjects.

**int branchID = -1**

The branch identifier of the object.

**Prototype:****int NumXTCObjects();****Remarks:**

This method is available in release 4.0 and later only.

Returns the number of extension objects maintained by this Object.

Implemented by the System.

**Prototype:****XTCObject \*GetXTCObject(int index);****Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the specified extension object.

Implemented by the System.

**Parameters:****int index**

The zero based index of the extension object to return.

**Prototype:****void RemoveXTCObject(int index);****Remarks:**

This method is available in release 4.0 and later only.

Removes the extension object as indicated by the index.

Implemented by the System.

**Parameters:****int index**

The zero based index of the extension object to remove.

**Prototype:**

```
void SetXTCObjectPriority(int index,int priority);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the priority for the extension object whose index is passed.

Implemented by the System.

**Parameters:****int index**

The zero based index of the extension object to remove.

**int priority**

The new priority to assign.

**Prototype:**

```
int GetXTCObjectPriority(int index);
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns the integer priority number of the extension object whose index is passed.

Implemented by the System.

**Parameters:****int index**

The zero based index of the extension object to check.

**Prototype:**

```
void SetXTCObjectBranchID(int index,int branchID);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the branch ID of the extension object whose index is passed.

Implemented by the System.

**Parameters:**

**int index**

The zero based index of the extension object whose branch ID is set.

**int branchID**

The branch identifier to set.

**Prototype:**

**int GetXTCObjectBranchID(int index);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns the integer branch ID of the extension object whose index is passed.

Implemented by the System.

**Parameters:**

**int index**

The zero based index of the extension object whose branch ID is to be returned.

**Prototype:**

**void MergeAdditionalChannels(Object \*from, int branchID);**

**Remarks:**

This method is available in release 4.0 and later only.

This method has to be called whenever the CompoundObject updates a branch (calling **Eval()** on it). Object **\*from** is the object returned from

**Eval(os.obj)**. The branchID is an integer that specifies that branch. The extension channel will get a callback to

**XTCObject::RemoveXTCObjectOnMergeBranches()** and

**XTCObject::MergeXTCObject()**. By default it returns true to

**RemoveXTCObjectOnMergeBranches** which means that the existing XTCObjects with that branchID will be deleted. The method

**MergeXTCObject** simply copies the XTCObjects from the incoming branch into the compound object.

Implemented by the System.

**Parameters:**

**Object \*from**

The object to merge additional channels from.

**int branchID**

The branch identifier to set.

**Prototype:**

**void BranchDeleted(int branchID, bool reorderChannels);**

**Remarks:**

This method is available in release 4.0 and later only.

This method has to be called on the CompoundObject so it can delete the XTCTObjects for the specified branch. The XTCTObject will again have the final decision if the XTCTObject gets really deleted or not in a callback to **XTCTObject::RemoveXTCTObjectOnBranchDeleted()** which will return true if the XTCTObject should be removed.

Implemented by the System.

**Parameters:**

**int branchID**

Specifies which branch of the compound object the extension objects are deleted from.

**bool reorderChannels**

TRUE to reorder the channels, otherwise FALSE.

**Prototype:**

**void CopyAdditionalChannels(Object \*from, bool deleteOld = true, bool bShallowCopy = false);**

**Remarks:**

This method is available in release 4.0 and later only.

This method copies all extension objects from the "from" object into the current object. In case deleteOld is false the objects will be appended. If it is true the old XTCTObjects will be deleted.

Implemented by the System.

**Parameters:**

**Object \*from**

The source object which contains extension objects.

**bool deleteOld = true**

If true the original objects are deleted after the copy; if false they remain after the copy.

**bool bShallowCopy = false**

If true only a ShallowCopy() is performed; if false a complete copy of the objects is done.

**Prototype:**

```
void DeleteAllAdditionalChannels();
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method will delete all additional channels.

**The following function is not part of this class but is available for use:**

**Function:**

```
void GetPolygonCount(TimeValue t, Object* pObj, int&  
numFaces, int& numVerts);
```

**Remarks:**

This global function is available in release 3.0 and later only.

This global function (not part of class Object) may be used to count the number of faces and vertices in an object. It uses **Object::PolygonCount()** if it is supported, and converts to **TriObject** and counts faces and vertices otherwise.

**Parameters:****TimeValue t**

The time at which to compute the number of faces and vertices.

**Object\* pObj**

Points to the object to check.

**int& numFaces**

The number of faces is returned here.

**int& numVerts**

The number of vertices is returned here.

# Class Mesh

See Also: [Class BitArray](#), [Class Face](#), [Class TVFace](#), [Class Point3](#), [Class TriObject](#), [Class RNormal](#), [Class RVertex](#), [Class TriObject](#), [Class MeshMap](#), [Working with Meshes](#), [Class PerData](#).

```
class Mesh : public BaseInterfaceServer
```

## Description:

The Mesh class is provided for use by plug-ins and is used by the system. It is the data structure for meshes in the system and maintains pointers to the vertices, faces, texture vertices, etc. It provides methods to handle most of the functionality that procedural objects must implement. All methods of this class are implemented by the system.

Note: There is a set of classes for working with parts of a mesh such as its face structure, element structure, and cluster structure. For details see: [Class AdjEdgeList](#), [Class AdjFaceList](#), [Class FaceElementList](#), [Class FaceClusterList](#).

## Method Groups:

The hyperlinks below take you to the start of groups of related methods within the class:

[Vertex Methods](#)

[Color Per Vertex Methods](#)

[Face Methods](#)

[Texture Vertex-Texture Face Methods](#)

[Multiple Map Support](#)

[Vertex Data Methods](#)

[Render, HitTest, Snap Methods](#)

[Bounding Box Methods](#)

[Edge Methods](#)

[Strips Methods](#)

[Mesh / Display Flags](#)

[Selection Access Methods](#)

[Data Flow Evaluation Methods](#)

[IntersectRay, Weld, Optimize, Apply Mapping Methods](#)

[Normals / Smoothing / Edges / Face check Functions](#)

[Boolean Operations / Combine Meshes](#)

[Sub-Object Selection Color Control](#)

## [Operators](#)

### **Data Members:**

private:

**MeshMap hmaps[NUM\_HIDDENMAPS];**

The "Hidden Maps". Eventually these will be supported by Patches and PolyMeshes, as well as in the general object interface. The purpose is to use a separate, private array for a fixed number of these maps (unlike the allocable regular map array). This allows us to reserve and support as many map channels as we need for features such as vertex alpha, vertex illumination, possible vertex normals, or other needs that may arise.

The interface for accessing these channels uses the usual map methods, with negative indices. For instance, **hmaps[0]** is indexed as

**MAP\_SHADING**, -1. **hmaps[1]** is indexed as **MAP\_ALPHA=-2**. So now methods like **Mesh::mapVerts (int mp)** have a range of -**NUM\_HIDDENMAPS** to **getNumMaps()**, instead of 0 to **getNumMaps()** as before.

**NUM\_HIDDENMAPS**: The number of "Hidden" or negative-indexed maps in objects which support hidden maps. (Currently set to 2.)

**MAP\_SHADING**: The shading (or illumination) map. Set to -1.

**MAP\_ALPHA**: The Alpha channel map. Note that only the x (or u) coordinate of the map vertices is currently used. Set to -2.

public:

## **Topology Data**

**int numVerts;**

Number of vertices.

**int numFaces;**

Number of faces.

**Face \*faces;**

Array of faces.

## Geometry Data

**Point3 \*verts;**

Array of vertex coordinates.

## Texture Coordinate Data

### **int numTVerts;**

Number of texture vertices.

### **UVVert \*tVerts;**

The array of texture vertices. This stores the UVW coordinates. For a 2D mapping only two of them are used, i.e. UV, VW, or WU. This just provides greater flexibility so the user can choose to use UV, VW, or WU.

Note: **typedef Point3 UVVert;**

### **TVFace \*tvFace;**

The array of texture faces. There needs to be one **TVFace** for every face, but there can be three indices into the UVVert array that are any UVs. Each face of the object can have its own mapping.

### **int numMaps;**

This data member is available in release 3.0 and later only.

The number of maps supported by the mesh. By default this is 2 but may be changed with the multiple map methods listed below.

### **MeshMap \*maps;**

This data member is available in release 3.0 and later only.

When the number of mapping channels is set to a value greater than 1 then an instance of this class is allocated for each channel up to **numMaps**. An instance maintains the mapping information for a single channel.

### **BitArray vdSupport;**

This data member is available in release 3.0 and later only.

This bit array indicates if a particular vertex data channel is supported in this mesh. If the bit is set the channel is supported.

### **PerData \*vData;**

This data member is available in release 3.0 and later only.

The array of PerData objects which maintain and provide access to the floating point vertex data. There is one of these for each supported channel. The first two PerData objects in this array are used internally by 3ds max.

## Color Per Vertex Data

**int numCVerts;**

Number of color vertices.

**VertColor \*vertCol;**

Array of color vertices.

**TVFace \*vcFace;**

Array of color per vertex faces.

**VertColor \*curVCArray;**

This data member is available in release 4.0 and later only.

Points to storage for a possible external color array (default = NULL). This can be either an external array or one of the mapping channels. See the method **Mesh::setVCDisplayData()**;

**int curVCChan;**

This data member is available in release 4.0 and later only.

Storage for the current mapping channel to use for vertex colors (default = 0).

**VertColor \*vertColArray;**

This data member is available in release 4.0 and later only.

When 3ds max is rendering the color values come from this variable. This array defaults to the internal **vertCol** but can be set to an external array, or a map channel. See the method **Mesh:: setVCDisplayData ()**;

**TVFace \*vcFaceData;**

This data member is available in release 4.0 and later only.

When 3ds max is rendering the vertex color lookup comes from this structure. This defaults to the **vcFace** data but if a mapping channel is used for color lookup, its **TVFace** structure is used.

## Selection Data

### **BitArray vertSel;**

Indicates the selected vertices. There is one bit for each vertex. Bits that are 1 indicate the vertex is selected.

### **BitArray faceSel;**

Indicates the selected faces. There is one bit for each face. Bits that are 1 indicate the face is selected.

### **BitArray edgeSel;**

Indicates the selected edges. There is one bit for each edge of each face. Bits that are 1 indicate the edge is selected. The edge is identified by  **$3*\text{faceIndex} + \text{edgeIndex}$** .

### **BitArray vertHide;**

Hidden flags for vertices.

## Display attribute flags

### **DWORD dispFlags;**

These control various aspect of the Mesh objects display and may be one or more of the following values:

**DISP\_VERTTICKS** - Display vertices as small tick marks.

**DISP\_SELVERTS** - Display selected vertices.

**DISP\_SELFACES** - Display selected faces.

**DISP\_SELEDGES** - Display selected edges.

**DISP\_SELPOLYS** - Display selected polygons. Polygons are defined as adjacent triangles with hidden edges. A selected face would show all edges regardless of if they were hidden edges. A polygon would only show the edges of the polygon that were not hidden.

## **Selection level**

### **DWORD selLevel;**

This is the current level of selection. When all the bits are 0, the object is at object level selection. The selection level bits are:

**MESH\_OBJECT** - Object level.

**MESH\_VERTEX** - Vertex level.

**MESH\_FACE** - Face level.

**MESH\_EDGE** - Edge level.

## **Normals**

**int normalsBuilt;**

Nonzero if normals have been built for the current mesh; otherwise 0.

## Render Data

**MeshRenderData \*renderData;**

Points to the render data used by the renderer.

## Methods:

### Prototype:

**Mesh();**

### Remarks:

Constructor. Initializes the mesh object. The mesh counts are set to 0 and its pointers are set to NULL.

### Prototype:

**Mesh(const Mesh& fromMesh);**

### Remarks:

Constructor. The mesh is initialized equal to **fromMesh**.

### Prototype:

**~Mesh();**

### Remarks:

Destructor. Frees any allocated arrays (faces, verts, tverts, tvfaces).

### Prototype:

**void Init();**

### Remarks:

Initializes the mesh object. The mesh counts are set to 0 and its pointers are set to NULL. Note: This method is not intended to be called by developers. It is used internally.

### Prototype:

**void DeleteThis();**

### Remarks:

This method is available in release 3.0 and later only.

This method deletes this mesh.

## Vertex Methods

### Prototype:

```
BOOL setNumVerts(int ct, BOOL keep=FALSE, BOOL  
synchSel=TRUE);
```

### Remarks:

Sets the number of geometric vertices in the mesh.

### Parameters:

#### **int ct**

Specifies the number of vertices.

#### **BOOL keep=FALSE**

Specifies if the previous vertices should be kept. If TRUE the previous vertices are kept; otherwise they are discarded.

#### **BOOL synchSel=TRUE**

This parameter is available in release 2.0 and later only.

If TRUE the selection set BitArrays are resized to fit the number of vertices; otherwise they are left unaltered.

### Return Value:

TRUE if storage was allocated and the number of vertices was set; otherwise FALSE.

### Prototype:

```
int getNumVerts() const
```

### Remarks:

Returns the number of vertices.

### Prototype:

```
void setVert(int i, const Point3 &xyz);
```

### Remarks:

Sets a single vertex in the **verts** array.

**Parameters:**

**int i**

A zero based index into the **verts** array of the vertex to store.

**const Point3 &xyz**

Specifies the coordinate of the vertex.

**Prototype:**

**void setVert(int i, float x, float y, float z);**

**Remarks:**

Sets a single vertex in the **verts** array.

**Parameters:**

**int i**

A zero based index into the **verts** array of the vertex to store.

**float x**

Specifies the X coordinate of the vertex.

**float y**

Specifies the Y coordinate of the vertex.

**float z**

Specifies the Z coordinate of the vertex.

**Prototype:**

**Point3& getVert(int i);**

**Remarks:**

Returns the 'i-th' vertex.

**Parameters:**

**int i**

Specifies the index of the vertex to retrieve.

**Prototype:**

**Point3\* getVertPtr(int i);**

**Remarks:**

Returns a pointer to the 'i-th' vertex.

**Parameters:**

**int i**

Specifies the index of the vertex address to retrieve.

## **Color Per Vertex Methods**

**Prototype:**

**BOOL setNumVertCol(int ct, BOOL keep=FALSE);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the number of color per vertex vertices.

**Parameters:**

**int ct**

The number of color vertices to set.

**BOOL keep=FALSE**

If TRUE previous values are kept; otherwise they are discarded.

**Return Value:**

TRUE if the value was set; otherwise FALSE.

**Prototype:**

**int getNumVertCol() const;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the number of color per vertex vertices.

**Prototype:**

**BOOL setNumVCFaces(int ct, BOOL keep=FALSE, int oldCt=0);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the number of color per vertex faces.

**Parameters:**

**int ct;**

The number of color per vertex faces to set.

**BOOL keep=FALSE**

Specifies if the old faces should be kept if the array is being resized. If FALSE they are freed.

**int oldCt=0**

The length of the existing VCFaces array.

**Return Value:**

TRUE if storage has been allocated and the number is set; otherwise FALSE.

**Prototype:**

```
void setVCDisplayData(int mapChan = 0, VertColor  
*VCArray=NULL, TVFace *VCf=NULL);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method would typically be called right before display, as with a node display callback, or through an extension object. If mapChan parameter is set to MESH\_USE\_EXT\_CVARRAY then the data in VCArray and TVFace is stored for internal use and consequent drawing. If the arrays are NULL then the internal source is used.

**Parameters:**

**int mapChan**

the mapping channel to use.

**VertColor \* VCArray**

An external array hosting the vertex colors

**TVFace \* vcf**

An external array of TVFace indexing into the color array

**Prototype:**

```
void setSmoothFlags(int f);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method should be called when the user has clicked on the 'Smooth' check box in a procedural object. It invalidates the appropriate caches of the mesh so the display is updated properly. If this method is not called, the internal topology cache might prevent the mesh from appearing changed.

**Parameters:**

**int f**

Nonzero indicates smoothed; zero unsmoothed.

**Prototype:**

**int getSmoothFlags();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the state of the smooth flags. See **setSmoothFlags()** above.

**Face Methods****Prototype:**

**BOOL setNumFaces(int ct, BOOL keep=FALSE, BOOL synchSel=TRUE);**

**Remarks:**

Sets the number of faces in the mesh.

**Parameters:**

**int ct**

Specifies the number of faces.

**BOOL keep=FALSE**

Specifies if the previous faces should be kept. If TRUE the previous faces are kept; otherwise they are discarded.

**BOOL synchSel=TRUE**

This parameter is available in release 2.0 and later only.

If TRUE the selection set BitArrays are resized to fit the number of faces; otherwise they are left unaltered.

**Return Value:**

TRUE if storage was allocated and the number of faces was set; otherwise FALSE.

**Prototype:**

```
int getNumFaces() const
```

**Remarks:**

Returns the number of faces in the mesh.

**Texture Vertex-Texture Face Methods****Prototype:**

```
BOOL setNumTVerts(int ct, BOOL keep=FALSE);
```

**Remarks:**

Sets the number of texture vertices (in mapping channel 1).

**Parameters:**

**int ct**

Specifies the number of texture vertices.

**BOOL keep=FALSE**

Specifies if the previous texture vertices should be kept. If TRUE the previous texture vertices are kept; otherwise they are discarded.

**Return Value:**

TRUE if storage was allocated and the number of texture vertices was set; otherwise FALSE.

**Prototype:**

```
int getNumTVerts() const
```

**Remarks:**

Returns the number of texture vertices (in mapping channel 1).

**Prototype:**

```
BOOL setNumTVFaces(int ct, BOOL keep=FALSE, int oldCt=0);
```

**Remarks:**

Sets the number of TVFaces. This method is automatically called if you set the number of faces to keep these two in sync (because the number of TVFaces should be the same as the number of faces). The following rules apply:

If you have no TVFaces and **keep** is TRUE then the TVFaces array stays empty.

If you have no TVFaces and **keep** is FALSE they are allocated.

If you have TVFaces and ct = 0 then the TVFaces are freed.

**Parameters:****int ct**

The number of TVFaces.

**BOOL keep=FALSE**

Specifies if the old faces should be kept.

**int oldCt=0**

The length of the existing TVFaces array.

**Return Value:**

TRUE if storage has been allocated and the number is set; otherwise FALSE.

**Prototype:****void setTVert(int i, const UVVert &xyz);****Remarks:**

Sets a single texture vertex in the **tVerts** array.

**Parameters:****int i**

A zero based index into the **tVerts** array of the texture vertex to store.

**const UVVert &xyz**

Specifies the coordinate of the vertex.

**Prototype:****void setTVert(int i, float x, float y, float z);****Remarks:**

Sets a single texture vertex in the **tVerts** array.

**Parameters:**

**int i**

A zero based index into the **tVerts** array of the texture vertex to store.

**float x**

Specifies the X coordinate of the texture vertex.

**float y**

Specifies the Y coordinate of the texture vertex.

**float z**

Specifies the Z coordinate of the texture vertex.

**Prototype:**

**UVVert& getTVert(int i)**

**Remarks:**

Returns the 'i-th' texture vertex.

**Parameters:**

**int i**

Specifies the index of the texture vertex to retrieve.

**Prototype:**

**UVVert\* getTVertPtr(int i)**

**Remarks:**

Returns a pointer to the 'i-th' texture vertex.

**Parameters:**

**int i**

Specifies the index of the texture vertex address to retrieve.

## **Multiple Map Support**

**Prototype:**

**void setNumMaps(int ct, BOOL keep=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

Set the number of texture maps used by this Mesh. Note that this call is made automatically if **Mesh::setMapSupport()** is called.

**Parameters:**

**int ct**

The number of texture maps to use. This is a value between 2 and **MAX\_MESHMAPS-1**.

**BOOL keep=FALSE**

TRUE to keep the old mapping information after the resize; FALSE to discard it.

**Prototype:**

**int getNumMaps() const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of mapping channels in use.

**Prototype:**

**BOOL mapSupport(int mp) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the specified mapping channel is supported; otherwise FALSE.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**Prototype:**

**void setMapSupport(int mp, BOOL support=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets whether the specified mapping channels is supported or not.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**BOOL support=TRUE**

TRUE to indicate the channel is supported; otherwise FALSE.

**Prototype:**

**void setNumMapVerts(int mp, int ct, BOOL keep=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the number of texture or vertex color vertices for the specified mapping channel of this mesh.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**int ct**

The number of vertices to allocate.

**BOOL keep=FALSE**

If TRUE previous values are kept; otherwise they are discarded.

**Prototype:**

**int getNumMapVerts(int mp) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of texture or vertex color vertices for the specified channel of this mesh.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**Prototype:**

**void setNumMapFaces(int mp, int ct, BOOL keep=FALSE, int**

**oldCt=0);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the number of texture or vertex color faces for the specified channel of this mesh.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**int ct**

The number of faces to allocate.

**BOOL keep=FALSE**

If TRUE previous values are kept; otherwise they are discarded.

**Prototype:**

**UVVert \*mapVerts(int mp) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the list of **UVVerts** for the specified channel of this mesh.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**Prototype:**

**TVFace \*mapFaces(int mp) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the list of **TVFaces** for the specified channel of this mesh.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**Prototype:**

```
void setMapVert(int mp, int i, const UVVert&xyz);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets a single texture or vertex color value for the specified channel of this mesh.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**int i**

The zero based index of the vertex to set.

**const UVVert&xyz**

The value to set.

**Prototype:**

```
void MakeMapPlanar(int mp);
```

**Remarks:**

This method is available in release 3.0 and later only.

Applies a simple planar mapping to the specified channel. This is done by copying the mesh topology and vertex locations into the map.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**Prototype:**

```
BitArray GetIsoMapVerts(int mp);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a BitArray with a bit set for each isolated vertex (un-referenced by any face) for the specified channel.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**Prototype:**

```
void DeleteMapVertSet(int mp, BitArray set, BitArray *fdel=NULL);
```

**Remarks:**

This method is available in release 3.0 and later only.

Deletes the map vertices indicated.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**BitArray set**

Indicates which map verts should be deleted. set.GetSize() should equal this mesh's getNumMapVerts(mp).

**BitArray \*fdel=NULL**

If non-NULL, this points to a BitArray that will be filled in with the faces that will need to be deleted or have new map verts assigned because they used a map vert that was deleted. (The size will be set to this mesh's **numFaces**.)

**Prototype:**

```
void DeleteIsoMapVerts(int mp);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method deletes each isolated vertex (un-referenced by any face) for the specified channel.

**Parameters:**

**int mp=-1**

Specifies the channel. See [List of Mapping Channel Index Values](#). The default value of -1 indicates to do all active maps.

**Prototype:**

```
void DeleteIsoMapVerts();
```

**Remarks:**

This method is available in release 4.0 and later only.

This method deletes each isolated vertex (un-referenced by any face) for the all active maps.

**Prototype:**

```
void freeMapVerts(int mp);
```

**Remarks:**

This method is available in release 3.0 and later only.

Deallocates the texture or vertex color vertices for the specified channel of this mesh.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

**Prototype:**

```
void freeMapFaces(int mp);
```

**Remarks:**

This method is available in release 3.0 and later only.

Deallocates the texture or vertex color faces for the specified channel of this mesh.

**Parameters:**

**int mp**

Specifies the channel. See [List of Mapping Channel Index Values](#).

## **Vertex Data Methods**

**Prototype:**

```
void setNumVData(int ct, BOOL keep=FALSE);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the number of channels of vertex data used by the mesh.

**Parameters:**

**int ct**

The number of elements of vertex data to set.

**BOOL keep=FALSE**

If TRUE any old vertex data is kept; otherwise it is discarded.

**Prototype:**

**int getNumVData() const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of vertex data channels maintained by this mesh.

**Prototype:**

**BOOL vDataSupport(int vd) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the specified channel of vertex data is available for this mesh; otherwise FALSE.

**Parameters:**

**int vd**

The vertex data channel. See [List of Vertex Data Index Options](#).

**Prototype:**

**void setVDataSupport(int vd, BOOL support=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets if the specified channel of vertex data is supported by this mesh.

**Parameters:**

**int vd**

The vertex data channel. See [List of Vertex Data Index Options](#).

**BOOL support=TRUE**

TRUE to indicate the channel is supported; FALSE to indicate it's not. If TRUE is specified then **numVerts** elements are allocated (if needed). If

FALSE is specified the data for the channel is freed.

**Prototype:**

**void \*vertexData(int vd) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the vertex data for the specified channel or NULL if the channel is not supported. If supported then the size of this array is **numVerts**.

**Parameters:**

**int vd**

The vertex data channel. See [List of Vertex Data Index Options](#).

**Prototype:**

**float \*vertexFloat(int vd) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the floating point vertex data for the specified channel of this mesh or NULL if the channel is not supported. If supported then the size of this array is **numVerts**.

**Parameters:**

**int vd**

The vertex data channel. See [List of Vertex Data Index Options](#).

**Prototype:**

**void freeVData(int vd);**

**Remarks:**

This method is available in release 3.0 and later only.

Deallocates the vertex data for the specified channel.

**Parameters:**

**int vd**

The vertex data channel. See [List of Vertex Data Index Options](#).

**Prototype:**

**void freeAllVData();**

**Remarks:**

This method is available in release 3.0 and later only.

Deallocates the vertex data from all the channels and sets the number of supported channels to 0.

**Prototype:**

**float \*getVertexWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the floating point vertex weight data.

**Prototype:**

**void SupportVertexWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the channel support for the vertex weights channel (**VDATA\_WEIGHT**).

**Prototype:**

**void ClearVertexWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Clears (deallocates) the vertex weights channel data.

**Prototype:**

**void freeVertexWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Deallocates the vertex weights channel data (same as **ClearVertexWeights()** above).

**Prototype:**

**float \*getVSelectionWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the floating point vertex selection weights data.

**Prototype:**

**void SupportVSelectionWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the channel support for the vertex weights channel (**VDATA\_SELECT**).

**Prototype:**

**void ClearVSelectionWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Clears (deallocates) the vertex selection weights channel data.

**Prototype:**

**void freeVSelectionWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Deallocates the vertex selection weights channel data (same as **ClearVSelectionWeights()** above).

## **Normals Methods**

**Prototype:**

**void setNormal(int i, const Point3 &xyz);**

**Remarks:**

Sets a single 'rendered' normal in the **rVerts** array of **RVertex** instances.

**Parameters:**

**int i**

A zero based index into the **rVerts** array of the normal to store.

**const Point3 &xyz**

The normal to store in device coordinates. This should be a unit vector.

**Prototype:**

**Point3& getNormal(int i);**

**Remarks:**

Returns the 'i-th' 'rendered' normal from the **rVerts** array.

**Parameters:**

**int i**

A zero based index into the **rVerts** array of the normal to get.

**Prototype:**

**void setFaceNormal(int i, const Point3 &xyz);**

**Remarks:**

Sets the 'i-th' face normal.

**Parameters:**

**int i**

A zero based index into the face normal array of the normal to store.

**const Point3 &xyz**

The face normal to store. This should be a unit vector.

**Prototype:**

**Point3& getFaceNormal(int i);**

**Remarks:**

Returns the 'i-th' face normal.

**Parameters:**

**int i**

Specifies the index of the face normal to retrieve.

**Prototype:**

```
void buildNormals();
```

**Remarks:**

This method resolves the normals on the **RVertex** array. If the Mesh already has normals at each vertex, the normal is just moved to the **RVertex** array. See [Class RVertex](#) and [Class RNormal](#).

If you are creating a **Mesh** by hand, after you are done specifying all the vertices and faces, this method should be called. This allocates the **RVertex** and **RNormal** database for the **Mesh**. This will allow you to query the **Mesh** and ask about normals on the vertices. Also, if you deform a Mesh (i.e. take one of the vertices and move it), you should call this method again. Actually, if you are only moving one normal you only need to smooth the polygons that share the vertex. However, there is no method to smooth a subset of a **Mesh**, you either have to do it by hand or call this method to smooth the entire **Mesh**.

This method also builds the face normals for the mesh.

**Prototype:**

```
void buildRenderNormals();
```

**Remarks:**

This method is similar to **buildNormals()** above, but ignores the material index (**mtlIndex**). In other words, the difference between this and **buildNormals()** is that it doesn't look at the **mtlIndex** of the faces: normals of faces with the same smoothing group are averaged regardless.

**Prototype:**

```
void checkNormals(BOOL illum);
```

**Remarks:**

This method can be used to build the normals and allocate **RVert** space only if necessary. This is a very inexpensive call if the normals are already calculated. When **illum** is FALSE, only the **RVerts** allocation is checked (since normals aren't needed for non-illum rendering). When **illum** is TRUE, normals will also be built, if they aren't already. So, to make sure normals are built, call this with **illum=TRUE**.

**Parameters:**

**BOOL illum**

If TRUE then normals are built. If FALSE then only the **RVert** array is allocated.

**Prototype:**

**RVertex &getRVert(int i)**

**Remarks:**

This method returns the 'i-th' RVertex.

**Parameters:**

**int i**

Specifies the index of the RVertex to retrieve.

**Prototype:**

**RVertex \*getRVertPtr(int i)**

**Remarks:**

This method returns a pointer to the 'i-th' RVertex.

**Parameters:**

**int i**

Specifies the index of the RVertex to retrieve.

## **Material Methods**

**Prototype:**

**MtlID getFaceMtlIndex(int i);**

**Remarks:**

Retrieves the zero based material index of the 'i-th' face.

**Parameters:**

**int i**

Specifies the face index. This is the zero based index into the **faces** array.

**Prototype:**

**void setFaceMtlIndex(int i, MtlID id);**

**Remarks:**

Sets the material index of the 'i-th' face.

**Parameters:**

**int i**

Specifies the face index. This is the zero based index into the **faces** array.

**MtlID id**

The material index for the 'i-th' face.

**Prototype:**

**void setMtlIndex(MtlID i);**

**Remarks:**

This method is no longer used.

**Prototype:**

**MtlID getMtlIndex();**

**Remarks:**

This method is no longer used.

**Prototype:**

**void createFaceMtlIndexList();**

**Remarks:**

This method is no longer used.

**Prototype:**

**void freeFaceMtlIndexList();**

**Remarks:**

This method is no longer used.

## **Render / Hit Test / Snap Methods**

**Prototype:**

```
void render(GraphicsWindow *gw, Material *ma, RECT *rp,  
int compFlags, int numMat=1, InterfaceServer *pi = NULL);
```

**Remarks:**

Renders this Mesh using the specified graphics window and array of materials.

Note: If a display routine makes multiple calls to this method you need to have called:

```
gw->setMaterial(inode->Mtls()[0]);
```

before calling **Mesh::render()**. If you don't then you may get the wrong material for material ID 0.

**Parameters:**

**GraphicsWindow \*gw**

Points to the graphics window to render to.

**Material \*ma**

The list of materials to use to render the mesh. See [Class Material](#), [Class INode - Material methods](#).

**RECT \*rp**

Specifies the rectangular region to render. If the mesh should be rendered to the entire viewport pass NULL.

**int compFlags**

One or more of the following flags:

**COMP\_TRANSFORM**

Forces recalculation of the model to screen transformation; otherwise attempt to use the cache.

**COMP\_IGN\_RECT**

Forces all polygons to be rendered; otherwise only those intersecting the box will be rendered.

**COMP\_LIGHTING**

Forces re-lighting of all vertices (as when a light moves); otherwise only re-light moved vertices

**COMP\_ALL**

All of the above flags.

**COMP\_OBJSELECTED**

If this bit is set then the node being displayed by this mesh is selected.

Certain display flags only activate when this bit is set.

**int numMat=1**

The number of materials for the mesh.

**InterfaceServer \*pi = NULL**

This pointer to an InterfaceServer can be used to get hold of the IXTCAccess pointer.

And IXTCAccess interface can also be obtained from the object by calling

**Object::GetInterface**

**(IXTCACCESS\_INTERFACE\_ID).**

#### Sample Code:

The following code shows this method being used to render the mesh as part of the **BaseObject::Display()** method:

```
int SimpleObject::Display(TimeValue t, INode* inode,
    ViewExp *vpt, int flags)
{
    if (!OKtoDisplay(t)) return 0;
    GraphicsWindow *gw = vpt->getGW();
    Matrix3 mat = inode->GetObjectTM(t);
    UpdateMesh(t); // UpdateMesh() just calls BuildMesh()
    gw->setTransform(mat);
    mesh.render(gw, inode->Mtls(),
        (flags&USE_DAMAGE_RECT) ? &vpt->GetDammageRect() : NULL,
        COMP_ALL, inode->NumMtls());
    return(0);
}
```

#### Prototype:

```
BOOL select(GraphicsWindow *gw, Material *ma,
    HitRegion *hr, int abortOnHit = FALSE, int numMat=1);
```

#### Remarks:

Checks the given HitRecord **hr** to see if it intersects this Mesh object.

#### Parameters:

**GraphicsWindow \*gw**

Points to the graphics window to check.

**Material \*ma**

The list of materials for the mesh.

**HitRegion \*hr**

This describes the properties of a region used for the hit testing. See [Class HitRegion](#).

**int abortOnHit = FALSE**

If nonzero, the hit testing is complete after any hit; otherwise all hits are checked.

**int numMat=1**

The number of materials for the mesh.

**Return Value:**

TRUE if the item was hit; otherwise FALSE.

**Prototype:**

```
void snap(GraphicsWindow *gw, SnapInfo *snap, IPoint2 *p,  
Matrix3 &tm);
```

**Remarks:**

Checks to see if there is a snap point near the given mouse point.

**Parameters:****GraphicsWindow \*gw**

The graphics window in which to check.

**SnapInfo \*snap**

This structure describes the snap settings used, and the results of the snap test. See [Structure SnapInfo](#).

**IPoint2 \*p**

The mouse point to check.

**Matrix3 &tm**

The object transformation matrix. This is the transformation to place the object into the world coordinate system.

**Sample Code:**

```
// Checks to see if there is a snap point near the given mouse point.  
void TestObject::Snap(TimeValue t, INode* inode, SnapInfo *snap,  
IPoint2 *p, ViewExp *vpt) {
```

```

// Grab the object TM
Matrix3 tm = inode->GetObjectTM(t);
// Grab the graphics window
GraphicsWindow *gw = vpt->getGW();
// Make sure our mesh is up to date
UpdateMesh(t); // UpdateMesh() just calls BuildMesh()
// Set the transform in the GW
gw->setTransform(tm);
// Let the mesh do the work...
mesh.snap( gw, snap, p, tm );
}

```

**Prototype:**

```

BOOL SubObjectHitTest(GraphicsWindow *gw, Material *ma,
HitRegion *hr, DWORD flags,
SubObjHitList& hitList, int numMat=1 );

```

**Remarks:**

This method may be called to perform sub-object hit testing on this mesh.

**Parameters:**

**GraphicsWindow \*gw**

The graphics window associated with the viewport the mesh is being hit tested in.

**Material \*ma**

The list of materials for the mesh. See [Class Material](#), [Class INode - Material methods](#).

**HitRegion \*hr**

This describes the properties of a region used for the hit testing. See [Class HitRegion](#).

**DWORD flags**

Flags for sub object hit testing. One or more of the following values:

**SUBHIT\_SELONLY**

Selected only.

**SUBHIT\_UNSELONLY**

Unselected only.

## **SUBHIT\_ABORTONHIT**

Abort hit testing on the first hit found.

## **SUBHIT\_SELSOLID**

This treats selected items as solid and unselected items as not solid. Treating an item as solid means the face will be hit if the mouse is anywhere inside the face region and not just over a visible edge.

## **SUBHIT\_USEFACESEL**

When this bit is set, the sel only and unsel only tests will use the faces selection when doing a vertex level hit test

## **SUBHIT\_VERTS**

Hit test vertices.

## **SUBHIT\_FACES**

Hit test faces.

## **SUBHIT\_EDGES**

Hit test edges.

## **SubObjHitList& hitList**

The results are stored here. See [Class SubObjHitList](#).

## **int numMat=1**

The number of materials for the mesh.

## **Return Value:**

TRUE if the item was hit; otherwise FALSE.

## **Prototype:**

**void displayNormals(int b, float sc)**

## **Remarks:**

Controls the display of surface normals on the mesh object.

Note that there may be more than one normal per vertex if faces that share the vertex are in non-overlapping smoothing groups. In this case, all normals associated with the given vertex are drawn.

## **Parameters:**

### **int b**

Nonzero to display the normals; zero to turn off normals display.

This can be a combination of **MESH\_DISP\_FACE\_NORMALS** and

**MESH\_DISP\_VERTEX\_NORMALS.** (The arguments may be or'ed together to display both.) For backwards compatibility, **MESH\_DISP\_FACE\_NORMALS** is #define'd to be "1", so sending in "TRUE" will turn on display of face normals, as before.

**float sc**

This specifies the length that should be used (in world units) to display the normals. Since all normals start out with length =1 they probably would be too small to see unless they were scaled

## **Bounding Box Methods**

**Prototype:**

**void buildBoundingBox();**

**Remarks:**

Computes the bounding box of the Mesh. If surface normals are displayed, they are taken into account in the computation of the box. The bounding box is stored with the Mesh object, use **getBoundingBox()** to retrieve it.

**Prototype:**

**Box3 getBoundingBox(Matrix3 \*tm=NULL);**

**Remarks:**

Retrieves the bounding box of the mesh object.

**Parameters:**

**Matrix3 \*tm=NULL**

The optional TM allows the box to be calculated in any space. NOTE: This computation will be slower because all the points must be transformed.

**Return Value:**

The bounding box of the Mesh.

## **Edge Methods**

**Prototype:**

**void displayAllEdges(int b);**

**Remarks:**

Controls the display of hidden edges of this mesh object. This just sets the Boolean in the mesh that controls whether "hidden" edges (for instance the diagonals on the sides of a cube) are displayed.

**Parameters:****int b**

Nonzero to display all the hidden edges; otherwise zero.

**Prototype:****void EnableEdgeList(int e);****Remarks:**

This method is used internally.

**Prototype:****void BuildVisEdgeList();****Remarks:**

This method is used internally.

**Prototype:****void DivideEdge(DWORD edge, float prop, bool visDiag1=TRUE, bool fixNeighbors=TRUE, bool visDiag2=TRUE);****Remarks:**

This method is available in release 2.5 and later only.

Divides the edge, creating a new point. The face directly using this edge (face edge/3) is also divided in two, and other faces using the edge may optionally be split to use the split edges.

**Parameters:****DWORD edge**

The edge to divide.

**float prop**

The proportion along the edge to make the division. An edge can be expressed as  $ff*3+ee$ , where  $ff$  is a face using this edge and  $ee$  represents which pair of vertices the edge is between, faces[ $ff$ ]->v[ $ee$ ] and faces[ $ff$ ]->v[( $ee+1$ )%3]. The

new point is created at (1-prop) times the first vertex plus prop times the second. prop may vary from 0 to 1. prop=.5 gives the same result that DivideEdge (DWORD edge) would generate.

**bool visDiag1=TRUE**

Indicates whether the "diagonal" used to split the primary face this edge is on (edge/3) is visible or not.

**bool fixNeighbors=TRUE**

Indicates whether other faces using this edge should also be split to use the two new edges, or if they should be left as they were. In a typical mesh, there is one other face using this edge, the face on the "other side" of the edge. If fixNeighbors is FALSE, the "other side" in this case would still use the original edge, while the face on this side would be split to use the two new edges. This would create a "hole" in the mesh.

**bool visDiag2=TRUE**

Indicates whether the "diagonals" used to split other faces using this edge are visible or not. This argument is not used if fixneighbors is FALSE.

**Prototype:**

```
void DivideFace(DWORD face, DWORD e1, DWORD e2, float prop1=.5f, float prop2=.5f, bool fixNeighbors=TRUE, bool split=FALSE);
```

**Remarks:**

This method is available in release 2.5 and later only.

Cuts a face into three faces, arranged as a quad and a triangle, by slicing from a point on one edge to a point on another.

**Parameters:**

**DWORD face**

The face to be divided.

**DWORD e1, e2**

The index of edges to be cut. For instance, if you wanted to cut from the edge between v[0] and v[1], to the edge between v[2] and v[0], you would use e1=0 and e2=2.

**float prop1=.5f**

The proportion along edge e1 to start cutting.

**float prop2=.5f**

The proportion along edge e2 to stop cutting.

**bool fixNeighbors=TRUE**

Indicates whether faces also using the cut edges should be split to use the new, subdivided edges.

**bool split=FALSE**

Indicates that the triangle and quad created by this action should use different vertices. If TRUE, the vertices created by the cut are duplicated, with one set being used for faces on one side and the other set being used by faces on the other side.

**Prototype:**

**void TurnEdge(DWORD edge, DWORD \*otherEdge);**

**Remarks:**

This method is available in release 2.5 and later only.

Turns an edge. The quadrilateral formed by the triangles on either side of this edge essentially has its diagonal switched.

**Parameters:**

**DWORD edge**

The edge to be turned.

**DWORD \*otherEdge**

If non-NULL, this should point to a variable in which the index of the "other side" of this edge should be stored. In essence, the two sides of an edge used by two faces, f1 and f2, are stored in two different locations, f1\*3+e1 and f2\*3+e2, where e1, e2 are 0, 1, or 2. This argument is provided so you have easy access to the other side of the edge, if desired, to make easy selection or visibility changes to the edge or other changes to the faces on both sides.

**Prototype:**

**void DrawVisEdgeList(GraphicsWindow \*gw, DWORD flags);**

**Remarks:**

This is used internally.

**Prototype:**

```
void HitTestVisEdgeList(GraphicsWindow *gw, int useFloats,  
int abortOnHit );
```

**Remarks:**

This is used internally.

**Prototype:**

```
void InvalidateEdgeList();
```

**Remarks:**

This is used internally.

**Prototype:**

```
void InvalidateGeomCache();
```

**Remarks:**

Call this method after the geometry of this Mesh has changed. It invalidates the bounding box, and tosses out the cached normals and edge list.

**Prototype:**

```
void InvalidateTopologyCache();
```

**Remarks:**

Call this method after you alter vertex or face lists or revise edge visibility flags. It will invalidate the edge and strip database maintained by the mesh.

**Prototype:**

```
void FreeAll();
```

**Remarks:**

Cleans up the allocated arrays. This frees the Faces, Verts, TVerts, TVFaces, FaceMtlIndexList and invalidates the geometry cache.

## **Strip Methods**

**Prototype:**

**BOOL BuildStrips();**

**Remarks:**

This method is available in release 2.0 and later only.

It builds the strips database inside the mesh. See the method **BuildStripsAndEdges()** below.

**Prototype:**

**void BuildStripsAndEdges();**

**Remarks:**

This method is available in release 2.0 and later only.

It builds the strips and edges database inside the mesh. When developers create a new mesh for display then this method should be called. See the section on Stripping in the Advanced Topics section [Working with Meshes](#) for details on this method.

**Prototype:**

**void InvalidateStrips();**

**Remarks:**

This method invalidates the strips database.

**Prototype:**

**void Stripify(Strip \*s, StripData \*sd, int vtx);**

**Remarks:**

This method is used internally.

**Prototype:**

**void getStripVertColor(GraphicsWindow \*gw, int cv, int flipped, MtlID mID, DWORD smGroup, Point3 &rgb);**

**Remarks:**

This method is used internally.

**Prototype:**

**void getStripNormal(GraphicsWindow \*gw, int cv, MtlID mID, DWORD smGroup, Point3 &nor);**

**Remarks:**

This method is used internally.

**Prototype:**

```
void Draw3DStrips(GraphicsWindow *gw, Material *ma, int numMat);
```

**Remarks:**

This method is used internally.

**Prototype:**

```
BOOL getStripTVert(GraphicsWindow *gw, int cv, Point3 &uvw);
```

**Remarks:**

This method is used internally.

**Prototype:**

```
void DrawStrips(GraphicsWindow *gw, Material *ma, int numMat);
```

**Remarks:**

This method is used internally.

## **Data Flow Evaluation Methods**

**Prototype:**

```
void ShallowCopy(Mesh *amesh, ULONG_PTR channels);
```

**Remarks:**

Makes a copy of the specified channels of the specified Mesh object's data structures (but not all the data in these structures) into this Mesh. For example the **verts**, **tVerts**, **tvFaces**, ... are not copied.

**Parameters:**

**Mesh \*amesh**

Specifies the source Mesh to copy.

**ULONG\_PTR channels**

Specifies the channels to copy. See [List of Channels](#). Note: Prior to R4 this parameter was an unsigned long.

**Prototype:**

```
void DeepCopy(Mesh *amesh, ULONG_PTR channels);
```

**Remarks:**

Makes a complete copy of the specified channels of the specified Mesh object (its data structures and all the data in these structures) into this Mesh.

**Parameters:**

**Mesh \*amesh**

Specifies the source Mesh to copy.

**ULONG\_PTR channels**

Specifies the channels to copy. See [List of Channels](#). Note: Prior to R4 this parameter was an unsigned long.

**Prototype:**

```
void NewAndCopyChannels(ULONG_PTR channels);
```

**Remarks:**

This method replaces the specified channels with newly allocated copies.

**Parameters:**

**ULONG\_PTR channels**

Specifies the channels to copy. See [List of Channels](#). Note: Prior to R4 this parameter was an unsigned long.

**Prototype:**

```
void FreeChannels(ULONG_PTR channels, int zeroOthers=1);
```

**Remarks:**

Release the memory associated with the specified channels. For example if the TOPO\_CHANNEL is specified the faces are freed, if the GEOM\_CHANNEL is specified the vertices are freed, etc.

**Parameters:**

**ULONG\_PTR channels**

Specifies the channels to free. Channels not specified are left intact. See [List of Channels](#). Note: Prior to R4 this parameter was an unsigned long.

**int zeroOthers=1**

If nonzero then the various pointers are set to NULL and their counts are zeroed. For example **faces**, **verts**, **tVerts**, and **tvFace** are set to NULL and **numFaces**, **numVerts** and **numTVerts** are set to 0. If this is passed as 0, these pointers and counts are left unaltered.

## Mesh / Display Flags

### Prototype:

**void SetDispFlag(DWORD f)**

### Remarks:

Sets the state of the display flags. See **Data Members:** above for a list of the display flags.

### Parameters:

**DWORD f**

Specifies the flags to set.

### Prototype:

**DWORD GetDispFlag(DWORD f)**

### Remarks:

Returns the state of the specified display flags. See **Data Members:** above for a list of the display flags.

### Parameters:

**DWORD f**

Specifies the flags to retrieve.

### Prototype:

**void ClearDispFlag(DWORD f)**

### Remarks:

Sets the state of the specified display flags to 0.

### Parameters:

**DWORD f**

Specifies the flags to clear.

**Prototype:**

**void SetFlag(DWORD f)**

**Remarks:**

Sets the state of the mesh flags.

**Parameters:**

**DWORD f**

Specifies the flags to set. See [List of Mesh Flags](#).

**Prototype:**

**DWORD GetFlag(DWORD f)**

**Remarks:**

Returns the state of the specified mesh flags. See **Data Members:** above for a list of the mesh flags.

**Parameters:**

**DWORD f**

Specifies the flags to retrieve. See [List of Mesh Flags](#).

**Prototype:**

**void ClearFlag(DWORD f)**

**Remarks:**

Sets the state of the specified mesh flags to 0.

**Parameters:**

**DWORD f**

Specifies the flags to clear. See [List of Mesh Flags](#).

## **Selection Access**

**Prototype:**

**BitArray& VertSel()**

**Remarks:**

Retrieves the bits representing the vertex selection status. See **Data Members:**

above.

**Return Value:**

The vertex selection status.

**Prototype:**

**BitArray& FaceSel()**

**Remarks:**

Retrieves the bits representing the face selection status. See **Data Members:** above.

**Return Value:**

The face selection status.

**Prototype:**

**BitArray VertexTempSel();**

**Remarks:**

Constructs a vertex selection list based on the current selection level. For example if the selection level is at object level all the bits are set (vertices, faces and edges). If the selection level is at vertex level only the selected vertex bits are set.

**Return Value:**

A BitArray reflecting the current selection level.

**Sample Code:**

```
BitArray sel = mesh->VertexTempSel();
for ( int i = 0; i < mesh->getNumVerts(); i++ ) {
    if ( sel[i] ) {
        ...
    }
}
```

## **Intersect Ray / Weld / Optimize / Apply Mapping**

**Prototype:**

**int IntersectRay(Ray& ray, float& at, Point3& norm);**

**Remarks:**

Calculates the intersection of the specified ray with this mesh object. This allows Mesh objects to easily implement the **Object::IntersectRay()** method.

**Parameters:****Ray& ray**

Specifies the origin and direction of the ray to intersect with the mesh. See [Class Ray](#).

**float& at**

The computed point of intersection on the surface of the mesh.

**Point3& norm**

The face normal at the point of intersection (**at**).

**Return Value:**

Nonzero if the ray intersected the mesh object; otherwise 0. Note that this method ignores backfaces when computing the result.

**Prototype:**

```
int IntersectRay(Ray& ray, float& at, Point3& norm, DWORD  
&fi, Point3 &bary);
```

**Remarks:**

This method is available in release 2.0 and later only.

Calculates the intersection of the specified ray with this mesh object. This new version also returns the face index that was intersected and the barycentric coordinates of that face.

Barycentric coordinates are the coordinates relative to the triangular face. The barycentric coordinates of a point **p** relative to a triangle describe that point as a weighted sum of the vertices of the triangle. If the barycentric coordinates are **b0**, **b1**, and **b2**, then:

$$\mathbf{p} = \mathbf{b0} * \mathbf{p0} + \mathbf{b1} * \mathbf{p1} + \mathbf{b2} * \mathbf{p2};$$

where **p0**, **p1**, and **p2** are the vertices of the triangle. The **Point3** returned by this method has the barycentric coordinates stored in its three coordinates. These coordinates are relative to the triangular face that was intersected. These barycentric coordinates can be used to interpolate any quantity whose value is

known at the vertices of the triangle.

**Parameters:**

**Ray& ray**

Specifies the origin and direction of the ray to intersect with the mesh. See [Class Ray](#).

**float& at**

The computed point of intersection on the surface of the mesh.

**Point3& norm**

The face normal at the point of intersection (**at**).

**DWORD &fi**

The face index of the face that was intersected is returned here.

**Point3 &bary**

The barycentric coordinates of the face that was hit.

**Return Value:**

Nonzero if the ray intersected the mesh object; otherwise 0.

**Prototype:**

```
void WeldCollinear(BitArray &set);
```

**Remarks:**

This method is used internally.

**Prototype:**

```
void Optimize(float normThresh, float edgeThresh,  
float bias, float maxEdge, DWORD flags, MeshOpProgress  
*prog=NULL);
```

**Remarks:**

Allows this Mesh to be reduced in complexity by reducing the number of faces based on a surface normal threshold. Adjacent faces whose difference in surface normal angle falls below the specified threshold will be collapsed into a single triangle. The Mesh may also have its edge visibility set based on a surface normal threshold.

**Parameters:**

**float normThresh**

When the angle between adjacent surface normals is less than this value the optimization is performed. This angle is specified in radians.

**float edgeThresh**

When the angle between adjacent surface normals is less than this value the auto edge is performed (if the **OPTIMIZE\_AUTOEDGE** flag is set). This angle is specified in radians.

**float bias**

When optimizing mesh objects, as the optimization increases, you can get lots of long skinny 'degenerate' triangles (that cause rendering artifacts). Increasing the bias parameter keeps triangles from becoming degenerate. The range of values is from 0 to 1 (where 0 turns bias off). Values close to 1 reduce the amount of optimization in favor of maintaining equilateral triangles.

**float maxEdge**

This parameter is available in release 2.0 and later only.

This will prevent the optimize function from creating edges longer than this value. If this parameter is  $\leq 0$  no limit is placed on the length of the edges.

**DWORD flags**

These flags control the optimization. Specify zero or more of the following values:

**OPTIMIZE\_SAVEMATBOUNDRIES**

Specifies that faces won't be collapsed across a material boundary.

**OPTIMIZE\_SAVESMOOTHBOUNDRIES**

Specifies that faces won't be collapsed across a dissimilar smoothing group boundary.

**OPTIMIZE\_AUTOEDGE**

Specifies that the edge visibility should be set automatically based on the angle between adjacent surface normals. This will only set edges as invisible - it will not set edges as visible.

**MeshOpProgress \*prog=NULL**

A callback used for lengthy optimize operations. See [Class MeshOpProgress](#).

**Prototype:**

**void ApplyUVWMap(int type, float utile, float vtile,**

**float wtile,int uflip, int vflip, int wflip,  
int cap,const Matrix3 &tm, int channel=1);**

**Remarks:**

This method may be called to map this Mesh with UVW mapping coordinates.

**Parameters:**

**int type**

The mapping type. One of the following values:

**MAP\_PLANAR**  
**MAP\_CYLINDRICAL**  
**MAP\_SPHERICAL**  
**MAP\_BALL**  
**MAP\_BOX**

**float utile**

Number of tiles in the U direction.

**float vtile**

Number of tiles in the V direction.

**float wtile**

Number of tiles in the W direction.

**int uflip**

If nonzero the U values are mirrored.

**int vflip**

If nonzero the V values are mirrored.

**int wflip**

If nonzero the W values are mirrored.

**int cap**

This is used with **MAP\_CYLINDRICAL**. If nonzero, then any face normal that is pointing more vertically than horizontally will be mapped using planar coordinates.

**const Matrix3 &tm**

This defines the mapping space. As each point is mapped, it is multiplied by this matrix, and then it is mapped.

**int channel=1**

This parameter is available in release 2.0 and later only.

This indicates which channel the mapping is applied to -- **channel==1** corresponds to the original texture mapping channel.

**Prototype:**

**MeshMap & Map(int mp);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the map for the specified map channel.

**Parameters:**

**int mp**

The map channel.

**Default Implementation:**

```
{ return (mp<0) ? hmaps[-1-mp] : maps[mp]; }
```

## **Normals / Smoothing / Edges / Face check Functions**

**Prototype:**

**void FlipNormal(int i);**

**Remarks:**

Flips the surface normal of the 'i-th' face (this just rearranges the indices for the face structure). This also makes sure the edge flags are rearranged as well. If there are UV coordinates these are rearranged appropriately.

**Parameters:**

**int i**

The index of the face to flip.

**Prototype:**

**void UnifyNormals(BOOL selOnly);**

**Remarks:**

Unifies the surfaces normals of this Mesh. This may be for selected faces, or the entire mesh.

**Parameters:**

## **BOOL selOnly**

If TRUE only the selected faces are unified.

### **Prototype:**

```
void AutoSmooth(float angle,BOOL useSel,BOOL  
preventIndirectSmoothing=FALSE);
```

### **Remarks:**

Performs an auto smooth on the mesh, setting the smoothing groups based on the surface normals.

### **Parameters:**

#### **float angle**

The minimum angle between surface normals for smoothing to be applied, in radians.

#### **BOOL useSel**

If TRUE only the selected faces are smoothed.

#### **BOOL preventIndirectSmoothing=FALSE**

This parameter is available in release 2.0 and later only.

TRUE to turn on; FALSE to leave off. This matches the option in the Smooth Modifier UI -- use this to prevent smoothing "leaks" when using this method.

If you use this method, and portions of the mesh that should **not** be smoothed become smoothed, then try this option to see if it will correct the problem.

Note that the problem it corrects is rare, and that checking this slows the Auto Smooth process.

### **Prototype:**

```
Edge *MakeEdgeList(int *edgeCount, int flagdbs=0);
```

### **Remarks:**

This method is used internally.

### **Prototype:**

```
int DeleteFlaggedFaces();
```

### **Remarks:**

This method removes faces from the face list with the **FACE\_WORK** flag

set.

**Return Value:**

The number of faces deleted.

**Prototype:**

**void DeleteSelected();**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes all selected elements of the current selection level.

**Prototype:**

**void DeleteVertSet(BitArray set);**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes the vertices as specified by the BitArray.

**Parameters:**

**BitArray set**

Set of bits to indicate the vertices to delete.

**Prototype:**

**void DeleteFaceSet(BitArray set, BitArray \*isoVert=NULL);**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes faces as specified by the BitArray.

**Parameters:**

**BitArray set**

Set of bits to indicate the faces to delete.

**BitArray \*isoVert=NULL**

If non NULL then this method will be setup to flag vertices that were isolated by the face deletion. This set can then be passed to **DeleteVertSet()** to delete isolated vertices.

**Prototype:**

**BOOL DoesFaceExist(DWORD v0, DWORD v1, DWORD v2);**

**Remarks:**

This method may be called to determine if an equivalent face already exists.

**Parameters:**

**DWORD v0**

Index of the first vertex.

**DWORD v1**

Index of the second vertex.

**DWORD v2**

Index of the third vertex.

**Return Value:**

TRUE if an equivalent face already exists; otherwise FALSE.

**Prototype:**

**BOOL RemoveDegenerateFaces();**

**Remarks:**

Removes faces that have two or more equal indices.

**Return Value:**

TRUE if any degenerate faces were found; otherwise FALSE.

**Prototype:**

**BOOL RemoveIllegalFaces();**

**Remarks:**

Removes faces that have indices that are out of range

**Return Value:**

TRUE if any illegal faces were found; otherwise FALSE.

**Prototype:**

**Point3 FaceNormal(DWORD fi, BOOL nrmlize=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns the normal of the specified face. If **nrmlize** is TRUE, it makes this normal unit length. Otherwise, it's the edge cross-product length, which is actually 2 times the area of the face.

**Parameters:**

**DWORD fi**

Specifies the face whose normal is returned.

**BOOL nrmlize=FALSE**

Use TRUE to make the normal unit length.

**Prototype:**

**Point3 FaceCenter(DWORD fi);**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns the center of the specified face.

**Parameters:**

**DWORD fi**

Specifies the face whose center is returned.

**Prototype:**

**float AngleBetweenFaces(DWORD f0, DWORD f1);**

**Remarks:**

Returns the angle between two face surface normals in radians.

**Parameters:**

**DWORD f0**

Index of the first face.

**DWORD f1**

Index of the second face.

**Prototype:**

**Point3 BaryCoords(DWORD face, Point3 p);**

**Remarks:**

Computes and returns the barycentric coordinates of a point in the plane of a

face relative to that face.

**Parameters:**

**DWORD face**

The index of the face to check.

**Point3 p**

The input point.

**Return Value:**

The point **p** barycentric coordinates. If the point **p** is inside the face the returned values will sum to one. Note: If the face (or set of 3 points) is degenerate, ie if it has a zero length normal vector  $((p1-p0) \wedge (p2-p0))$ , the methods return Point3(-1,1,1).

**Prototype:**

**void FaceCenterTessellate(BOOL ignoreSel=FALSE);**

**Remarks:**

This method is available in release 2.0 and later only.

Tesselates the mesh (or only the selected faces) using the face/center method. This means each face is subdivided by lines from a new vertex at the center to the original vertices.

**Parameters:**

**BOOL ignoreSel=FALSE**

If TRUE the entire mesh is tessellated; otherwise only the selected faces.

**Prototype:**

**void EdgeTessellate(float tens,BOOL ignoreSel=FALSE);**

**Remarks:**

This method is available in release 2.0 and later only.

Edge tessellates the mesh using the specified tension parameter. This method can operate on the entire mesh or only the selection set. Edge tessellation means that faces are internally subdivided, with additional faces generated from a new vertex in the middle of each face edge.

**Parameters:**

**float tens**

The tension setting. This value can range from -100.0 to 100.0. This value matches the parameter in the Editable Mesh user interface when tessellating faces.

**BOOL ignoreSel=FALSE**

If TRUE the entire mesh is tessellated; otherwise only the selected faces.

**Prototype:**

**void IndentSelFaces(float amount);**

**Remarks:**

This method is available in release 3.0 and later only.

Indents the selected faces, in a manner consistent with the outlining used in Bevel.

**Parameters:**

**float amount**

The amount to indent.

**Prototype:**

**void ExtrudeFaces(BOOL doFace=TRUE);**

**Remarks:**

This method is available in release 2.0 and later only.

Extrudes the selected faces. Note that this is just a topological change. The new extruded faces do not change position but are left on top of the original faces.

**Parameters:**

**BOOL doFace=TRUE**

If TRUE the faces are extruded. If FALSE then the selected edges are extruded.

**Prototype:**

**void BreakVerts(BitArray set);**

**Remarks:**

This method is available in release 2.0 and later only.

Splits the vertices specified in the BitArray so that they are only used by a single face.

**Parameters:**

**BitArray set**

This array of bits, one per vertex in the mesh. If the bit is set, the corresponding vertex in the mesh is copied as required so it is only used by a single face. If the bit is not set the vertex is ignored.

**Prototype:**

**BitArray GetIsoVerts();**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns a BitArray (of size **numVerts**), where isolated verts are selected.

**Prototype:**

**void DeleteIsoVerts();**

**Remarks:**

This method is available in release 2.0 and later only.

Deletes the vertices that aren't used by any faces.

**Prototype:**

**void CloneFaces(BitArray fset);**

**Remarks:**

This method is available in release 2.0 and later only.

Creates a copy of the faces and verts used by those faces as specified by the **BitArray** passed. If texture faces and vertex color faces are present they are cloned as well.

**Parameters:**

**BitArray fset**

There is one bit in this array for each face in the mesh. If the bit is set, the corresponding face in the mesh is cloned. If the bit is zero the face is not

cloned.

**Prototype:**

**void PolyFromFace(DWORD f, BitArray &set, float thresh, BOOL ignoreVisEdges, AdjFaceList \*af=NULL);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets bits for all faces in the same polygon with face **f**. Faces already selected in **set** will not be processed -- so if **f** is "set", nothing happens.

The definition of a polygon is all faces sharing invisible edges with edge angles below "thresh".

**Parameters:**

**DWORD f**

Specifies which face to evaluate -- the zero based index into the **faces** array.

**BitArray &set**

Specifies which faces are not processed. One bit for each face with set bits not considered.

**float thresh**

The angle in radians which is the threshold for defining a polygon. A polygon is all faces sharing invisible edges with edge angles below this angle.

**BOOL ignoreVisEdges**

If TRUE, the edge visibility is ignored but the threshold is still relevant.

**AdjFaceList \*af=NULL**

This adjacent face list can be passed if there's one handy; otherwise a new one will be computed by the method.

**Prototype:**

**void ElementFromFace(DWORD f, BitArray &set, AdjFaceList \*af=NULL);**

**Remarks:**

This method is available in release 3.0 and later only.

This method sets bits for all faces in the same "element", or connected component, with face **f**. Faces already selected in **set** will be considered

"walls" for this processing and will not be evaluated.

**Parameters:**

**DWORD f**

Specifies which face to evaluate -- the zero based index into the **faces** array.

**BitArray &set**

Specifies which faces are considered as barriers to the element and are not processed. One bit for each face with set bits not considered.

**AdjFaceList \*af=NULL**

This adjacent face list can be passed if there's one handy; otherwise a new one will be computed by the method.

**Prototype:**

```
void FindVertsUsedOnlyByFaces(BitArray & fset, BitArray & vset);
```

**Remarks:**

This method is available in release 3.0 and later only.

When faces are deleted in Edit or Editable Mesh, we often want to be able to delete the verts that are isolated by this action. This method generates the list of verts that are used only by the specified set of faces.

**Parameters:**

**BitArray & fset**

This method finds those vertices used only by the faces indicated in this BitArray.

**BitArray & vset**

This BitArray is completely overwritten with the result, and will be set to the right size (numVerts) if needed.

**Prototype:**

```
void FindOpenEdges(BitArray & edges);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method fills in a BitArray with the edges in the mesh that are "open" or "one-sided". (This is the same as the Edit Mesh "Select Open Edges"

function.)

**Parameters:**

**BitArray & edges**

This BitArray will be set to size numFaces\*3, and only the open edge bits will be set.

**Prototype:**

**void FindVertexAngles(float \*vang, BitArray \*set=NULL);**

**Remarks:**

This method is available in release 3.0 and later only.

This method calculates, for each vertex, the sum of the angles of this vertex's corner in each face it's on. So for instance, a point lying in the middle of a grid would always have vertex angle  $2*PI$ , whereas a corner of a box would only have  $3*PI/2$ .

**Parameters:**

**float \*vang**

This should be a pointer to an array of size numVerts. The vertex angle for each vertex is put in this array (in radians).

**BitArray \*set=NULL**

If non-NULL, this points to a BitArray describing which vertices to find angles of. If only some bits are set, some computation time is saved. If NULL, all vertices' angles are found.

**Prototype:**

**void SetRenderData(MeshRenderData \*p);**

**Remarks:**

Sets the mesh render data hung off this Mesh. This method and

**GetRenderData()** allow the renderer to 'hang' data on a mesh. This data can be anything the renderer wants. The data will automatically be deleted when the mesh is deleted via the **DeleteThis()** method.

**Parameters:**

**MeshRenderData \*p**

See [Class MeshRenderData](#).

**Prototype:****MeshRenderData \*GetRenderData();****Remarks:**

Returns the mesh render data hung off this Mesh. See [Class MeshRenderData](#). This method and **SetRenderData()** allow the renderer to 'hang' data on a mesh. This data can be anything the renderer wants. The data will automatically be deleted when the mesh is deleted via the **DeleteThis()** method.

**Operators:****Prototype:****Mesh& operator=(const Mesh& fromMesh);****Remarks:**

Assignment operator. Note: This operator does not copy the **rVerts** array. This means that developers who have specified normals will have to explicitly copy them after a mesh assignment is done using this operator. This is because **rVerts** are instance-specific. In general, normals are computed from smoothing groups, and hence are "generated data". In the case where normals are specified, 3ds max stores the data in the **rVerts** array so as to not waste space. The way 3ds max uses the mesh = operator assumes that the **rVerts** are not copied, and thus developers must do the copying themselves.

**Parameters:****const Mesh& fromMesh**

Specifies the mesh to copy.

**Prototype:****Mesh operator+(Mesh &mesh);****Remarks:**

Performs a boolean union operation.

**Parameters:****Mesh &mesh**

Specifies the mesh to use as the other operand of the boolean operation.

**Return Value:**

A new Mesh resulting from the boolean operation. If the operation fails an empty Mesh is returned.

**Prototype:**

**Mesh operator-(Mesh &mesh);**

**Remarks:**

Performs a boolean difference operation.

**Parameters:**

**Mesh &mesh**

Specifies the mesh to use as the other operand of the boolean operation.

**Return Value:**

A new Mesh resulting from the boolean operation. If the operation fails an empty Mesh is returned.

**Prototype:**

**Mesh operator\*(Mesh &mesh);**

**Remarks:**

Performs a boolean intersection operation.

**Parameters:**

**Mesh &mesh**

Specifies the mesh to use as the other operand of the boolean operation.

**Return Value:**

A new Mesh resulting from the boolean operation. If the operation fails an empty Mesh is returned.

**The following global functions are not part of the Mesh class:****Function:**

**void setUseVisEdge(int b);**

**Remarks:**

This is used internally.

**Function:**

```
int getUseVisEdge();
```

**Remarks:**

This is used internally.

**Boolean Operations / CombineMeshes:**

The following functions are not part of class Mesh but are available for use:

**Prototype:**

```
int CalcBoolOp(Mesh &mesh, Mesh &mesh1, Mesh &mesh2, int  
op,  
MeshOpProgress *prog = NULL, Matrix3 *tm1 = NULL,  
Matrix3 *tm2 = NULL, int whichInv = 0, int weld = TRUE);
```

**Remarks:**

Note: This method is still in the SDK, but it is now obsolete. Calls to **CalcBoolOp()** should be replaced with calls to the new **MNMesh** form of Boolean. Please see the method **MNMesh::MakeBoolean** in [Class MNMesh](#) for details.

This function stores the result of a boolean operation between **mesh1** and **mesh2** into **mesh**. This operation may be a union, intersection or difference. If **tm1** or **tm2** are non-NULL, the points of the corresponding mesh will be transformed by these matrices before the boolean operation. The mesh will be transformed back by either **Inverse(tm1)** or **Inverse(tm2)** depending on **whichInv** (a value of 0 will use **tm1**, a value of 1 will use **tm2**, unless **whichInv** is -1 in which case it will not be transformed back).

**Parameters:****Mesh &mesh**

The result of the boolean operation is stored here. **mesh = mesh1 op mesh2**.

**Mesh &mesh1**

The first operand.

**Mesh &mesh2**

The second operand.

**int op**

The boolean operation. One of the following values:

**MESHBOOL\_UNION**

**MESHBOOL\_INTERSECTION**

**MESHBOOL\_DIFFERENCE**

**MeshOpProgress \*prog = NULL**

A callback to display a progress. See [Class MeshOpProgress](#).

**Matrix3 \*tm1 = NULL**

If non-NULL then the points of mesh1 will transformed by this matrix before the boolean operation.

**Matrix3 \*tm2 = NULL**

If non-NULL then the points of mesh2 will transformed by this matrix before the boolean operation.

**int whichInv = 0**

If 0, the resulting mesh will be transformed by Inverse(tm1). If 1, the resulting mesh will be transformed by Inverse(tm2). If -1, the mesh will not be transformed back.

**int weld = TRUE**

If TRUE, the vertices of the resulting mesh are welded.

**Return Value:**

Nonzero if the operation completed successfully; otherwise zero.

**Prototype:**

```
void CombineMeshes(Mesh &mesh, Mesh &mesh1, Mesh  
&mesh2, Matrix3 *tm1=NULL, Matrix3 *tm2=NULL, int  
whichInv=0);
```

**Remarks:**

This function is available in release 2.0 and later only.

This function may be used to simply combine two meshes into one.

**Parameters:**

**Mesh &mesh**

The result of the combine operation is stored here. **mesh = mesh1+mesh2.**

**Mesh &mesh1**

The first operand.

**Mesh &mesh2**

The second operand.

**Matrix3 \*tm1 = NULL**

If non-NULL then the points of mesh1 will transformed by this matrix before the boolean operation.

**Matrix3 \*tm2 = NULL**

If non-NULL then the points of mesh2 will transformed by this matrix before the boolean operation.

**int whichInv=0**

If 0, the resulting mesh will be transformed by Inverse(tm1). If 1, the resulting mesh will be transformed by Inverse(tm2). If -1, the mesh will not be transformed back.

## **Sub-Object Selection Color Control**

The following functions are not part of class Mesh but are available for use:

**Prototype:**

**void SetSubSelColor(Point3 \*clr);**

**Remarks:**

This methods sets the color used for sub-object selection.

**Parameters:**

**Point3 \*clr**

The color to use.

**Prototype:**

**Point3 GetSubSelColor();**

**Remarks:**

Returns the color used for sub-object selection.

## Class DADMgr

See Also: [Class ReferenceTarget](#), [List of Super Class IDs](#).

```
class DADMgr : public InterfaceServer
```

### Description:

Drag and drop functionality has been expanded to include all map and material buttons<sup>¾</sup>including those in the non-standard materials, plus most cases of bitmap buttons. As a result, whenever you see a button representing a material or map you can drag the button over a like button to display the Swap/Copy/Cancel dialog. Likewise, you can drag any materials or maps from the modeless version of the Materials/Maps Browser.

The drag-and-drop functions distinguish between material maps and bitmaps. A bitmap is an image file, such as a .tga, or .jpg. A map is an image used by the Materials Editor. It might consist of an image file, but could just as easily be a parametric image, such as Checkers or Noise, or it could be a map tree consisting of several different types of maps or bitmaps. Users can drag any map slot or button to any other map slot or button<sup>¾</sup>including the sample slots. Users can drag the Bitmap button in the Bitmap Parameters rollout to the Bitmap button in the Image area of the Displace modifier, and vice-versa.

Users can drag **from**:

- Sample slots
- Browser lists (text or iconic)
- The sample-sphere preview window in the Browser.
- Material map buttons, including:
  - The buttons in the Maps rollout
  - The shortcut map buttons
  - Any map buttons at any level
- Submaterial buttons, such as those found in the Multi/Subobject material
- Projector light map button
- Environment background map button
- Fog Color and Opacity maps buttons

Users can drag **to**:

- Objects in the viewports
- The Type button in the Materials Editor from the Browser.

- All of the items in the FROM list, with this exception: You can only drag to the Browser when it displays the material library.

All methods of this class are implemented by the plug-in. For developers of plug-in textures and materials see [Class TexDADMgr](#), [Class MtlDADMgr](#). These classes provide implementations of these methods and the objects can simply be used.

### Methods:

#### Prototype:

**virtual SClass\_ID GetDragType(HWND hwnd, POINT p)=0;**

#### Remarks:

This method is called on the item that supports drag and drop to see what (if anything) can be dragged from the point **p**. This method returns a super class id to indicate the type of item that can be dragged away. If it does not support anything being dragged from the specified point a SClass\_ID of **0** should be returned.

#### Parameters:

**HWND hwnd**

The source window handle

**POINT p**

The screen point (relative to the window upper left as 0,0).

#### Prototype:

**virtual BOOL IsNew(HWND hwnd, POINT p, SClass\_ID type);**

#### Remarks:

If the method **GetInstance()** creates a new instance every time it is called, then the this method should return TRUE. Otherwise it should return FALSE. This prevents **GetInstance()** from being called repeatedly as the drag progresses.

#### Parameters:

**HWND hwnd**

The source window handle.

**POINT p**

The point to drag from.

**SClass\_ID type**

The super class ID to create.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual ReferenceTarget *GetInstance(HWND hwnd, POINT p,  
SClass_ID type)=0;
```

**Remarks:**

This method should return a pointer to the drag source.

**Parameters:**

**HWND hwnd**

The source window where the mouse down occurred.

**POINT p**

The point to drag from (position within **hwnd**).

**SClass\_ID type**

The super class ID of the item to create.

**Prototype:**

```
virtual BOOL OkToDrop(ReferenceTarget *dropThis, HWND  
hfrom, HWND hto, POINT p, SClass_ID type, BOOL isNew =  
FALSE)=0;
```

**Remarks:**

This method is called on potential dropee to see if can accept the specified type at the specified point.

**Parameters:**

**ReferenceTarget \*dropThis**

A pointer to the item to check.

**HWND hfrom**

The window handle of the source.

**HWND hto**

The window handle of the destination.

**POINT p**

The point to check.

**SClass\_ID type**

The super class ID of **dropThis**.

**BOOL isNew = FALSE**

TRUE if the item is a new instance; otherwise FALSE.

**Return Value:**

TRUE if the specified item can be dropped; otherwise FALSE.

**Prototype:**

```
virtual HCURSOR DropCursor(ReferenceTarget *dropThis,  
HWND hfrom, HWND hto, POINT p, SClass_ID type, BOOL  
isNew = FALSE);
```

**Remarks:**

This method is called on a potential target to allow it to substitute custom cursors. It returns the handle for the custom cursor to use (or NULL to ignore).

**Parameters:**

**ReferenceTarget \*dropThis**

The pointer to the item to check.

**HWND hfrom**

The window handle of the source.

**HWND hto**

The window handle of the destination.

**POINT p**

The point to check.

**SClass\_ID type**

The super class ID of **dropThis**.

**BOOL isNew = FALSE**

TRUE if the item is a new instance; otherwise FALSE.

**Default Implementation:**

```
{ return NULL;}
```

**Prototype:**

```
virtual int SlotOwner();
```

**Remarks:**

Returns a predefined value to indicate the source of the drag.

**Return Value:**

One of the following values:

**OWNER\_MEDIT\_SAMPLE**

From a materials editor sample slot.

**OWNER\_NODE**

From a node in the scene.

**OWNER\_MTL\_TEX**

From a button in a material or texture.

**OWNER\_SCENE**

From a button in a light, modifier, atmospheric effect, etc.

**OWNER\_BROWSE\_NEW**

From the browser in the new category.

**OWNER\_BROWSE\_LIB**

From the browser in the library category.

**OWNER\_BROWSE\_MEDIT**

From the browser in the materials editor category.

**OWNER\_BROWSE\_SCENE**

From the browser in the scene category.

**Default Implementation:**

```
{ return OWNER_MTL_TEX; }
```

**Prototype:**

```
virtual void Drop(ReferenceTarget *dropThis, HWND hwnd,  
POINT p, SClass_ID type)=0;
```

**Remarks:**

This is the method called to actually process the drop operation. This routine is called on the target with the pointer returned by the source's

**GetInstance()**, or possibly a clone of it as the **dropThis**.

**Parameters:**

**ReferenceTarget \*dropThis**

A pointer to the item to drop.

**HWND hwnd**

The destination window handle (where the mouse was released).

**POINT p**

The destination point (within **hwnd**).

**SClass\_ID type**

The type of object being dropped -- the super class ID of **dropThis**.

**Prototype:**

```
virtual void SameWinDragAndDrop(HWND h1, POINT p1,  
POINT p2);
```

**Remarks:**

This method is called when the source and target WINDOW are the same.

**Parameters:**

**HWND h1**

The source/target window handle.

**POINT p1**

The source point.

**POINT p2**

The target point.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual BOOL LetMeHandleLocalDAD();
```

**Remarks:**

This lets the manager know whether to call **LocalDragAndDrop()** if the same DADMgr is handling both the source and target windows, or just ignore this condition. Return TRUE if **LocalDragAndDrop()** should be called;

otherwise FALSE.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual void LocalDragAndDrop(HWND h1, HWND h2, POINT  
p1, POINT p2);
```

**Remarks:**

This is called if the same **DADMgr** is handling both the source and target windows, if **LetMeHandleLocalDAD()** returned TRUE.

**Parameters:**

**HWND h1**

The window handle.

**HWND h2**

The window handle.

**POINT p1**

The drag source point.

**POINT p2**

The drop destination point.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BOOL AutoTooltip();
```

**Remarks:**

If this method returns TRUE, then Custom Buttons that use this DAD Manager will automatically support a tooltip that matches the button text. Note that this method will only show a tooltip when the button text is too long and thus exceeds the button size.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual BOOL CopyOnly(HWND hwnd, POINT p, SClass\_ID type);**

**Remarks:**

If a drag source doesn't want any references being made to the instance returned, then this method should return TRUE: it will force a copy to be made; otherwise return FALSE.

**Parameters:**

**HWND hwnd**

The source window handle.

**POINT p**

The source point (within **hwnd**).

**SClass\_ID type**

The type of object being dragged.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual BOOL AlwaysSendButtonMsgsOnDrop();**

**Remarks:**

Normally the mouse down and mouse up messages are not sent to the source window when doing drag and drop, but if you need them, return TRUE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual BOOL OkToDropInstance(ReferenceTarget \*dropThis, HWND hfrom, HWND hto, POINT p, SClass\_ID type);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is called on potential target to see if can instance "dropThis" at

the specified point. Returns TRUE if it is okay to drop the specified item and FALSE if not.

**Parameters:**

**ReferenceTarget \*dropThis**

The pointer to the item to check.

**HWND hfrom**

The window handle of the source.

**HWND hto**

The window handle of the destination.

**POINT p**

The point to check.

**SClass\_ID type**

The super class ID of **dropThis**.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0);
```

**Remarks:**

This method is available in release 3.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

**Parameters:**

**int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value (defined uniquely for each **cmd**).

**Default Implementation:**

```
{ return 0; }
```

## Class TexDADMgr

See Also: [Class DADMgr](#), [Class ParamDlg](#).

```
class TexDADMgr : public DADMgr
```

### **Description:**

This class is available in release 2.0 and later only.

Use this class to provide drag and drop functionality for materials sub-  
Texmaps. It provides implementations of the methods of **DADMgr**. If this  
class is used the **ParamDlg** method **FindSubTexFromHWND()** must be  
implemented.

## Class MtlDADMgr

See Also: [Class DADMgr](#).

class MtlDADMgr: public DADMgr

### Description:

All methods of this class are implemented by the system.

Use this class to implement drag and drop functionality for materials sub-materials. . It provides implementations of the methods of **DADMgr**. If this class is used the **ParamDlg** method **FindSubMtlFromHWND()** must be implemented.

## Class DADBitmapCarrier

See Also: [Class ReferenceTarget](#).

```
class DADBitmapCarrier: public ReferenceTarget
```

### Description:

This class is available in release 2.0 and later only.

It is used to provide drag and drop functionality for Bitmaps. See the following global function for getting a pointer to the instances of this class.

All methods of this class are implemented by the system.

### Function:

```
DADBitmapCarrier *GetDADBitmapCarrier();
```

### Remarks:

This global function returns a pointer to the **BitmapCarrier**. Note that there is only two of these: one for the source, and one for the destination.

Developers should not delete these. Sample code using this class is available in **\MAXSDK\SAMPLES\MATERIALS\BMTEX.CPP**.

### Methods:

#### Prototype:

```
void SetName(TSTR &nm);
```

#### Remarks:

Sets the name of the bitmap carrier.

#### Parameters:

```
TSTR &nm
```

The name to set.

#### Prototype:

```
TSTR& GetName();
```

#### Remarks:

This method is available in release 2.0 and later only.

Returns the name of the bitmap carrier.

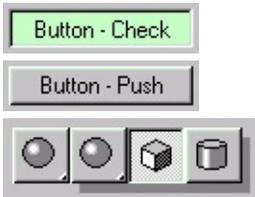
## Class ICustButton

See Also: [Class ICustomControl](#), [Custom Controls](#), [Class ICustToolbar](#), [Class FlyOffData](#), [Class DADMgr](#), [Class MAXBmpFileIcon](#).

```
class ICustButton : public ICustomControl
```

### Description:

Custom buttons may be one of two different forms. A Check button (which stays pressed in until the user clicks on it again), or a Push button (which pops back out as soon as it is released). Buttons may be implemented as a Fly offs. A fly off offers several alternative buttons which fly out from the button after it is press and held briefly.



The buttons may contain text or graphic images. Fly off buttons only use graphic images. The plug-in developer has control over the appearance of the button in each of its four states (Enabled&Out, Enabled&In, Disabled&Out, Disabled&In).

Note: When the user presses a button a **WM\_MENUSELECT** message is sent so that the client can display a status prompt describing the function of the tool. The **fuFlags** parameter is set to this value: **CMF\_TOOLBUTTON**.

In 3dsmax version 4.0 you can remove borders from an ICustButton;

```
ICustButton *cb = ();
```

```
cb->Execute(I_EXE_CB_NO_BORDER);
```

To initialize the pointer to the control call:

### Prototype:

```
ICustButton *GetICustButton(HWND hCtrl);
```

To release the control call:

### Prototype:

```
ReleaseICustButton(ICustButton *ics);
```

The value to use in the Class field of the Custom Control Properties dialog is:

**CustButton**

**Methods:****Prototype:**

**virtual void GetText(TCHAR \*text, int ct)=0;**

**Remarks:**

This retrieves the text displayed by the button.

**Parameters:**

**TCHAR \*text**

Storage for the text to retrieve.

**int ct**

Specifies the maximum length of the string returned.

**Prototype:**

**virtual void SetText(TCHAR \*text)=0;**

**Remarks:**

This specifies the text displayed by the button.

**Parameters:**

**TCHAR \*text**

The text to be displayed by the button.

**Prototype:**

**virtual void SetImage(HIMAGELIST hImage, int iOutEn, int iInEn, int iOutDis, int iInDis, int w, int h)=0;**

**Remarks:**

This method is used to establish the images used for the buttons.

**Parameters:**

**HIMAGELIST hImage**

The image list. An image list is a collection of same-sized images, each of which can be referred to by an index. Image lists are used to efficiently manage large sets of icons or bitmaps in Windows. All images in an image list are contained in a single, wide bitmap in screen device format. An image list may also include a monochrome bitmap that contains masks used to draw images transparently (icon style). The Windows API provides image list

functions, which enable you to draw images, create and destroy image lists, add and remove images, replace images, and merge images.

The next four parameters (**iOutEn**, **iInEn**, **iOutDis**, **iInDis**) are indices into the image list. They indicate which images to use for each of the four possible button states. You may specify a unique image for each one of these states by passing a different index for each state. Or you may supply a single image to be used for all the states by specifying the same index four times.

**int iOutEn**

Out&Enabled.

**int iInEn**

In&Enabled.

**int iOutDis**

Out&Disabled.

**int iInDis**

In&Disabled.

**int w**

The width of the button image.

**int h**

The height of the button image.

**Prototype:**

**virtual void SetIcon(MaxBmpFileIcon\* pIcon, int w, int h) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This sets the icon image used for a button.

**Parameters:**

**MaxBmpFileIcon\* pIcon**

Points to the icon.

**int w**

The width of the button image.

**int h**

The height of the button image.

**Prototype:**

```
virtual void SetInIcon(MaxBmpFileIcon* pInIcon, int w, int h) =  
0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This sets the icon image used when a button is pressed.

**Parameters:**

**MaxBmpFileIcon\* pInIcon**

Points to the icon.

**int w**

The width of the button image.

**int h**

The height of the button image.

**Prototype:**

```
virtual void SetType(CustButType type)=0;
```

**Remarks:**

This method sets the button type.

**Parameters:**

**CustButType type**

One of the following values:

**CBT\_PUSH**

A Push button pops back out as soon as it is released.

**CBT\_CHECK.**

A Check button stays pressed in until the user clicks on it again.

**Prototype:**

```
virtual void SetFlyOff(int count,FlyOffData *data,int timeOut,int  
init,int dir=FLY_VARIABLE)=0;
```

**Remarks:**

This method sets the button to work as a fly off control.

**Parameters:**

**int count**

The number of buttons in the fly off.

**FlyOffData \*data**

An array of instances of the class **FlyOffData** . This class uses four indices into the image list to describe the button in each of the possible states: Out&Enabled, In&Enabled, Out&Disabled and In&Disabled.

In the simple case, where all the buttons have the same image, you can do the following:

```
FlyOffData fod[3] = { // A three button flyoff  
  { 0,0,0,0 }, // The first button uses a single image.  
  { 1,1,1,1 }, // So does the second button...  
  { 2,2,2,2 }, // So does the third...  
};
```

Each button will use the same images regardless of its pressed in / disabled state. Note the button is automatically drawn pushed in (i.e. shaded lighter) when the user is dragging the cursor over the button, but the actual image on the button is not changed.

If you require different images for these states, supply different indices into the image list for each. See the sample program

**\MAXSDK\SAMPLES\HOWTO\CUSTCTRL\CUSTCTRL.CPP**  
for an example of how this is done.

**int timeOut**

This is the time in milliseconds the button must be held pressed before the fly off appears. You may specify 0 if you want the buttons to fly off immediately. To retrieve the value that 3ds max uses internally for its flyoffs use a method of [Class Interface](#) called **GetFlyOffTime()**. This returns a value in milliseconds.

**int init**

This is the initial button displayed.

**int dir=FLY\_VARIABLE**

This parameter is optional. It is used to indicate which direction the buttons should fly off. The choices for direction are:

**FLY\_VARIABLE**

The default. The system will determine the direction of the fly off.

**FLY\_UP**

The buttons fly off above.

**FLY\_DOWN**

The buttons fly off beneath.

**FLY\_HVARIABLE**

The buttons will fly off either left or right with the system determining the direction.

**FLY\_LEFT**

The buttons fly off to the left.

**FLY\_RIGHT**

The buttons fly off to the right.

**Prototype:**

```
virtual void SetCurFlyOff(int f,BOOL notify=FALSE)=0;
```

**Remarks:**

This method establishes which button is displayed by passing its index.

**Parameters:****int f**

The index of the flyoff button to display.

**BOOL notify=FALSE**

This indicates if the call to this method should notify the dialog proc. If TRUE it is notified; otherwise it isn't.

**Prototype:**

```
virtual int GetCurFlyOff()=0;
```

**Remarks:**

Returns the index of the button which is currently displayed.

**Prototype:**

```
virtual BOOL IsChecked()=0;
```

**Remarks:**

Determines if the button is checked. This method returns TRUE if the check

button is currently in the In state (i.e. checked) and FALSE otherwise.

**Prototype:**

**virtual void SetCheck( BOOL checked )=0;**

**Remarks:**

Passing TRUE to this method sets the button to the In or checked state.

**Parameters:**

**BOOL checked**

If TRUE the button is set to the checked state; if FALSE the button is unchecked.

**Prototype:**

**virtual void SetCheckHighlight( BOOL highlight )=0;**

**Remarks:**

This method controls if the check button is displayed in the highlight color when pressed in.

**Parameters:**

**BOOL highlight**

TRUE if you want the button to use the highlight color; otherwise pass FALSE.

**Prototype:**

**virtual void SetButtonDownNotify(BOOL notify)=0;**

**Remarks:**

Specifies if messages are sent when the user clicks or releases the button. If this method is called with TRUE, a message is sent immediately whenever the button is pressed down or released. The message **BN\_BUTTONDOWN** is sent on button down and **BN\_BUTTONUP** is sent when the button is released. The **BN\_BUTTONUP** message is sent even if the button is released outside the button rectangle.

**Parameters:**

**BOOL notify**

TRUE if notification should be send by the button; FALSE if notifications should not be sent.

**Prototype:**

**virtual void SetRightClickNotify(BOOL notify)=0;**

**Remarks:**

Specifies if messages are sent when the user right clicks the button.

**Parameters:**

**BOOL notify**

If TRUE, the **BN\_RIGHTCLICK** message is sent whenever the users right clicks on the button. If FALSE no message are sent on right clicks.

**Prototype:**

**virtual void SetHighlightColor(COLORREF clr)=0;**

**Remarks:**

This methods sets the highlight color for the check button.

**Parameters:**

**COLORREF clr**

The color for the button. It may be specified using the RGB macro, for example: **SetHighlightColor(RGB(0,0,255));**

There are several pre-defined colors which may be used. These are:

**RED\_WASH, BLUE\_WASH** and **GREEN\_WASH**. **GREEN\_WASH** is the standard color used for check buttons in 3ds max that instigate a command mode. While the command mode is active the button should be displayed in **GREEN\_WASH**. When the mode is finished the button should be returned to normal.

**Prototype:**

**virtual COLORREF GetHighLightColor()=0;**

**Remarks:**

This methods returns the highlight color for the check button.

**Prototype:**

**virtual void SetTooltip(BOOL onOff, LPSTR text)=0;**

**Remarks:**

This method allows a developer to add tooltip text to single custom buttons.

**Parameters:****BOOL onOff**

TRUE to turn the tooltip on; FALSE to turn it off.

**LPSTR text**

This may be one of two things:

The tooltip text (as a **TCHAR \***)

This is simply the text to show up in the tooltip.

The symbol **LPSTR\_TEXTCALLBACK**

This indicates a callback will be used. If this is specified, the parent window of the button will get the usual tooltip "need text" notify message (**TTN\_NEEDTEXT**).

**Prototype:**

**virtual void SetDADMgr(DADMgr \*dad)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the drag and drop manager for this button control.

**Parameters:****DADMgr \*dad**

A pointer to the drag and drop manager to set.

**Prototype:**

**virtual DADMgr \*GetDADMgr()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the drag and drop manager for this button control.

**Prototype:**

**virtual void SetMacroButtonData(MacroButtonData \*md)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the macro data for this button.

**Parameters:**

**MacroButtonData \*md**

The data to set. See [Class MacroButtonData](#).

**Prototype:**

**virtual MacroButtonData \*GetMacroButtonData()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to any macro button data for this button. See [Class MacroButtonData](#).

**Prototype:**

**virtual void SetDisplayProc(PaintProc \*proc)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the callback object used to display the button.

Note: This method is only available in 3D Studio VIZ (not MAX).

**Parameters:**

**PaintProc \*proc**

Points to the callback object for displaying the button.

Note: **typedef LRESULT CALLBACK PaintProc(  
HDC hdc, Rect rect, BOOL in, BOOL checked, BOOL  
enabled);**

## Class IDADWindow

See Also: [Class ICustomControl](#), [Class DADMgr](#), [Custom Controls](#).

```
class IDADWindow : public ICustomControl
```

### Description:

This class is available in release 2.0 and later only.

This is a new type of custom control used to provide drag and drop to and from things other than Custom Buttons. An example of an item that uses this control is a sample sphere window in the Material Editor.

To initialize the pointer to the control call:

### Prototype:

```
IDADWindow *GetIDADWindow(HWND hWnd);
```

To release the control call:

### Prototype:

```
void ReleaseIDADWindow(IDADWindow *idw);
```

The value to use in the Class field of the Custom Control Properties dialog is:

### **DragDropWindow**

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
virtual void SetDADMgr(DADMgr *dadMgr)=0;
```

#### Remarks:

Set the drag and drop manager for this control.

#### Parameters:

```
DADMgr *dadMgr
```

A pointer to the drag and drop manager for this control.

#### Prototype:

```
virtual DADMgr *GetDADMgr()=0;
```

#### Remarks:

Returns a pointer to the drag and drop manager for this control.

**Prototype:**

```
virtual void SetWindowProc(WindowProc *proc)=0;
```

**Remarks:**

This method establishes a window proc that is called to handle all the normal processing after the drag and drop processing is done.

**Parameters:**

```
WindowProc *proc
```

The window proc. Note the following typedef:

```
typedef LRESULT CALLBACK WindowProc(HWND hwnd,  
UINT message, WPARAM wParam, LPARAM lParam);
```

```
#define SLIDERWINDOWCLASS _T("SliderControl")
```

```
// LOWORD(wParam) = ctrlID,
```

```
// HIWORD(wParam) = TRUE if user is dragging the slider  
interactively.
```

```
// lParam = pointer to ISliderControl
```

```
#define CC_SLIDER_CHANGE WM_USER + 611
```

```
// LOWORD(wParam) = ctrlID,
```

```
// lParam = pointer to ISliderControl
```

```
#define CC_SLIDER_BUTTONDOWN WM_USER + 612
```

```
// LOWORD(wParam) = ctrlID,
```

```
// HIWORD(wParam) = FALSE if user cancelled - TRUE  
otherwise
```

```
// lParam = pointer to ISliderControl
```

```
#define CC_SLIDER_BUTTONUP WM_USER + 613
```

# Class Spline3D

See Also: [Class SplineKnot](#), [Class PolyLine](#), [Structure Knot](#).

class Spline3D

## Description:

General-purpose 3D spline class. The **BezierShape** class has a list of these splines that make up the bezier shape. Methods of this class are used to access the properties of the spline. All methods of this class are implemented by the system.

## Method Groups:

The following hyperlinks take you to the start of groups of related methods within the class:

[Constructors / Destructor](#)

[NewSpline\(\) / InvalidateGeomCache\(\)](#)

[KnotCount\(\), Flags\(\), Segments\(\), Closed\(\)](#)

[AddKnot\(\), DeleteKnot\(\), GetKnot\(\), SetKnot\(\)](#)

[Create\(\) / StartInsert\(\)](#)

[Get/SetKnotType\(\) / Get/SetLineType\(\)](#)

[ComputeBezPoints\(\)](#)

[InterpBezier\(\) / InterpBezier3D\(\) / TangentBezier3D\(\) / TangentCurve3D\(\)](#)

[IsAuto\(\) / IsBezierPt\(\) / IsCorner\(\)](#)

[GetBBox\(\) / Transform\(\)](#)

[SetClosed\(\) / SetOpen\(\)](#)

[Reverse\(\) / Append\(\) / Prepend\(\)](#)

[Get/SetInVec\(\) / Get/SetOutVec\(\)](#)

[Get/SetVert\(\) / Verts\(\) / Get/SetKnotPoint\(\)](#)

[IsClockWise\(\) / SelfIntersects\(\) / IntersectsSpline\(\) / SurroundsPoint\(\) /](#)

[SplineLength\(\)](#)

[MakePolyLine\(\)](#)

[Dump\(\)](#)

[Internal Methods / Methods that should not be used by plug-in developers](#)

[Operators](#)

## Friend Classes:

**friend class BezierShape;**

**friend class SplineShape;**

**Methods:**

## **Constructors / Destructor**

**Prototype:**

```
Spline3D(int itype = KTYPE_CORNER,int dtype =  
KTYPE_BEZIER,  
int ptype = PARM_UNIFORM);
```

**Remarks:**

Constructor. The data members are initialized to the values passed.

**Prototype:**

```
~Spline3D();
```

**Remarks:**

Destructor.

## **NewSpline() / InvalidateGeomCache()**

**Prototype:**

```
void NewSpline();
```

**Remarks:**

This method clears out the spline. It frees the knots attributes array and the bezier points array.

**Prototype:**

```
void InvalidateGeomCache();
```

**Remarks:**

This method makes sure the shape has flushed out any cached data it may have had.

## **KnotCount(), Flags(), Segments(), Closed()**

**Prototype:**

**inline int KnotCount()**

**Remarks:**

Returns the Knot (point) count.

**Prototype:**

**inline int Flags();**

**Remarks:**

Returns the private spline flags.

**SPLINE\_CLOSED**

This indicates if the spline is closed or not.

**Prototype:**

**inline int Segments();**

**Remarks:**

Returns the number of line segments in the spline. For example if you have a 4 knot spline that is open you'll get 3 segments.

**Prototype:**

**inline int Closed();**

**Remarks:**

Returns the closed status. Nonzero if closed; otherwise zero.

**AddKnot(), DeleteKnot(), GetKnot, SetKnot()**

**Prototype:**

**int AddKnot(SplineKnot &k,int where = -1);**

**Remarks:**

Add a knot to the spline at the specified location.

**Parameters:**

**SplineKnot &k**

The knot to add.

**int where = -1**

The location to add the knot. **where** < 0 indicates the end of the spline.

**Return Value:**

Nonzero on success; otherwise zero.

**Prototype:**

**int DeleteKnot(int where);**

**Remarks:**

Delete the specified knot.

**Parameters:**

**int where**

The location of the knot to delete.

**Return Value:**

Nonzero if the knot was deleted; otherwise zero.

**Prototype:**

**void SetKnot(int i, SplineKnot &k);**

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the 'i-th' knot object which contain the knot point, in and out vectors, knot and line types and auxiliary values.

**Parameters:**

**int i**

Specifies the knot to set.

**SplineKnot &k**

The knot to set.

**Prototype:**

**SplineKnot GetKnot(int i);**

**Remarks:**

This method is available in release 2.0 and later only.

This method gets the 'i-th' knot object which contain the knot point, in and out vectors, knot and line types and auxiliary values.

**Parameters:**

**int i**

Specifies the knot to retrieve.

## **Create() / StartInsert()**

**Prototype:**

```
int Create(ViewExp *vpt,int msg, int point, int flags, IPoint2 m,  
Matrix3* mat, IObjParam *ip=NULL);
```

**Remarks:**

This method is used internally by the free form line object **SPLINE.CPP**. This method allows the user to use the mouse to create a line. See the sample code in **\MAXSDK\SAMPLES\OBJECTS\SPLINE.CPP** for an example of this method in use.

New for 3ds max 2.0 is an additional parameter is a pointer to an **IObjParam** object, which is used to access the **DisplayTempPrompt()** mechanism. The parameter is optional; omitting it allows the spline operation to work as in 3ds max 1.x. Adding the parameter causes the spline to display the delta, distance and angle of the current segment being edited.

**Prototype:**

```
int StartInsert(ViewExp *vpt,int msg, int point, int flags,  
IPoint2 theP, Matrix3* mat, int where );
```

**Remarks:**

This method is used internally by the free form line object **SPLINE.CPP**. See the sample code in **\MAXSDK\SAMPLES\OBJECTS\SPLINE.CPP** for an example of this method in use.

## **Get/SetKnotType() / Get/SetLineType()**

**Prototype:**

**inline int GetKnotType(int index);**

**Remarks:**

Returns the knot type from the specified knot. See [List of Spline Knot Types](#).

**Parameters:**

**int index**

The index of the knot type to return.

**Prototype:**

**int SetKnotType(int index,int type);**

**Remarks:**

Sets the knot type of the specified knot.

**Parameters:**

**int index**

The knot to set.

**int type**

See [List of Spline Knot Types](#).

**Return Value:**

Nonzero if set; otherwise zero.

**Prototype:**

**inline int GetLineType(int index);**

**Remarks:**

Returns the type of line segment between knots for the specified segment.

**Parameters:**

**int index**

The index of the segment whose line type to return.

**Return Value:**

See [List of Spline Line Types](#).

**Prototype:**

**int SetLineType(int index,int type);**

**Remarks:**

Sets the line type of the specified segment.

**Parameters:****int index**

The index of the segment.

**int type**

See [List of Spline Line Types](#).

**Return Value:**

Nonzero if set; otherwise zero.

**ComputeBezPoints()****Prototype:**

```
void ComputeBezPoints();
```

**Remarks:**

This method should be called whenever you finish changing points on the spline. This updates all the information internal to the spline needed to calculate all the bezier handles.

**InterpBezier() / InterpBezier3D() / TangentBezier3D() /  
TangentCurve3D()****Prototype:**

```
Point2 InterpBezier(IPoint2 *bez, float t);
```

**Remarks:**

Developers should use **InterpBezier3D()** below instead of this method and just use the x and y values returned.

**Prototype:**

```
Point3 InterpBezier3D(int segment, float t, int  
ptype=SPLINE_INTERP_SIMPLE);
```

**Remarks:**

This method returns a point interpolated on a segment between two knots.

**Parameters:****int segment**

The index of the segment to interpolate.

**float t**

A value in the range of 0.0 to 1.0. 0 is the first knot and 1 is the second knot.

**int ptype=SPLINE\_INTERP\_SIMPLE**

The spline type to use.

**Return Value:**

The interpolated point.

**Prototype:**

```
Point3 InterpCurve3D(float u, int  
ptype=SPLINE_INTERP_SIMPLE);
```

**Remarks:**

This method returns a point interpolated on the entire curve. This method returns a point but you don't know which segment the point falls on. Typically the method **InterpBezier3D()** will give better control of the curve as it interpolates a bezier segment.

**Parameters:****float u**

A value in the range of 0.0 to 1.0 for the entire curve.

**int ptype=SPLINE\_INTERP\_SIMPLE**

The spline type to use.

**Return Value:**

The interpolated point.

**Prototype:**

```
Point3 TangentBezier3D(int segment, float t, int  
ptype=SPLINE_INTERP_SIMPLE);
```

**Remarks:**

This method returns a tangent vector interpolated on a segment between two knots.

**Parameters:****int segment**

The index of the segment.

**float t**

A value in the range of 0.0 to 1.0. 0 is the first knot and 1 is the second knot.

**int ptype=SPLINE\_INTERP\_SIMPLE**

The spline type to use.

**Return Value:**

The tangent vector.

**Prototype:****Point3 TangentCurve3D(float u, int  
ptype=SPLINE\_INTERP\_SIMPLE);****Remarks:**

This method returns a tangent vector interpolated on the entire curve.

**Parameters:****float u**

A value in the range of 0.0 to 1.0 for the entire curve.

**int ptype=SPLINE\_INTERP\_SIMPLE**

The spline type to use.

**Return Value:**

The tangent vector.

**IsAuto() / IsBezierPt() / IsCorner()****Prototype:****int IsAuto(int i);****Remarks:**Returns nonzero if the knot type is **KTYPE\_AUTO** otherwise zero.**Parameters:****int i**

The index of the knot.

**Prototype:**

**int IsBezierPt(int i);**

**Remarks:**

Returns nonzero if the knot type is **KTYPE\_BEZIER** otherwise zero.

**Parameters:**

**int i**

The index of the knot.

**Prototype:**

**int IsCorner(int i);**

**Remarks:**

Returns nonzero if the knot type is **KTYPE\_CORNER** otherwise zero.

**Parameters:**

**int i**

The index of the knot.

## **GetBBox() / Transform()**

**Prototype:**

**void GetBBox(TimeValue t, Matrix3& tm, Box3& box);**

**Remarks:**

Returns the bounding box of the curve in the space specified by the **tm**.

**Parameters:**

**TimeValue t**

This parameter is not used.

**Matrix3& tm**

The **tm** to transform the points by prior to computing the bounding box.

**Box3& box**

The bounding box

**Prototype:**

**void Transform(Matrix3 \*tm);**

**Remarks:**

This method may be used to transform the points of the spline into another space defined by the specified transformation matrix.

**Parameters:**

**Matrix3 \*tm**

The transformation matrix.

**SetClosed / SetOpen()****Prototype:**

**int SetClosed(int flag = 1);**

**Remarks:**

This method may be used to close or open the spline. If the optional parameter is not specified it is closed.

**Parameters:**

**int flag = 1**

Nonzero to close; zero to open.

**Return Value:**

Nonzero if changed, zero if not.

**Prototype:**

**int SetOpen();**

**Remarks:**

Sets the spline to open.

**Return Value:**

Nonzero if changed, zero if not.

**Get/SetInVec() / Get/SetKnotPoint() Get/SetOutVec()****Prototype:**

**Point3 GetInVec(int i);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to retrieve the 'i-th' in vector position in absolute coordinates.

**Parameters:**

**int i**

The vector position to retrieve.

**Prototype:**

**void SetInVec(int i, const Point3 &p);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to set the 'i-th' in vector position in absolute coordinates.

**Parameters:**

**int i**

The position to alter.

**const Point3 &p**

The value to set in absolute coordinates.

**Prototype:**

**Point3 GetRelInVec(int i);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to retrieve the 'i-th' in vector position relative to the knot point.

**Parameters:**

**int i**

The position to retrieve.

**Prototype:**

**void SetRelInVec(int i, const Point3 &p);**

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the 'i-th' bezier in vector position relative to the knot point.

**Parameters:**

**int i**

The vector to set.

**const Point3 &p**

The vector data to set, relative to the knot point.

**Prototype:**

**Point3 GetKnotPoint(int i);**

**Remarks:**

This method is available in release 2.0 and later only.

This methods returns the 'i-th' knot point.

**Parameters:**

**int i**

Specifies which knot point to get.

**Prototype:**

**void SetKnotPoint(int i, const Point3 &p);**

**Remarks:**

This method is available in release 2.0 and later only.

This methods sets the 'i-th' knot point to the specified value.

**Parameters:**

**int i**

Specifies which knot point to set.

**const Point3 &p**

The value to set.

**Prototype:**

**Point3 GetOutVec(int i);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to retrieve the 'i-th' out vector position in absolute coordinates.

**Parameters:****int i**

Specifies which out vector point to get.

**Prototype:****void SetOutVec(int i, const Point3 &p);****Remarks:**

This method is available in release 2.0 and later only.

This method is used to set the 'i-th' out vector position in absolute coordinates.

**Parameters:****int i**

Specifies which out vector point to get.

**const Point3 &p**

The out vector to set in absolute coordinates.

**Prototype:****Point3 GetRelOutVec(int i);****Remarks:**

This method is available in release 2.0 and later only.

This method is used to retrieve the 'i-th' out vector position relative to the knot point.

**Parameters:****int i**

Specifies the point to get.

**Prototype:****void SetRelOutVec(int i, const Point3 &p);****Remarks:**

This method is available in release 2.0 and later only.

This method is used to set the 'i-th' out vector position relative to the knot point.

**Parameters:**

**int i**

Specifies the point to set.

**const Point3 &p**

The out vector position to set relative to the knot point.

**Get/SetVert() / Verts() / KnotPoint()**

**Prototype:**

**inline Point3& GetVert(int i);**

**Remarks:**

Each control point is made up of three points. The in vector coming off the bezier control point, the knot point itself, and the out vector. There are these three points for every control point. This method will return any item in this list.

**Parameters:**

**int i**

The index into the vertex list.

**Prototype:**

**inline void SetVert(int i, const Point3& p);**

**Remarks:**

This method should not be used.

**Prototype:**

**inline int Verts();**

**Remarks:**

Returns the number of vertices. This is always the number of knots times 3.

**Prototype:**

**int GetAux(int knot);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the auxiliary data associated with the specified knot. This is used internally for tracking topological changes to the spline during editing.

Developers can use it for temporary purposes but it will be altered by the EditableSpline (SplineShape) code.

**Parameters:**

**int knot**

The knot to retrieve the data from.

**Prototype:**

**void SetAux(int knot, int value);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the first integer auxiliary data associated with the specified knot.

**Parameters:**

**int knot**

Specifies the knot to whose auxiliary data is set.

**int value**

The value to set.

**Prototype:**

**int GetAux2(int knot);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the second integer auxiliary data associated with the specified knot

**Parameters:**

**int knot**

The knot to retrieve the data from.

**Prototype:**

**void SetAux2(int knot, int value);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the second integer auxiliary data associated with the specified knot.

**Parameters:**

**int knot**

Specifies the knot to whose auxiliary data is set.

**int value**

The value to set.

**Prototype:**

**int GetAux3(int knot);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the third integer auxiliary data associated with the specified knot.

This field is available for any use.

**Parameters:**

**int knot**

Specifies the knot to whose auxiliary data is returned.

**Prototype:**

**void SetAux3(int knot, int value);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the third integer auxiliary data associated with the specified knot. This field is available for any use.

**Parameters:**

**int knot**

Specifies the knot whose auxiliary data is set.

**int value**

The value to set.

**Prototype:**

**int GetKnotAux(int knot, int which);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the specified integer auxiliary data associated with the specified knot.

**Parameters:****int knot**

Specifies the knot whose auxiliary data is returned.

**int which**

Specifies which auxiliary field. One of the following values:

**0**=aux1

**1**=aux2

**2**=aux3

**Prototype:**

```
void SetKnotAux(int knot, int which, int value);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified integer auxiliary data associated with the specified knot.

**Parameters:****int knot**

Specifies the knot whose auxiliary data is set.

**int which**

Specifies which auxiliary field. One of the following values:

**0**=aux1

**1**=aux2

**2**=aux3

**int value**

The value to set.

**Prototype:**

```
int GetInAux(int knot, int which);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the specified integer auxiliary data associated with the specified in vector.

**Parameters:**

**int knot**

Specifies the knot whose auxiliary data is returned.

**int which**

Specifies which auxiliary field. One of the following values:

**0**=aux1

**1**=aux2

**2**=aux3

**Prototype:**

**void SetInAux(int knot, int which, int value);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified integer auxiliary data associated with the specified in vector.

**Parameters:**

**int knot**

Specifies the knot whose auxiliary data is set.

**int which**

Specifies which auxiliary field. One of the following values:

**0**=aux1

**1**=aux2

**2**=aux3

**int value**

The value to set.

**Prototype:**

**int GetOutAux(int knot, int which);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the specified integer auxiliary data associated with the specified out vector.

**Parameters:****int knot**

Specifies the knot whose auxiliary data is returned.

**int which**

Specifies which auxiliary field. One of the following values:

**0**=aux1

**1**=aux2

**2**=aux3

**Prototype:**

**void SetOutAux(int knot, int which, int value);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified integer auxiliary data associated with the specified out vector.

**Parameters:****int knot**

Specifies the knot whose auxiliary data is set.

**int which**

Specifies which auxiliary field. One of the following values:

**0**=aux1

**1**=aux2

**2**=aux3

**int value**

The value to set.

**Prototype:**

**int GetVertAux(int i, int which);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the specified integer auxiliary data associated with the specified bezier vertex.

**Parameters:****int i**

The zero based bezier vertex index.

**int which**

Specifies which auxiliary field. One of the following values:

**0**=aux1

**1**=aux2

**2**=aux3

**Prototype:**

**void SetVertAux(int i, int which, int value);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified integer auxiliary data associated with the specified bezier vertex.

**Parameters:****int i**

The zero based bezier vertex index.

**int which**

Specifies which auxiliary field. One of the following values:

**0**=aux1

**1**=aux2

**2**=aux3

**int value**

The value to set.

**Prototype:**

**MtlID GetMatID(int seg);**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the material ID for the specified spline segment.

**Parameters:****int seg**

The zero based index of the segment.

**Prototype:****void SetMatID(int seg, MtlID id);****Remarks:**

This method is available in release 3.0 and later only.  
Sets the material ID for the specified spline segment.

**Parameters:****int seg**

The zero based index of the segment.

**MtlID id**

The material ID to set.

**Prototype:****void GetSmoothingMap(IntTab &map);****Remarks:**

This method is available in release 3.0 and later only.

This method allows this Spline3D to create a map of smoothing groups that eases the creation of meshes.

It fills in a developer supplied IntTab with smoothing groups for each segment of the spline. A spline with 4 segments will cause the IntTab to be set to 4 entries, for example.

Five smoothing groups are used for this operation, 1<<0 through 1<<4. Once you have them, you can shift them as needed for your application. The smoothing groups are set up so that segments connected by knots with **KTYPE\_SMOOTH** or **KTYPE\_BEZIER** types are smoothed together.

**Parameters:**

## IntTab &map

A table of integers. See [Template Class Tab](#).

## Reverse() / Append() / Prepend()

### Prototype:

```
void Reverse(BOOL keepZero = FALSE, BOOL  
weldCoincidentFirstVertex=TRUE);
```

### Remarks:

This method reverses all the points of the spline.

### Parameters:

**BOOL keepZero = FALSE**

This optional parameter is available in release 2.0 and later only.

This parameter defaults to FALSE in order to retain backwards compatibility. Setting it to TRUE insures that a closed spline will have the same vertex as its first point when it is reversed. The parameter is ignored on open splines.

**BOOL weldCoincidentFirstVertex=TRUE**

Set this to TRUE to weld coincident first vertices. FALSE to disable welding.

### Prototype:

```
void Append(Spline3D *spline, BOOL  
weldCoincidentFirstVertex=TRUE);
```

### Remarks:

This method appends the specified spline onto the end of this one. The splines should both be opened.

### Parameters:

**Spline3D \*spline**

The spline to append.

**BOOL weldCoincidentFirstVertex=TRUE**

Set this to TRUE to weld coincident first vertices. FALSE to disable welding.

### Prototype:

```
void Prepend(Spline3D *spline);
```

**Remarks:**

This method takes the specified spline and puts it on the front of this spline.

**Parameters:**

**Spline3D \*spline**

The spline to prepend.

**IsClockWise() / SelfIntersects() / IntersectsSpline() / SurroundsPoint() / SplineLength()****Prototype:**

**BOOL IsClockWise();**

**Remarks:**

Returns TRUE if the spline is clockwise in the XY plane (it ignores Z); otherwise FALSE. This call is meaningless if the shape self intersects.

**Prototype:**

**BOOL SelfIntersects();**

**Remarks:**

Returns TRUE if the spline intersects itself in the XY plane (it ignores Z); otherwise FALSE.

**Prototype:**

**BOOL IntersectsSpline(Spline3D \*spline);**

**Remarks:**

Returns TRUE if this spline intersects the specified spline in the XY plane (it ignores Z); otherwise FALSE.

**Parameters:**

**Spline3D \*spline**

The spline to check.

**Prototype:**

**BOOL SurroundsPoint(Point2 p);**

**Remarks:**

Returns TRUE if the specified point is surrounded (contained within) this spline.

**Parameters:****Point2 p**

The point to check.

**Prototype:**

**float SplineLength();**

**Remarks:**

Returns the length of the spline.

**Prototype:**

**float SegmentLength(int seg);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the length of the specified segment of this spline.

**Parameters:****int seg**

The zero based index of the segment to check.

**MakePolyLine()****Prototype:**

**void MakePolyLine(PolyLine &line, int steps = -1, BOOL optimize = FALSE);**

**Remarks:**

This creates a PolyLine from this spline given a steps setting and an optimize parameter.

Note the following constraints on this method. When a ShapeObject is asked to output a PolyShape with a given number of steps and FALSE is specified for optimization, it must output a PolyLine with  $[\text{steps} * \text{pieces} + \text{pieces} + 1]$  vertices if it's an open shape and  $[\text{steps} * \text{pieces} + \text{pieces}]$  vertices if it's closed.

**Parameters:****PolyLine &line**

The result is stored here.

**int steps = -1**

The number of steps between knots in the spline.

**BOOL optimize = FALSE**

If TRUE, linear segments between control points in the spline will not generate steps in between. It will just be one line segment.

**Dump()****Prototype:****void Dump(int where);****Remarks:**

This displays data about the specified knot using **DebugPrints()**. See [Debugging](#).

**Parameters:****int where**

The index of the knot.

**Internal Methods / Methods that should not be used by plug-in developers****Prototype:****int ShiftKnot(int where, int direction);****Remarks:**

This method is used internally.

**Prototype:****Point3 AverageTangent(int i);****Remarks:**

This method should not be used.

**Prototype:**

**void MakeBezCont(int i);**

**Remarks:**

This method should not be used.

**Prototype:**

**void RedistTangents(int i, Point3 d);**

**Remarks:**

This method should not be used.

**Prototype:**

**void FixAdjBezTangents(int i);**

**Remarks:**

This method should not be used.

**Prototype:**

**void DrawCurve(GraphicsWindow \*gw, Material \*mtl);**

**Remarks:**

This method should not be used.

**Prototype:**

**inline void SetEditMode(int mode);**

**Remarks:**

This method should not be used.

**Prototype:**

**IPoint2 ProjectPoint(ViewExp \*vpt, Point3 fp, Matrix3 \*mat);**

**Remarks:**

This method is used internally.

**Prototype:**

**Point3 UnProjectPoint(ViewExp \*vpt, IPoint2 p, Matrix3 \*mat);**

**Remarks:**

This method is used internally.

**Prototype:**

**void Snap(GraphicsWindow \*gw, SnapInfo \*snap, IPoint2 \*p, Matrix3 &tm);**

**Remarks:**

This method is used internally.

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

This method is used internally to save the class data from disk.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

This method is used internally to load the class data from disk.

**Prototype:**

**Point3 GetDragVector(ViewExp \*vpt, IPoint2 p, int i, Matrix3\* mat);**

**Remarks:**

This method is used internally.

**Prototype:**

**int AppendPoint(ViewExp \*vpt, const Point3& p, int where = -1);**

**Remarks:**

This method is used internally.

**Prototype:**

**int DrawPhase();**

**Remarks:**

This method is used internally.

**Prototype:**

**int GetiCur();**

**Remarks:**

This method is used internally.

**Prototype:**

**inline int ParmType()**

**Remarks:**

This method is used internally.

**Prototype:**

**int SetParam(int index,float param);**

**Remarks:**

This method is used internally.

**Prototype:**

**float GetParam(int index);**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual void CustomParams()**

**Remarks:**

This method is used internally.

**Prototype:**

**void CompParams();**

**Remarks:**

This method is used internally.

**Operators:****Prototype:**

**Spline3D& operator=(Spline3D& fromSpline);**

**Remarks:**

Assignment operator.

**Prototype:**

**Spline3D& operator=(PolyLine& fromLine);**

**Remarks:**

Assignment operator. This generates a PolyLine from the spline, where points are added in between the knots on the spline. For example if the steps value was 5, it will interpolate 5 points in between each knot on the spline.

## Class ShapeObject

See Also: [Class GeomObject](#), [Class PolyShape](#).

```
class ShapeObject : public GeomObject
```

### Description:

ShapeObjects are open or closed hierarchical shapes made up of one or more pieces. This base class defines a set of methods that plug-in shapes must implement.

Note: Many plug-in shapes may be derived from [Class SimpleSpline](#) rather than this class and have fewer methods to implement. See that class for more details.

In release 2.0 and later of 3ds max, ShapeObjects are now renderable. This has introduced a couple of important ramifications. First, any classes subclassing off of ShapeObject should be sure to call the ShapeObject constructor in their constructor, in order to properly initialize the fields contained in the ShapeObject. At present, this is the thickness field, which specifies the thickness of the mesh generated from the shape at rendering time. For example:

```
LinearShape::LinearShape() : ShapeObject() {  
    ...  
}
```

Second, the ShapeObject now contains Load and Save methods, which handle the storage of the data contained within the ShapeObject. In order to properly store this information, classes which subclass off of ShapeObject need to call the ShapeObject Load and Save methods before storing their information. For example:

```
IOResult LinearShape::Save(ISave *isave) {  
    IOResult res = ShapeObject::Save(isave);  
    if(res != IO_OK)  
        return res;  
    ...  
}
```

```
IOResult LinearShape::Load(ILoad *iload) {  
    IOResult res = ShapeObject::Load(iload);
```

```
if(res != IO_OK)
    return res;
    ...
}
```

The number of ShapeObject references/subanimns are defined as **SHAPE\_OBJ\_NUM\_REFS** and **SHAPE\_OBJ\_NUM\_SUBS** in `\include\object.h` and are set to the number of references and subanimns in the ShapeObject class, you can use them to make your code more bullet-proof should the number of references change in the future. See `maxsdk\include\splshape.h` for an example of how they can be used.

**Methods:**

**Prototype:**

```
void CopyBaseData(ShapeObject &from);
```

**Remarks:**

This method is available in release 2.0 and later only.

In order to simplify things for subclasses of ShapeObject, this method is now available. It should be called whenever the ShapeObject-based object is copied. It takes care of copying all the data to the ShapeObject from another ShapeObject-based object

Implemented by the System.

**Parameters:**

**ShapeObject &from**

The ShapeObject to copy from.

**Prototype:**

```
virtual int IntersectRay(TimeValue t, Ray& ray, float& at, Point3& norm)
```

**Remarks:**

Implemented by the Plug-In.

Computes the intersection point of the ray passed and the shape.

Note: In release 3 and later this method has a default implementation and it is no longer necessary to define this method in classes derived from

## **ShapeObject.**

### **Parameters:**

#### **TimeValue t**

The time to compute the intersection.

#### **Ray& ray**

Ray to intersect.

#### **float& at**

The point of intersection.

#### **Point3& norm**

This parameter is available in release 3.0 and later only.

The surface normal at the point of intersection.

### **Return Value:**

Nonzero if a point of intersection was found; otherwise 0.

### **Default Implementation:**

```
{return FALSE;}
```

## **Access methods**

### **Prototype:**

```
virtual BOOL GetRenderable();
```

### **Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

The ShapeObject class now has a "renderable" flag contained within it. Access to this is via this method and SetRenderable(). If this is set to TRUE and the node is set to renderable, the spline will be rendered. This defaults to FALSE.

### **Prototype:**

```
virtual void SetRenderable(BOOL sw);
```

### **Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Sets the rendering flag to the specified value.

**Parameters:**

**BOOL sw**

TRUE for on; FALSE for off.

**Prototype:**

**virtual float GetThickness(TimeValue t, Interval &ivalid);**

**Remarks:**

Implemented by the System.

Returns the shape's thickness setting.

**Parameters:**

**TimeValue t**

This parameter is available in release 4.0 and later only.

The time to obtain the thickness.

**Interval &ivalid**

This parameter is available in release 4.0 and later only.

The validity interval.

**Prototype:**

**virtual void SetThickness(TimeValue t, float thick);**

**Remarks:**

Implemented by the System.

Sets the thickness setting of the shape to the specified value.

**Parameters:**

**TimeValue t**

This parameter is available in release 4.0 and later only.

The time at which to set the thickness.

**float thick**

The new thickness setting for the shape.

**Prototype:**

**int GetSides(TimeValue t, Interval &ivalid);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the number of sides for the cross-section of the rendering mesh version of the shape for the specified time.

**Parameters:**

**TimeValue t**

The time to obtain the thickness.

**Interval &ivalid**

The validity interval.

**Prototype:**

**void SetSides(TimeValue t, int s);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the number of sides for the rendering mesh version of the shape for the specified time. The allowable ranges for this parameter are 3-100.

**Parameters:**

**TimeValue t**

The time at which to set the number of sides.

**int s**

The number of sides you wish to set.

**Prototype:**

**float GetAngle(TimeValue t, Interval &ivalid);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the angle that the cross-section of the rendering mesh will be rotated to, for the specified time.

**Parameters:**

**TimeValue t**

The time to obtain the thickness.

### **Interval &ivalid**

The validity interval.

### **Prototype:**

**void SetAngle(TimeValue t, float a);**

### **Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the cross-section rotation angle for the rendering mesh version of the shape, in degrees, for the specified time.

### **Parameters:**

#### **TimeValue t**

The time at which to set the angle.

#### **float a**

The angle you wish to set, in degrees.

### **Prototype:**

**float GetViewportThickness();**

### **Remarks:**

This method is available in release 4.0 and later only.

This method returns the thickness of the viewport version of the rendering mesh. This is not an animated parameter.

### **Prototype:**

**int GetViewportSides();**

### **Remarks:**

This method is available in release 4.0 and later only.

This method returns the number of sides for the cross-section for the viewport version of the rendering mesh. This is not an animated parameter.

### **Prototype:**

**void SetViewportSides(int s);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the number of sides for the viewport version of the rendering mesh. This is not an animated parameter.

**Parameters:**

**int s**

The number of viewport sides you wish to set.

**Prototype:**

**float GetViewportAngle();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the angle that the cross-section of the viewport version of the rendering mesh will be rotated to. This is not an animated parameter.

**Prototype:**

**void SetViewportAngle(float a);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the angle that the cross-section of the viewport version of the rendering mesh will be rotated to, in degrees. This is not an animated parameter.

**Parameters:**

**float a**

The viewport angle you wish to set, in degrees.

**Prototype:**

**BOOL GetDispRenderMesh();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns TRUE if the "Display Render Mesh" switch is on. FALSE when the switch is off.

**Prototype:**

**void SetDispRenderMesh(BOOL sw);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to turn the "Display Render Mesh" switch on or off.

**Parameters:**

**BOOL sw**

TRUE or FALSE to set or unset the "Display Render Mesh" switch.

**Prototype:**

**BOOL GetUseViewport();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns TRUE if the "Use Viewport Settings" switch is on.

FALSE when the switch is off.

**Prototype:**

**void SetUseViewport(BOOL sw);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to turn the "Use Viewport Settings" switch on or off.

**Parameters:**

**BOOL sw**

TRUE or FALSE to set or unset the "Use Viewport Settings" switch.

**Prototype:**

**BOOL GetViewportOrRenderer();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the value of the Viewport/Render switch and either returns **GENMESH\_VIEWPORT** or **GENMESH\_RENDER**.

**Prototype:**

**void SetViewportOrRenderer(BOOL sw);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the value of the Viewport/Render switch.

**Parameters:**

**BOOL sw**

Set this parameter to **GENMESH\_VIEWPORT** or **GENMESH\_RENDER**.

**Prototype:**

**virtual BOOL GetGenUVs();**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Returns TRUE if the generate UVs switch is on; FALSE if off.

**Prototype:**

**virtual void SetGenUVs(BOOL sw);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Pass TRUE to set the generate UVs switch to on; FALSE to set it off.

**Parameters:**

**BOOL sw**

TRUE for on; FALSE for off.

**Prototype:**

**void GetRenderMeshInfo(TimeValue t, INode \*inode, View& view, int &nverts, int &nfaces);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns information on the rendering mesh.

Implemented by the System.

**Parameters:**

**TimeValue t**

The time to get the information.

**INode \*inode**

The node associated with the mesh.

**View& view**

Describes properties of the view associated with the render. See [Class View](#).

**int &nverts**

The number of vertices in the render mesh.

**int &nfaces**

The number of faces in the render mesh.

**Prototype:**

**virtual int NumberOfVertices(TimeValue t, int curve = -1);**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method is used by the Summary Info and Object Properties dialogs to inform the user how many vertices or CVs are in the object. The method is passed a TimeValue and a curve index; if the curve index is <0, the function should return the number of vertices/CVs in the entire shape. Otherwise, it should return the number of vertices/CVs in the specified curve.

**Parameters:**

**TimeValue t**

The time at which the number of vertices is to be computed.

**int curve = -1**

The curve index. See note above.

**Default Implementation:**

**{ return 0; }**

**Prototype:**

**virtual int NumberOfCurves()=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the number of polygons in the shape.

**Prototype:**

**virtual BOOL CurveClosed(TimeValue t, int curve)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to determine if the specified curve of the shape is closed at the time passed.

**Parameters:**

**TimeValue t**

The time to check.

**int curve**

The index of the curve to check.

**Return Value:**

TRUE if the curve is closed; otherwise FALSE.

**Prototype:**

**virtual Point3 InterpCurve3D(TimeValue t, int curve, float param, int ptype=PARAM\_SIMPLE)=0;**

**Remarks:**

Implemented by the Plug-In.

This method returns a point interpolated on the entire curve. This method returns the point but you don't know which segment the point falls on. See method **InterpPiece3D()**.

**Parameters:**

**TimeValue t**

The time to evaluate.

**int curve**

The index of the curve to evaluate.

**float param**

The 'distance' along the curve where 0 is the start and 1 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The interpolated point on the curve.

**Prototype:**

```
virtual Point3 TangentCurve3D(TimeValue t, int curve, float param, int ptye=PARAM_SIMPLE)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method returns a tangent vector interpolated on the entire curve. Also see method **TangentPiece3D()**.

**Parameters:**

**TimeValue t**

The time at which to evaluate the curve.

**int curve**

The index of the curve to evaluate.

**float param**

The 'distance' along the curve where 0.0 is the start and 1.0 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The tangent vector

**Prototype:**

```
virtual float LengthOfCurve(TimeValue t, int curve)=0;
```

**Remarks:**

Implemented by the Plug-In.

Returns the length of the specified curve.

Note: This method makes no allowance for non-uniform scaling in the object transform. To do that, see the following code fragment (**os** is the **ObjectState** with the shape object and **xfm** is the NodeTM of the shape object node).

```
if (os.obj->SuperClassID() == SHAPE_CLASS_ID) {  
    ShapeObject *sobj;  
    sobj = (ShapeObject *) os.obj;  
    int cct = sobj->NumberOfCurves();  
    PolyShape workShape;  
    sobj->MakePolyShape(ip->GetTime(), workShape);  
    workShape.Transform(xfm);  
    float len = 0.0f;  
    for (int i=0; i<cct; i++)  
        len += workShape.lines[i].CurveLength();  
    }
```

**Parameters:**

**TimeValue t**

The time at which to compute the length.

**int curve**

The index of the curve.

**Prototype:**

```
virtual int NumberOfPieces(TimeValue t, int curve)=0;
```

**Remarks:**

Implemented by the Plug-In.

Returns the number of sub-curves in a curve.

**Parameters:**

**TimeValue t**

The time at which to check.

**int curve**

The index of the curve.

**Prototype:**

**virtual Point3 InterpPiece3D(TimeValue t, int curve,  
int piece, float param, int ptye=PARAM\_SIMPLE)=0;**

**Remarks:**

Implemented by the Plug-In.

This method returns the interpolated point along the specified sub-curve (segment). For example consider a shape that is a single circle with four knots. If you called this method with curve=0 and piece=0 and param=0.0 you'd get back the point at knot 0. If you passed the same parameters except param=1.0 you'd get back the point at knot 1.

**Parameters:**

**TimeValue t**

The time to evaluate the sub-curve.

**int curve**

The curve to evaluate.

**int piece**

The segment to evaluate.

**float param**

The position along the curve to return where 0.0 is the start and 1.0 is the end.

**int ptye=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The point in world space.

**Prototype:**

**virtual Point3 TangentPiece3D(TimeValue t, int curve,  
int piece, float param, int ptye=PARAM\_SIMPLE)=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the tangent vector on a sub-curve at the specified 'distance' along the

curve.

**Parameters:**

**TimeValue t**

The time to evaluate the sub-curve.

**int curve**

The curve to evaluate.

**int piece**

The sub-curve (segment) to evaluate.

**float param**

The position along the curve to return where 0 is the start and 1 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The tangent vector.

**Prototype:**

**virtual MtlID GetMatID(TimeValue t, int curve, int piece);**

**Remarks:**

Implemented by the Plug-In.

This method is available in release 3.0 and later only.

This method provides access to the material IDs of the shape. It returns the material ID of the specified segment of the specified curve of this shape at the time passed. There is a default implementation so there is no need to implement this method if the shape does not support material IDs.

Note: **typedef unsigned short MtlID;**

**Parameters:**

**TimeValue t**

The time to evaluate the sub-curve.

**int curve**

The zero based index of the curve to evaluate.

**int piece**

The sub-curve (segment) to evaluate.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual BOOL CanMakeBezier()
```

**Remarks:**

Implemented by the Plug-In.

This method is called to determine if the shape can be converted to a bezier representation.

**Return Value:**

TRUE if the shape can turn into a bezier representation; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void MakeBezier(TimeValue t, BezierShape &shape);
```

**Remarks:**

Implemented by the Plug-In.

Creates the bezier representation of the shape.

**Parameters:**

**TimeValue t**

The time to convert.

**BezierShape &shape**

The bezier representation is stored here.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual ShapeHierarchy &OrganizeCurves(TimeValue t,  
ShapeHierarchy *hier=NULL)=0
```

**Remarks:**

Implemented by the Plug-In.

This method is called to prepare the shape for lofting, extrusion, etc. This methods looks at the shape organization, and puts together a shape hierarchy. This provides information on how the shapes are nested.

**Parameters:**

**TimeValue t**

The time to organize the curves.

**ShapeHierarchy \*hier=NULL**

This class provides information about the hierarchy. See [Class ShapeHierarchy](#).

**Prototype:**

```
virtual void MakePolyShape(TimeValue t, PolyShape &shape,  
    int steps = PSHAPE_BUILTIN_STEPS, BOOL optimize =  
    FALSE)=0;
```

**Remarks:**

Implemented by the Plug-In.

Create a PolyShape representation with optional fixed steps.

**Parameters:**

**TimeValue t**

The time to make the **PolyShape**.

**PolyShape &shape**

The **PolyShape** representation is stored here.

**int steps = PSHAPE\_BUILTIN\_STEPS**

The number of steps between knots. Values  $\geq 0$  indicates the use of fixed steps:

**PSHAPE\_BUILTIN\_STEPS**

Use the shape's built-in steps/adaptive settings (default).

**PSHAPE\_ADAPTIVE\_STEPS**

Force adaptive steps.

**BOOL optimize = FALSE**

If TRUE intermediate steps are removed from linear segments.

**Prototype:**

```
virtual int MakeCap(TimeValue t, MeshCapInfo &capInfo, int capType)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method generates a mesh capping info for the shape.

**Parameters:****TimeValue t**

The time to create the cap info.

**MeshCapInfo &capInfo**

The cap info to update.

**int capType**

See [List of Cap Types](#).

**Return Value:**

Nonzero if the cap info was generated; otherwise zero.

**Prototype:**

```
virtual int MakeCap(TimeValue t, PatchCapInfo &capInfo)
```

**Remarks:**

Implemented by the Plug-In.

This method creates a patch cap info out of the shape. Only implement this method if **CanMakeBezier()** returns TRUE.

**Parameters:****TimeValue t**

The time to create the cap info.

**PatchCapInfo &capInfo**

The cap info to update.

**Return Value:**

Nonzero if the cap info was generated; otherwise zero.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual BOOL AttachShape(TimeValue t, INode *thisNode, INode
*attachNode, BOOL weldEnds=FALSE, float
weldThreshold=0.0f);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to attach the shape of **attachNode** to **thisNode** at the specified time. If any endpoints of the curves in the shape being attached are within the threshold distance to endpoints of an existing curve, and the weld flag is TRUE, they should be welded.

**Parameters:****TimeValue t**

The time to attach.

**INode \*thisNode**

This is the node associated with this shape object.

**INode \*attachNode**

The node of the shape to attach.

**BOOL weldEnds=FALSE**

This parameter is available in release 3.0 and later only.

If TRUE the endpoints of the shape should be welded together (based on the threshold below). If FALSE no welding is necessary.

**float weldThreshold=0.0f**

This parameter is available in release 3.0 and later only.

If any endpoints of the curves in the shape being attached are within this threshold distance to endpoints of an existing curve, and the weld flag is TRUE, they should be welded

**Return Value:**

Return TRUE if attached; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual IOResult Save(ISave *isave);
```

**Remarks:**

Implemented by the System.

This method handles the storage of the data contained within the ShapeObject. In order to properly store this information, classes which subclass off of ShapeObject need to call this methods before storing their information.

**Parameters:****ISave \*isave**

An interface for saving data. See [Class ISave](#).

**Prototype:**

**virtual IOResult Load(ILoad \*iload);**

**Remarks:**

Implemented by the System.

This method handles the loading of the data contained within the ShapeObject. In order to properly load this information, classes which subclass off of ShapeObject need to call this methods before loading their information.

**Parameters:****ILoad \*iload**

An interface for loading data. See [Class ILoad](#).

**Prototype:**

**virtual Class\_ID PreferredCollapseType();**

**Remarks:**

Implemented by the System.

This is an implementation of the **Object** method. It simply returns **splineShapeClassID**.

**Prototype:**

**virtual BOOL GetExtendedProperties(TimeValue t, TSTR &prop1Label, TSTR &prop1Data, TSTR &prop2Label, TSTR &prop2Data);**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

This is an implementation of the **Object** method. It fills in the property fields with the number of vertices and curves in the shape.

**Prototype:**

```
virtual void RescaleWorldUnits(float f);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Objects derived from this class which have RescaleWorldUnits methods implemented need to call this method. The following example is the

**SplineShape** implementation of this method from core.

```
void SplineShape::RescaleWorldUnits(float f) {  
    if (TestAFlag(A_WORK1))  
        return;  
    // Call the base class's rescale (this sets the A_WORK1 flag)  
    ShapeObject::RescaleWorldUnits(f);  
    // Now rescale stuff inside our data structures  
    Matrix3 stm = ScaleMatrix(Point3(f, f, f));  
    shape.Transform(stm);  
}
```

Note that the A\_WORK1 flags is tested first to be sure it isn't processing the rescale twice. The code then calls ShapeObject::RescaleWorldUnits, which sets the A\_WORK1 flag and performs the necessary rescale methods for all references for the object, and scales the renderable thickness value.

**Parameters:**

**float f**

The parameter to scale.

**Prototype:**

```
virtual void GenerateMesh(TimeValue t, int option, Mesh *mesh);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method will generate a mesh based on either the viewport or rendering parameters for the specified time.

**Parameters:****TimeValue t**

The time at which to generate the mesh.

**int option**

The option can be either GENMESH\_VIEWPORT, GENMESH\_RENDER, or GENMESH\_DEFAULT. When using the default definition the mesh generator will use whatever is in the Viewport/Render switch in the parameter block.

**Mesh \*mesh**

A pointer to a Mesh object. If this is set to NULL, the mesh will be generated and cached, but not returned.

**Prototype:**

```
virtual RefResult NotifyRefChanged(Interval changeInt,  
RefTargetHandle hTarget, PartID& partID, RefMessage  
message);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

This method will notify the Shape Object of changes in values in its parameter block. The ShapeObject's parameter block is reference number zero. If subclasses implement this method, they should pass any messages referring to the ShapeObject's parameter block to it. For example:

```
// If this isn't one of our references, pass it on to the ShapeObject...
```

```
if(hTarget == GetReference(0))  
    return ShapeObject::NotifyRefChanged(  
        changeInt, hTarget, partID, message);
```

This is a vital part of the mechanism; When a parameter in the parameter block changes, the ShapeObject must be able to flush its cached mesh which will no longer be valid.

**Parameters:****Interval changeInt**

This is the interval of time over which the message is active.

**RefTargetHandle hTarget**

This is the handle of the reference target the message was sent by. The reference maker uses this handle to know specifically which reference target sent the message.

**PartID& partID**

This contains information specific to the message passed in. Some messages don't use the partID at all. See the section [List of Reference Messages](#) for more information about the meaning of the partID for some common messages.

**RefMessage message**

The msg parameter passed into this method is the specific message which needs to be handled. See [List of Reference Messages](#).

**Return Value:**

The return value from this method is of type **RefResult**. This is usually **REF\_SUCCEED** indicating the message was processed. Sometimes, the return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

**Prototype:**

```
virtual RefTargetHandle GetReference(int i);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows the ShapeObject to return a pointer to its parameter block. Any subclasses implementing this method must pass on the call if it indicates the ShapeObject's reference. For example:

```
RefTargetHandle SomeShape::GetReference(int i) {  
    If(i == 0) return ShapeObject::GetReference(i);  
}
```

**Parameters:**

```
int i
```

The reference handle to retrieve.

**Return Value:**

The handle to the Reference Target.

**Prototype:**

```
virtual void SetReference(int i, RefTargetHandle rtarg);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method sets the ShapeObject's parameter block pointer. Any subclasses implementing this method must pass on the call to the ShapeObject if it refers to index 0. For example:

```
void SomeShape::SetReference(int i, RefTargetHandle rtarg) {  
    if(i == 0) ShapeObject::SetReference(i, rtarg);  
}
```

**Parameters:**

**int i**

The virtual array index of the reference to store.

**RefTargetHandle rtarg**

The reference handle to store.

**Prototype:**

```
virtual Animatable* SubAnim(int i);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the ShapeObject's animatable pointer. Derived classes implementing this method must pass on references to index 0 to the ShapeObject. For example::

```
Animatable* SomeShape::SubAnim(int i) {  
    if(i == 0) return ShapeObject::SubAnim(i);  
}
```

**Parameters:**

**int i**

This is the index of the sub-anim to return.

**Prototype:**

**virtual TSTR SubAnimName(int i);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the name of the animatable's name. Derived classes implementing this method must pass on references to index 0 to the ShapeObject. For example:

```
TSTR SomeShape::SubAnimName(int i) {  
    if(i == 0) return ShapeObject::SubAnimName(i);  
}
```

**Parameters:**

**int i**

This is the index of the sub-anim's name to return.

**Prototype:**

**ParamDimension \*GetParameterDim(int pbIndex);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the parameter dimension of the parameter whose index is passed.

**Parameters:**

**int pbIndex**

The index of the parameter to return the dimension of.

**Return Value:**

Pointer to a ParamDimension.

**Prototype:**

**TSTR GetParameterName(int pbIndex);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the name of the parameter whose index is passed.

**Parameters:**

**int pbIndex**

The index of the parameter to return the dimension of.

**Prototype:**

**virtual int RemapRefOnLoad(int iref);**

**Remarks:**

This method is available in release 4.0 and later only.

This method remaps references at load time so that files saved from previous versions of 3ds max get their references adjusted properly to allow for the new ShapeObject reference. If derived classes implement this method, they must properly pass on the call to the ShapeObject's code. An example from the SplineShape code:

```
int SplineShape::RemapRefOnLoad(int iref) {  
    // Have the ShapeObject do its thing first...  
    iref = ShapeObject::RemapRefOnLoad(iref);  
    if(loadRefVersion == ES_REF_VER_0)  
        return iref+1;  
    return iref;  
}
```

Note that the SplineShape first calls ShapeObject's remapper, then applies its remapping operation to the index returned by the ShapeObject code.

**IMPORTANT NOTE:** For this remapping to operate properly, the derived class **MUST** call ShapeObject::Save as the first thing in its ::Save method, and must call ShapeObject::Load as the first thing in its ::Load method. This allows the ShapeObject to determine file versions and the need for remapping references.

**Parameters:**

**int iref**

The input index of the reference.

**Return Value:**

The output index of the reference.

**Prototype:**

```
virtual int NumRefs();
```

**Remarks:**

This method is available in release 4.0 and later only.

The ShapeObject makes 1 reference; this is where it tells the system. Any derived classes implementing this method must take this into account when returning the number of references they make. A good idea is to implement NumRefs in derived classes as:

```
Int SomeShape::NumRefs() {  
    return myNumRefs + ShapeObject::NumRefs();  
}
```

**Default Implementation:**

```
{return 1;}
```

**Prototype:**

```
void BeginEditParams(IObjParam *ip, ULONG flags,Animatable  
*prev);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows the ShapeObject to create its new "Rendering" rollup. To use it, the derived class simply calls it first thing in its own BeginEditParams method. An example from the SplineShape code:

```
void SplineShape::BeginEditParams(IObjParam *ip, ULONG  
flags,Animatable *prev )  
{  
    ShapeObject::BeginEditParams(ip, flags, prev);  
    // ?  
}
```

**Parameters:**

```
IObjParam *ip
```

The interface pointer passed to the plug-in.

## **ULONG flags**

The flags passed along to the plug-in in **Animatable::BeginEditParams()**.

## **Animatable \*prev**

The pointer passed to the plug-in in **Animatable::BeginEditParams()**.

### **Prototype:**

```
void EndEditParams(IObjParam *ip, ULONG flags,Animatable *next);
```

### **Remarks:**

This method is available in release 4.0 and later only.

Similarly to BeginEditParams, this method allows the ShapeObject to remove its "Rendering" rollup. A derived class simply calls this first thing in its own EndEditParams. An example from the SplineShape code:

```
void SplineShape::EndEditParams( IObjParam *ip, ULONG flags,Animatable *next )
```

```
{  
    ShapeObject::EndEditParams(ip, flags, next);  
    // ?  
}
```

### **Parameters:**

#### **IObjParam \*ip**

The interface pointer passed to the plug-in.

#### **ULONG flags**

The flags passed along to the plug-in in **Animatable::BeginEditParams()**.

#### **Animatable \*prev**

The pointer passed to the plug-in in **Animatable::BeginEditParams()**.

### **Prototype:**

```
Interval GetShapeObjValidity(TimeValue t);
```

### **Remarks:**

This method is available in release 4.0 and later only.

This method gets the validity interval for the ShapeObject's internal

parameters only. It DOES NOT include those of the derived classes. So, if you called this method on a ShapeObject that was a circle with an animated radius, you wouldn't see the effect of the animated radius on the interval. All you'd see would be the interval of the ShapeObject's rendering parameters. To get the entire ShapeObject's interval, you would call **ShapeObject::ObjectShapeObjValidity(t).**

**Parameters:**

**TimeValue t**

The time about which the interval is computed.

**Prototype:**

**int Display(TimeValue t, INode \*inode, ViewExp\* vpt, int flags);**

**Remarks:**

This method is available in release 4.0 and later only.

This method displays the shape's generated mesh if necessary. Objects derived from ShapeObject will want to have the ShapeObject code display the rendering mesh in the viewport; this method will do that for them. Simply set the viewport transform and call this method. An example from the SplineShape code:

**int SplineShape::Display(TimeValue t, INode \*inode, ViewExp\* vpt, int flags)**

```
{  
    Eval(t);  
    GraphicsWindow *gw = vpt->getGW();  
    gw->setTransform(inode->GetObjectTM(t));  
    ShapeObject::Display(t, inode, vpt, flags);  
    // ?  
}
```

If the ShapeObject's "Display Render Mesh" switch is off, it will do nothing. Otherwise, it will display the proper mesh as specified by its parameter block.

**Parameters:**

**TimeValue t**

The time to display the object.

**INode\* inode**

The node to display.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**int flags**

See [List of Display Flags](#).

**Return Value:**

The return value is not currently used.

**Prototype:**

**virtual Box3 GetBoundingBox(TimeValue t, Matrix3 \*tm=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns a bounding box for the shape, if it's active, if the "Display Render Mesh" switch is on. It is necessary to include this box when computing the bounding box for a shape, otherwise the viewport display will not work properly.

**Parameters:**

**TimeValue t**

The time to get the bounding box.

**Matrix3 \*tm**

The points of ShapeObject are transformed by this matrix prior to the bounding box computations.

**Prototype:**

**virtual void InvalidateGeomCache();**

**Remarks:**

This method is available in release 4.0 and later only.

This method is very important. It causes the ShapeObject to flush its cached rendering mesh. Most objects have their own "InvalidateGeomCache"

methods; simply call this when a shape derived from ShapeObject changes and it will ensure that the rendering mesh is regenerated the next time it is evaluated. Failure to call this method will result in improper rendering mesh updates.

## Class CollisionObject

See Also: [Class Point3](#).

```
class CollisionObject : public InterfaceServer
```

### Description:

A collision object can be applied to a particle system by a Space Warp. The collision object checks a particle's position and velocity and determines if the particle will collide with it in the next **dt** amount of time. If so, it modifies the position and velocity.

### Methods:

### Prototype:

```
virtual BOOL CheckCollision(TimeValue t,Point3 &pos,  
    Point3 &vel, float dt, int index, float *ct=NULL, BOOL  
    UpdatePastCollide=TRUE)=0;
```

### Remarks:

Implemented by the Plug-In.

This method checks a particles position and velocity to determine if there was a collision between the particle and this collision object. If there was a collision, the particles position and velocity should be modified.

The plug-in may compute a line segment between where the particle is now, and where it will be in **dt** amount of time. The plug-in then checks if the line segment intersects the collision object. If so, it would compute the resulting position, and modify the velocity vector **vel** to point in the new direction (presumably reflected across the surface normal).

3ds max 3.0 introduced interparticle collision (where particles may collide with other particles). In order to implement interparticle collision (IPC) in the presence of collision objects, it became necessary to generalize the operation of the deflectors so that they didn't always work in discrete time intervals.

That is, in the general case of an unidentified number of particles interacting in an unspecified way, it was necessary to allow everything involved in that system to update to specified times without moving **past** that time.

In the absence of IPC enabled, the particle system calls the bound collision objects with the parameter **UpdatePastCollide == TRUE**, and the

deflector checks all collisions, updates particles based on their collisions with deflectors and the ensuing, remaining time intervals in **dt** subsequent to the collisions.

In the presence of IPC that won't work. When IPC is active, all particles need to be updated to the time of the first collision in the system in **dt**, whether that collision be between particles or between particles and deflectors. Thus, in the presence of IPC, all particle updates to bound deflectors are called with **UpdatePastCollide == FALSE**. In that case, the collision objects return both the position and velocity of the updated particles **and** the time at which the collision occurred.

All such times are compared, along with all possible internally calculated IPC event times. If there are any nonnegative times returned greater than or equal to zero, all particle states are reverted to their states at the beginning of the interval and then updated to the precise minimum time returned as the earliest collision. **And then** everything starts up again trying to update itself to the next integral time, when control can pass back to whatever is asking the particles to update themselves. If there are other collisions in that time, it happens again.

This whole set of operations happens after any true returns from the trilinear sort/sweep correlator that looks for the **possibility** of collisions. If there are no possible collisions, everything proceeds through a complete interval normally.

#### **Parameters:**

##### **TimeValue t**

The time to check for collision.

##### **Point3 &pos**

The position of the particle to check and potentially modify.

##### **Point3 &vel**

The velocity vector of the particle to check and potentially modify.

##### **float dt**

This is an increment of time - the step size. The method checks if the particle will collide in this amount of time.

##### **int index**

This parameter is available in release 2.0 and later only.

The index of the particle being collided.

**float \*ct=NULL**

This parameter is available in release 3.0 and later only.

An array of floating point times at which the collisions occurred.

**BOOL UpdatePastCollide=TRUE**

This parameter is available in release 3.0 and later only.

This is a flag to tell the collision object to update the particle past the collision to the remainder of input **dt** or to output the state of the particle at the collision. In the presence of interparticle collision enable, we have to update to the times of collisions and then retest from there. See the Remarks.

**Return Value:**

TRUE if there was a collision and the position and velocity have been modified; otherwise FALSE.

**Prototype:**

**virtual Object \*GetSWObject()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method provides a way of identifying the 'parent' Deflector for a CollisionObject available to a particle system. This must be implemented by all Deflectors. It returns the object pointer to the Deflector from which the Collision object is derived.

# Class ForceField

See Also: [Class Point3](#), [Class WSMObject](#).

class ForceField

## Description:

A Space Warp modifier usually uses an instance of this class and implements the **Force()** method. The force field is then applied to the particle system when the particle system is bound to the Space Warp. This class is similar in some respects to the Deformer class as used by a modifier. The difference is that a deformer modifies the points of the object. A force field is really an acceleration - it modifies velocity rather than position.

The force field provides a function of position in space, velocity and time that gives a force. The force is then used to compute an acceleration on a particle which modifies its velocity. Typically, particles are assumed to have a normalized uniform mass equal to 1 so the acceleration is  $F/M = F$ .

## Methods:

### Prototype:

```
virtual Point3 Force(TimeValue t, const Point3 &pos, const Point3  
&vel, int index)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called to compute a force on the particle based on its position, velocity and the current time.

### Parameters:

#### **TimeValue t**

The time to compute the force.

#### **const Point3 &pos**

The current position of the particle.

#### **const Point3 &vel**

The current velocity of the particle.

#### **int index**

This parameter is available in release 2.0 and later only.

The index of the particle being forced.

**Return Value:**

The force vector as a Point3. This vector is added to the velocity.

**Prototype:**

**virtual void DeleteThis();**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to delete this instance of the ForceField. This should be called, for example, by developers who use the

**WSMObject::GetForceField()** method.

**Default Implementation:**

```
{}
```

# Class Animatable

See Also: [Class Interface](#), [Class INode](#), [List of Animatable Flags](#), [Class IGraphObjectManager](#), [Class IGraphNode](#), [Class Object](#).

```
class Animatable : public InterfaceServer
```

## Description:

This is the base class for almost all classes related to animation. Methods are available for getting the ClassID and SuperClassID of the plug-in, deleting this instance of the plug-in class, and parameter editing in the command panel. There are also methods that deal with the sub-animatables of the plug-in. Most of the track view related methods are here as well.

## Method Groups

These hyperlinks take you to the start of groups of related methods within the class.

[Bitmap Related Methods](#)

[Class Name, ClassID and SuperClassID Methods](#)

[Memory Management](#)

[Parameter Editing Methods](#)

[Sub-Anims Methods](#)

[Controller Assignment](#)

[Animation Properties](#)

[Make Unique Control](#)

[Flag Access Methods](#)

[RenderBegin and RenderEnd](#)

[System Plug-in Related Methods](#)

[Sub-Class Indication](#)

[AppData, Interfaces, and Set/GetProperties Methods](#)

[Operations to Keys](#)

[Track View Methods](#)

[Clipboard Methods](#)

[Interactive Adjustment](#)

[AnimTree and Auxiliary File Enumeration](#)

[NoteTracks](#)

[Parameter Block2 Methods](#)

[Schematic View Methods](#)

## [Custom Attributes](#)

### **Data Members:**

protected:

#### **unsigned long aflag;**

The flags. These may be manipulated using the methods **SetAFlag()**, **ClearAFlag()**, and **TestAFlag()**. See [List of Animatable Flags](#).

#### **AnimPropertyList aprops;**

This is a table of properties that a plug-in may use for its own storage needs. This table is also used by the system (for example Note Tracks and APP\_DATA). A plug-in may use this, for example, when a class has some data that is used while its user interface is up. It can store the UI data on the property list temporarily and not have to carry around the data when it is not needed. See the methods of [Template Class Tab](#) for how to add and delete items from the list. Also see the methods **GetProperty()** and **SetProperty()** and [Class AnimPropertyList](#).

### **Methods:**

## **Class Name, ClassID and SuperClassID Methods**

### **Prototype:**

```
virtual void GetClassName(TSTR& s)
```

### **Remarks:**

Implemented by the Plug-In.

Retrieves the name of the plug-in class. This name is usually used internally for debugging purposes. For Material plug-ins this method is used to put up the material "type" name in the Material Editor.

### **Parameters:**

```
TSTR& s
```

The string to store the name of the class.

### **Prototype:**

```
virtual Class_ID ClassID();
```

### **Remarks:**

Implemented by the Plug-In.

This method must return the **unique** ID for the object. If two ClassIDs conflict, the system will only load the first one it finds. Therefore all ClassIDs must be unique to ensure they are loaded properly. A program is provided to generate these randomly. [Generate a random Class ID](#).

**Return Value:**

The Class\_ID of the plug-in.

See Also: [Class ClassID](#), [List of Class IDs](#).

**Prototype:**

**virtual SClass\_ID SuperClassID();**

**Remarks:**

Implemented by the Plug-In.

This method returns a system defined constant describing the class this plug-in class was derived from. The entire list of available super class IDs is in the include file **PLUGAPI.H**. Note: **typedef ulong SClass\_ID;**

**Return Value:**

The super class ID of the plug-in. See [List of Super Class IDs](#).

## Memory Management

**Prototype:**

**virtual void DeleteThis();**

**Remarks:**

Implemented by the Plug-In.

When the system needs to delete an instance of a plug-in class it calls this method. The developer must use the same memory manager to allocate and deallocate memory. For example, if the developer used the 'new' operator to allocate memory, he or she should use the 'delete' operator to deallocate it. See the method **ClassDesc::Create()** for details on the allocation of memory.

Note: The default implementation of this method contains an **assert(0)** statement. This is because developers **MUST** override this default implementation. If the memory is not to be deleted this method should be

implemented as `{}` so the assertion doesn't happen. See [Memory Allocation](#).  
See Also: [DLL Functions and Class Descriptors](#), [Class ClassDesc](#).

**Prototype:**

**virtual void BeginEditParams(IObjParam \*ip,ULONG flags,Animatable \*prev=NULL)**

**Remarks:**

Implemented by the Plug-In.

This method is called by the system when the user may edit the item's (object, modifier, controller, etc.) parameters.

**Parameters:**

**IObjParam \*ip**

This is an interface pointer passed in. The developer may use the interface pointer to call methods such as **AddRollupPage()**. Note that this pointer is only valid between **BeginEditParams()** and **EndEditParams()**. A developer should not hang onto and use this pointer outside this interval.

**ULONG flags**

These flags describe which branch of the command panel or dialog the item is being editing in.

**BEGIN\_EDIT\_CREATE**

Indicates an item is being edited in the create branch.

**BEGIN\_EDIT\_MOTION**

Indicates a controller is being edited in the motion branch.

**BEGIN\_EDIT\_HIERARCHY**

Indicates a controller is being edited in the Pivot subtask of the hierarchy branch.

**BEGIN\_EDIT\_IK**

Indicates a controller is being edited in the IK subtask of the hierarchy branch.

**BEGIN\_EDIT\_LINKINFO**

Indicates a controller is being edited in the Link Info subtask of the hierarchy branch.

**Animatable \*prev=NULL**

This parameter may be used in the motion and hierarchy branches of the command panel. This pointer allows a plug-in to look at the ClassID of the previous item that was being edited, and if it is the same as this item, to not replace the entire UI in the command panel, but simply update the values displayed in the UI fields. This prevents the UI from 'flickering' when the current item begins its edit. For example, if you are in the motion branch and are looking at an item's PRS controller values, and then select another item that is displayed with a PRS controller, the UI will not change - only the values displayed in the fields will change. If however you selected a target camera that has a lookat controller (not a PRS controller) the UI will change because a different set of parameters need to be displayed. Note that for items that are edited in the modifier branch this field can be ignored.

**Prototype:**

**virtual void EndEditParams(IObjParam \*ip, ULONG flags,Animatable \*next=NULL)**

**Remarks:**

Implemented by the Plug-In.

This method is called when the user is finished editing an objects parameters. The system passes a flag into the **EndEditParams()** method to indicate if the rollup page should be removed. If this flag is TRUE, the plug-in must unregister the rollup page, and delete it from the panel.

**Parameters:**

**IObjParam \*ip**

This is an interface pointer passed in. The developer may use the interface pointer to call methods such as **DeleteRollupPage()**.

**ULONG flags**

The following flag may be set:

**END\_EDIT\_REMOVEUI**

If TRUE, the item's user interface should be removed.

**Animatable \*next=NULL**

This parameter may be used in the motion and hierarchy branches of the command panel. This pointer allows a plug-in to look at the ClassID of the next item that was being edited, and if it is the same as this item, to not replace

the entire UI in the command panel. Note that for items that are edited in the modifier branch this field can be ignored.

## Bitmap Related Methods

### Prototype:

```
virtual void FreeAllBitmaps();
```

### Remarks:

This method is available in release 2.0 and later only.

This method frees all bitmaps in this **Animatable** but doesn't recurse. This is used for freeing all the scene bitmaps after a render.

### Default Implementation:

```
{ }
```

## Methods for Sub-Anims.

See Also: Advanced Topics [Sub-Anims](#).

### Prototype:

```
virtual int NumSubs()
```

### Remarks:

Implemented by the Plug-In.

The system uses a [= 4\) BSPSPopupOnMouseOver\(event\);">virtual array](#) mechanism to access the sub-anims of a plug-in. This method returns the total number of sub-anims maintained by the plug-in. If a plug-in is using a parameter block to manage its parameters it should just return 1 for all the parameters directed by the parameter block.

### Return Value:

The number of sub-anims used by the plug-in.

### Default Implementation:

```
{ return 0; }
```

### Prototype:

```
virtual Animatable* SubAnim(int i)
```

**Remarks:**

Implemented by the Plug-In.

This method returns a pointer to the 'i-th' sub-anim. If a plug-in is using a parameter block to manage all its parameters it should just return a pointer to the parameter block itself from this method. This method may return NULL so developers need to check the return value before calling other sub anim methods (such as **SubAnimName()**).

**Parameters:**

**int i**

This is the index of the sub-anim to return.

**Default Implementation:**

```
{ return NULL };
```

**Prototype:**

```
virtual TSTR SubAnimName(int i);
```

**Remarks:**

Implemented by the Plug-In.

This method returns the name of the 'i-th' sub-anim to appear in track view. The system has no idea what name to assign to the sub-anim (it only knows it by the virtual array index), so this method is called to retrieve the name to display. Developer need to make sure the 'i-th' **SubAnim()** is non-NULL or this method will fail.

**Parameters:**

**int i**

The index of the parameter name to return

**Return Value:**

The name of the 'i-th' parameter.

**Prototype:**

```
virtual BOOL SelectSubAnim(int subNum);
```

**Remarks:**

This method is available in release 2.0 and later only.

When a user is in Track View in Edit Keys mode and clicks on the green triangle of a controller then this method will be called on the client with the appropriate sub number that corresponds to it. For instance, the Editable Mesh object implements this to allow the user to select vertices that are animated from the Track View.

**Parameters:**

**int subNum**

The index of the sub-anim that was clicked on.

**Return Value:**

TRUE if implemented; otherwise FALSE. (Track View will call **RedrawViewports()** if something returns TRUE from this method).

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual ParamDimension* GetParamDimension(int i)
```

**Remarks:**

Implemented by the Plug-In.

Returns the type of dimension of the 'i-th' sub-anim. A dimension describes the type and order of magnitude of a sub-anim.

**Parameters:**

**int i**

Specifies the sub-anim (parameter) to return the dimension of.

**Return Value:**

The dimension of the 'i-th' sub-anim (parameter).

**Default Implementation:**

```
{return defaultDim;}
```

**Prototype:**

```
virtual DWORD GetSubAnimCurveColor(int subNum);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Return the suggested color to draw a sub-anim's function curve. For example, the independent X, Y, Z position controller implements this method to return the suggested color for each of its sub-controllers. The Euler Angle Controller uses these so its 3 sub-controllers are drawn in different colors.

**Parameters:**

**int subNum**

The index of the sub-anim.

**Return Value:**

One of the following values:

**PAINTCURVE\_GENCOLOR**

**PAINTCURVE\_XCOLOR**

**PAINTCURVE\_YCOLOR**

**PAINTCURVE\_ZCOLOR**

**Default Implementation:**

```
{return PAINTCURVE_GENCOLOR;}
```

**Prototype:**

```
virtual int SubNumToRefNum(int subNum)
```

**Remarks:**

Implemented by the Plug-In.

This method is used for copying and pasting in the track view. It converts an anim index to a reference index or returns -1 if there is no correspondence. If a client does not wish an anim to be copied or pasted then it can return -1 even if there is a corresponding reference num.

**Parameters:**

**int subNum**

The anim index to return the corresponding reference index of.

**Default Implementation:**

```
{ return -1}
```

**Return Value:**

The reference index corresponding to the anim index passed. Return -1 if there

is no correspondence.

**Prototype:**

**virtual BOOL CanCopyAnim()**

**Remarks:**

Implemented by the Plug-In.

In addition to **SubNumToRefNum()**, if an anim doesn't want to be copied (via Track View or the Edit Modifier Stack 'Copy' button) it can return FALSE from this method, otherwise it can use the default implementation to return TRUE.

**Default Implementation:**

**{return TRUE;}**

**Prototype:**

**int HasSubElements(int type=0);**

**Remarks:**

Implemented by the System.

This method is used to determine if this Animatable has children or sub-anim. The type passed indicates what is tested.

**Parameters:**

**int type=0**

One of the following values:

**0:** Test for node children.

**1:** Test for sub-anim.

**Return Value:**

Nonzero if the item has children or sub-anim; otherwise zero.

**Prototype:**

**virtual BOOL CanDeleteSubAnim(int i);**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

Returns TRUE if the specified sub-anim controller can be deleted; otherwise FALSE.

A new "Delete Controller" button has been added to the Track View toolbar that is enabled when one or more delete-able tracks are selected. This method allows a plug-in to indicate to the Track View that one or more of its sub-controllers are delete-able. This provides a way to allow the user to delete node sub-controllers such as the visibility track, "Image Motion Blur Multiplier", "Object Motion Blur On/Off", etc. If the user selects one of the above-mentioned tracks in the Track View the "Delete Controller" button will become available.

**Parameters:**

**int i**

The zero based index of the sub-anim.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void DeleteSubAnim(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method is called to delete the specified sub-anim controller. See the remarks in **CanDeleteSubAnim()** above.

**Parameters:**

**int i**

The zero based index of the sub-anim.

**Default Implementation:**

```
{}
```

## **Controller Assignment**

**Prototype:**

```
virtual BOOL AssignController(Animatable *control, int
```

**subAnim);**

**Remarks:**

Implemented by the Plug-In.

This method is called to assign the controller to the sub-anim whose index is passed.

**Parameters:**

**Animatable \*control**

The controller to assign.

**int subAnim**

The index of the sub-anim to assign the controller to.

**Default Implementation:**

```
{ return FALSE; }
```

**Return Value:**

Returns TRUE if the controller was assigned; otherwise FALSE.

## **Animation Properties**

**Prototype:**

```
virtual BOOL IsAnimated();
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if this animatable actually has animation; otherwise FALSE.

This method is recursive, so for example, if you call **node->IsAnimated()** it will return TRUE if any aspect of the node is animated; otherwise it will return FALSE.

**Default Implementation:**

The default implementation returns TRUE if a child anim has animation.

## **Make Unique Control**

**Prototype:**

```
virtual BOOL CanMakeUnique()
```

**Remarks:**

Implemented by the Plug-In.

An anim can implement this method to return FALSE to prohibit make unique from being applied to it.

**Default Implementation:**

```
{return TRUE;}
```

## **Flag Access**

**Prototype:**

```
void SetAFlag(int mask)
```

**Remarks:**

Implemented by the System.

Sets one or more of the bits of **aflag**.

**Parameters:**

**int mask**

The flags to set.

**Prototype:**

```
void ClearAFlag(int mask)
```

**Remarks:**

Implemented by the System.

Clears (sets to zero) one or more of the bits of **aflag**.

**Parameters:**

**int mask**

The flags to clear.

**Prototype:**

```
int TestAFlag(int mask)
```

**Remarks:**

Implemented by the System.

Used to test the state of one or more bits of **aflag**.

**Parameters:****int mask**

The flags to test.

**Return Value:**

Nonzero if the flags are set; otherwise 0.

**Prototype:****virtual void FreeCaches()****Remarks:**

Implemented by the Plug-In.

This is called to delete any item that can be rebuilt. For example, the procedural sphere object has a mesh that it caches. It could call **Mesh::FreeAll()** on the mesh from this method. This will free the vertex/face/uv arrays. If the sphere is ever evaluated again it can just rebuild the mesh. If an object (like a sphere) has modifiers applied to it, and those modifiers are not animated, then the result of the pipeline is cached in the node. So there is no reason for the sphere to also have a cache of its representation. Therefore when this method is called, the sphere can free the data of the mesh.

**Default Implementation:****{}****AppData, Interfaces, and Properties.**

The following methods deal with AppData. This is application specific data that may be attached to any Animatable in the scene. With these APIs any 3ds max object (controller, object, node, modifier, material, etc.) can have custom data attached by other objects. These chunks are saved in the .MAX file and can be accessed through the object they are attached to. Sample code using these APIs can be found in **\MAXSDK\SAMPLES\UTILITIES\APPDATA.CPP**.

**Prototype:****void AddAppDataChunk(Class\_ID cid, SClass\_ID sid, DWORD sbid, DWORD len, void \*d);****Remarks:**

Implemented by the System.

This method is used to add an AppDataChunk to this Animatable. The chunk is identified using the Class\_ID, and SuperClassID of the owner, and an ID for sub-chunks.

Note: Developers who want to add appdata to the scene should see the method:

**ReferenceTarget \*Interface::GetScenePointer().**

**Parameters:**

**Class\_ID cid**

The Class\_ID of the owner of the chunk.

**SClass\_ID sid**

The SuperClassID of the owner of the chunk.

**DWORD sbid**

An extra ID that lets the owner identify its sub-chunks.

**DWORD len**

The length of the data in bytes.

**void \*d**

Points to the actual data. The data should be allocated using standard **malloc()** as it will be freed by the system using **free()**.

**Prototype:**

**AppDataChunk \*GetAppDataChunk(Class\_ID cid, SClass\_ID sid, DWORD sbid);**

**Remarks:**

Implemented by the System.

This method is used to retrieve a pointer to an AppDataChunk. The chunk is identified using the Class\_ID, SuperClassID and sub-chunk ID of the owner.

**Parameters:**

**Class\_ID cid**

The Class\_ID of the owner of the chunk.

**SClass\_ID sid**

The SuperClassID of the owner of the chunk.

**DWORD sbid**

An extra ID that lets the owner identify its sub-chunks.

**Return Value:**

A pointer to the AppDataChunk or NULL if it could not be found. See [Class AppDataChunk](#).

**Prototype:**

**BOOL RemoveAppDataChunk(Class\_ID cid, SClass\_ID sid, DWORD sbid);**

**Remarks:**

Implemented by the System.

This method is used to delete an AppDataChunk. The chunk is identified using the Class\_ID, SuperClassID and sub-chunk ID of the owner. Returns TRUE if the data was deleted and FALSE if it could not be found.

**Parameters:**

**Class\_ID cid**

The Class\_ID of the owner of the chunk.

**SClass\_ID sid**

The SuperClassID of the owner of the chunk.

**DWORD sbid**

An extra ID that lets the owner identify its sub-chunks.

**Return Value:**

TRUE if the chunk was removed; otherwise FALSE.

**Prototype:**

**void ClearAllAppData();**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Calling this method will remove **all** the AppData associated with this Animatable. Note: Plugins that call this method will erase **all** appdata chunks, not just their own. Therefore, it is usually more appropriate to use the RemoveAppDataChunk() API to remove AppData associated with a specific Class\_ID.

**Prototype:**

**virtual void\* GetInterface(ULONG id)**

**Remarks:**

Implemented by the Plug-In.

This method provides a mechanism for extending the class in the future. In 3ds max 4.0 there are new interfaces that are accessed by passing an id to this method and it will respond by returning the corresponding interface pointer.

This method has been used however for a different purpose. It currently is used to determine if an object is of a particular class. With controllers for example, there is one base class Control, however there are many super classes (**CTRL\_FLOAT\_CLASS\_ID**, **CTRL\_SCALE\_CLASS\_ID**, etc.). If you wanted to find out if a given Animatable was a controller you would need to compare its SuperClassID to all the known types and only if it wasn't one of the known types could you be sure it wasn't a controller. Having to do this is inconvenient for a developer.

Instead the Control class implements this method. It looks at the id, and if it matches a predefined constant **I\_CONTROL**, it returns its **this** pointer. In this way, given any Animatable, it is easy to find out if it is a controller by simply asking for the control interface. There is a macro that does this:

```
#define GetControlInterface(anim)  
((Control*)anim->GetInterface(I_CONTROL))
```

A plug-in developer may use this macro as follows:

```
Control *c = GetControlInterface(anim);
```

This will either be NULL or a pointer to a valid controller.

This is the list of IDs currently defined for use with this macro. Note that these do NOT need to be released with method **ReleaseInterface()**.

```
#define I_CONTROL 0x00001001  
#define I_MASTER 0x00001010  
#define I_EASELIST 0x00001020  
#define I_MULTLIST 0x00001030  
#define I_BASEOBJECT 0x00001040  
#define I_PARTICLEOBJ 0x00001050  
#define I_KEYCONTROL 0x00001060  
#define I_TEXTOBJECT 0x00001070
```

```
#define I_WAVESOUND 0x00001080
#define I_SUBMTLAPI 0x00001090
#define I_MESHSELECT 0x000010A0
#define I_MESHSELECTDATA 0x000010B0
#define I_MAXSCRIPTPLUGIN 0x000010C0
#define I_MESHDELTAUSER 0x000010D0
#define I_MESHDELTAUSERDATA 0x000010E0
#define I_SPLINESELECT 0x000010F0
#define I_SPLINESELECTDATA 0x00001100
#define I_SPLINEOPS 0x00001110
#define I_PATCHSELECT 0x00001120
#define I_PATCHSELECTDATA 0x00001130
#define I_PATCHOPS 0x00001140
#define I_COMPONENT 0x0000F010
#define I_REFARRAY 0x0000F030
#define I_LINK_TARGET 0x0000F020
#define I_LAYER 0x0000F040
#define I_LAYER_MANAGER 0x0000F050
#define I_REAGENT 0x0000F060
```

Note: Plug-in defined interfaces should be greater than the following value:

```
#define I_USERINTERFACE 0x0000ffff
```

These other macros are defined for a similar purpose:

```
#define GetControlInterface(anim)
    ((Control*)anim->GetInterface(I_CONTROL))
#define GetObjectInterface(anim)
    ((BaseObject*)anim->GetInterface(I_BASEOBJECT))
#define GetParticleInterface(anim)
    ((ParticleObject*)anim->GetInterface(I_PARTICLEOBJ))
#define GetKeyControlInterface(anim)
    ((IKeyControl*)anim->GetInterface(I_KEYCONTROL))
#define GetMasterController(anim)
    ((ReferenceTarget*)anim->GetInterface(I_MASTER))
#define GetTextObjectInterface(anim)
    ((ITextObject*)anim->GetInterface(I_TEXTOBJECT))
```

```

#define GetWaveSoundInterface(anim)
    ((IWaveSound*)anim->GetInterface(I_WAVESOUND))
#define GetMeshSelectInterface(anim)
    ((IMeshSelect*)anim->GetInterface(I_MESHSELECT))
#define GetMeshSelectDataInterface(anim)
    ((IMeshSelectData*)anim-
    >GetInterface(I_MESHSELECTDATA))
#define GetMeshDeltaUserInterface(anim)
    ((MeshDeltaUser*)anim-
    >GetInterface(I_MESHDELTAUSER))
#define GetMeshDeltaUserDataInterface(anim)
    ((MeshDeltaUserData*)anim-
    >GetInterface(I_MESHDELTAUSERDATA))

```

If a plug-in implements this method for its own purposes, it would, in general, switch on the id and if it is not aware of the id it would call this method on the base class. Otherwise it could respond to the id as it needed. See the sample code below for the how the Control class implements this method.

**Parameters:**

**ULONG id**

The id of the interface.

**Default Implementation:**

```
{ return NULL; }
```

**Sample Code:**

The following is the Control class implementation of this method. It looks at the **id** passed, and if it matches **I\_CONTROL** it returns its **this** pointer. Otherwise it calls the base class method.

```

void* Control::GetInterface(ULONG id)
{
    if (id==I_CONTROL) {
        return this;
    } else {
        return Animatable::GetInterface(id);
    }

```

}

**Prototype:**

**virtual void ReleaseInterface(ULONG id,void \*i)**

**Remarks:**

Implemented by the Plug-In.

This method is not currently used. It is reserved for future use. Its purpose is for releasing an interface created with **GetInterface()**.

**Prototype:**

**virtual BaseInterface\* GetInterface(Interface\_ID id);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the Base Interface for the interface ID passed. The default implementation of this method retrieves this information from the ClassDesc for the plug-in.

Any future object-based interfaces should be allocated unique Interface\_IDs (you can use Gencid.exe for this) and made available through this call.

The default implementation of GetInterface(Interface\_ID) looks up a standalone interface of the given ID on the object's ClassDesc. This gives access to standalone interfaces via any of a plug-in's objects, without having to dig around for the ClassDesc, so you should fall back to calling the default implementation if you don't recognize an ID in your implementation of GetInterface(Interface\_ID).

**Parameters:**

**Interface\_ID id**

The unique ID of the interface to get. See [Class Interface ID](#).

**Prototype:**

**virtual void ReleaseInterface(Interface\_ID id, void\* i);**

**Remarks:**

This method is available in release 4.0 and later only.

Releases the interface retrieved above.



```

JointParamsPath *jp = (JointParamsPath*)GetProperty(id);
if (jp) {
    *jp = *((JointParamsPath*)data);
    delete (JointParamsPath*)data;
 } else {
    aprops.Append(1,(AnimProperty**)&data);
 }
 }
return 1;
 } else
if (id==PROPID_INTERPUI) {
    if (!data) {
        int index = aprops.FindProperty(id);
        if (index>=0) {
            aprops.Delete(index,1);
         }
     } else {
        InterpCtrlUI *ui = (InterpCtrlUI*)GetProperty(id);
        if (ui) {
            *ui = *((InterpCtrlUI*)data);
         } else {
            aprops.Append(1,(AnimProperty**)&data);
         }
     }
return 1;
 } else {
        return Animatable::SetProperty(id,data);
     }
 }

```

**Prototype:**

**virtual void \*GetProperty(ULONG id)**

**Remarks:**

Implemented by the Plug-In.

This method is used to retrieve a property specified by the id passed (as stored by **SetProperty()**).

Note for 3ds max version 1.1:

Two new property IDs have been added:

**PROPID\_CLEARCACHES:** When passed to a texture map or material, the material should dump any of its caches. For example, the bitmap texture responds to this by freeing the bitmap from memory. For sample code see

**\\MAXSDK\\SAMPLES\\MATERIALS\\BMTEX.CPP.**

**PROPID\_HAS\_WSM:** When passed to an INode, will return TRUE if the node has World Space Modifiers applied to it or FALSE if it does not. For sample code see

**\\MAXSDK\\SAMPLES\\IMPEXP\\3DSEXP.CPP.**

Note for 3ds max version 1.2:

A new **id** has been created and assigned the constant:

**#define PROPID\_EVAL\_STEPSIZE\_BUG\_FIXED 0x1000.**

This only effects the evaluation of objects when rendering them using motion blur. Motion blur works by evaluating the object numerous times (at fractions of a frame apart) and combining these images by blending them together.

Originally, 3ds max would make these evaluations in reverse order within a sub-frame -- from the last one, to the second to the last one, back to the first one. There is a problem with this for certain plug-ins that need to compute their state from time 0 forward. For these objects, the backwards approach may be too computationally intensive.

Both the forward and backward approaches exist in 3ds max and the developer may choose which method to use. 3ds max interrogates the object to see how it should handle the evaluation process -- either going backwards or forwards. It calls this method with **id** set to the constant **PROPID\_EVAL\_STEPSIZE\_BUG\_FIXED**. If a plug-in implements this method to return nonzero, it means the plug-in works correctly using forward stepping, and 3ds max will use that approach. If a plug-in does not implement this method and handle the **id** of **PROPID\_EVAL\_STEPSIZE\_BUG\_FIXED** it will return the default value of zero. This means the older method of backwards evaluation will be used.

Therefore, a plug-in object that wants to handle motion blur using forward

stepping should implement this method, and if passed an **id** of **PROPID\_EVAL\_STEPSIZE\_BUG\_FIXED**, should return nonzero.

**Parameters:**

**ULONG id**

The id of the property to retrieve.

**Default Implementation:**

```
{ return NULL; }
```

**Sample Code:**

This code is from

**\MAXSDK\SAMPLES\CONTROLLERS\PATHCTRL.CPP**. It is used to restore the inverse kinematics user interface parameters of the path controller. It retrieves the property data on the **aprops** property list. See the Data Members at the beginning of Animatable for details on **aprops**.

```
void* PathPosition::GetProperty(ULONG id)
{
    if (id==PROPID_INTERPUI || id==PROPID_JOINTPARAMS) {
        int index = aprops.FindProperty(id);
        if (index>=0) {
            return aprops[index];
        } else {
            return NULL;
        }
    } else {
        return Animatable::GetProperty(id);
    }
}
```

## Clipboard Methods

**Prototype:**

```
virtual BOOL CanCopyTrack(Interval iv, DWORD flags)
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if this item can copy its data over the specified range; otherwise returns FALSE.

**Parameters:**

**Interval iv**

The interval of time that would be copied.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

**TIME\_INCRIGHT**

Include the right endpoint.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL CanPasteTrack(TrackClipObject *cobj,Interval iv,  
DWORD flags)
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if this item can paste its data over the specified range; otherwise returns FALSE.

**Parameters:**

**TrackClipObject \*cobj**

The clipboard object that would be pasted. The item should look at the SuperClassID and Class\_ID of the creator of the clip object to determine if it is a suitable object to paste. See [Class TrackClipObject](#).

**Interval iv**

The interval of time that would be pasted.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

## **TIME\_INCRIGHT**

Include the right endpoint.

### **Default Implementation:**

```
{return FALSE;}
```

### **Prototype:**

```
virtual TrackClipObject *CopyTrack(Interval iv, DWORD flags)
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to copy the item's track data over the specified interval.

### **Parameters:**

#### **Interval iv**

The interval of time over which to copy the track data.

#### **DWORD flags**

One or more of the following values:

#### **TIME\_INCLEFT**

Include the left endpoint.

#### **TIME\_INCRIGHT**

Include the right endpoint.

### **Return Value:**

The item should return an instance of a class derived from TrackClipObject that contains the data for the item. See [Class TrackClipObject](#).

### **Default Implementation:**

```
{return NULL;}
```

### **Prototype:**

```
virtual void PasteTrack(TrackClipObject *cobj, Interval iv,  
DWORD flags)
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to paste the specified clip object to this track. This method will not be called unless **CanPasteTrack()** returned TRUE.

**Parameters:**

**TrackClipObject \*cobj**

The data to paste.

**Interval iv**

The interval of time to paste.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

**TIME\_INCRIGHT**

Include the right endpoint.

**Prototype:**

**virtual BOOL CanCopySubTrack(int subNum, Interval iv, DWORD flags);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

If **CanCopyTrack()** returns FALSE then this method is called on the sub-anim (passing the sub number).

This is used in particular for Parameter Blocks. In that case, if there is no controller plugged into the track, the copying and pasting of controllers can't be done (since there is no controller). However, this method allows the Parameter Block to handle it.

**Parameters:**

**int subNum**

Specifies the sub-anim to check.

**Interval iv**

The interval of time over which to copy the track data.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

## **TIME\_INCRIGHT**

Include the right endpoint.

### **Return Value:**

TRUE if the specified item can copy its data over the specified range; otherwise FALSE.

### **Default Implementation:**

```
{return FALSE;}
```

### **Prototype:**

```
virtual BOOL CanPasteSubTrack(int subNum, TrackClipObject  
*cobj, Interval iv, DWORD flags);
```

### **Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Returns TRUE if the specified item can paste its data over the specified range; otherwise returns FALSE.

Plug-ins can implement pasting for cases where their sub-anim's don't implement it. An example of this is the Parameter Block class. It implements this method to allow pasting parameters that don't have controllers assigned to them. These aren't called on the client unless the sub-anim doesn't implement **CanPasteTrack()**.

### **Parameters:**

**int subNum**

Specifies the sub-anim to check.

**TrackClipObject \*cobj**

The data to paste.

**Interval iv**

The interval of time to paste.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

**TIME\_INCRIGHT**

Include the right endpoint.

**Return Value:**

TRUE if the specified item can paste its data over the specified range;  
otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual TrackClipObject *CopySubTrack(int subNum, Interval iv,  
DWORD flags);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

This method is called to copy the specified sub anim's track data over the specified interval.

**Parameters:**

**int subNum**

The number of the sub-anim to copy.

**Interval iv**

The interval of time over which to copy the track data.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

**TIME\_INCRIGHT**

Include the right endpoint.

**Return Value:**

The item should return an instance of a class derived from TrackClipObject that contains the data for the item. See [Class TrackClipObject](#).

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual void PasteSubTrack(int subNum, TrackClipObject  
*cobj, Interval iv, DWORD flags);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

This method is called to paste the specified clip object to the specified sub-anim track.

**Parameters:**

**int subNum**

The number of the sub-anim to paste.

**TrackClipObject \*cobj**

The data to paste.

**Interval iv**

The interval of time to paste.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

**TIME\_INCRIGHT**

Include the right endpoint.

**Default Implementation:**

```
{}
```

**Interactive Adjustment****Prototype:**

```
virtual void MouseCycleCompleted(TimeValue t);
```

**Remarks:**

This method is available in release 2.0 and later only (previously in Class Control in 1.x).

Implemented by the Plug-In.

This method is called on whatever controller the user is modifying with the

mouse -- when the mouse button is released. For example when the user selects a node in the viewports, then drags, then releases the mouse button, this method is called. This method will also be called when the user clicks on a key in the track view and lets up. If a controller performs extensive calculation in its evaluation this method is handy. The controller could perhaps perform a simplified calculation during interactive adjustment of a node. Then when the user releases the mouse button this method is called and the extensive calculations are performed.

The default implementation of this method is recursive so it gets called on all sub-animis affected by a range bar operation.

**Parameters:**

**TimeValue t**

The time the mouse was released.

**Prototype:**

**virtual void MouseCycleStarted(TimeValue t);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

This method is called on whatever controller the user is modifying with the mouse -- when the mouse button is pressed.

The default implementation of this method is recursive so it gets called on all sub-animis affected by a range bar operation.

**Parameters:**

**TimeValue t**

The time the mouse was first pressed.

**Methods called when rendering is started and finished**

**Prototype:**

**virtual int RenderBegin(TimeValue t, ULONG flags=0)**

**Remarks:**

Implemented by the Plug-In.

This method is called once at the beginning of each render. A plug-in can use this method to do any work required before a rendering actually begins. For example, some of the standard 3ds max plug-ins use this method to toggle between their 'viewport' state and the 'rendering' state. The Optimize modifier has two settings, one for the viewports and one for the rendering. When this method is called it then performs the switch from viewport to render.

**Parameters:**

**TimeValue t**

The time that the render is beginning.

**ULONG flags=0**

This is not used in 3ds max 1.x.

In 3ds max 2.0 and later the following flag value may be checked:

**RENDERBEGIN\_IN\_MEDIT**

Indicates that the render is occurring in the Material Editor.

**Return Value:**

Nonzero if the method is implemented; otherwise 0.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual int RenderEnd(TimeValue t)
```

**Remarks:**

Implemented by the Plug-In.

This method is called once at the end of each render.

**Parameters:**

**TimeValue t**

The time of the last rendered frame.

**Return Value:**

Nonzero if the method is implemented; otherwise 0.

**Default Implementation:**

```
{ return 0; }
```

## System Plug-In Related Methods

### Prototype:

**virtual void GetSystemNodes(INodeTab &nodes, SysNodeContext)**

### Remarks:

Implemented by the Plug-In.

The master controller of a system plug-in should implement this method to give 3ds max a list of nodes that are part of the system. The master controller should fill in the given table with the INode pointers of the nodes that are part of the system. This will ensure that operations like cloning and deleting affect the whole system.

Said another way, **GetSystemNodes()** should be implemented for the master controller of a system, and should return a list of pointers to all nodes that are part of the system. **GetInterface()** should be implemented for the slave TM controllers of the system and return a pointer to the master controller.

3ds max will use **GetInterface()** in the TM controller of each selected node to retrieve the master controller and then call **GetSystemNodes()** on the master controller to get the list of nodes.

### Parameters:

#### **INodeTab &nodes**

The table of nodes that are part of the system.

#### **SysNodeContext**

This parameter is available in release 4.0 and later only.

Previously, this method gathered related (system) nodes during cloning, deleting, file merging and saving.

This parameter can be used to specify the context under that the "system nodes" are used. These are; **kSNCClone**, **kSNCCDelete**, **kSNCFFileMerge**, and **kSNCFFileSave**.

### Default Implementation:

```
{}
```

**The following methods deal with operations to Keys and Track View:**

**Prototype:****virtual BOOL BypassTreeView()****Remarks:**

Implemented by the Plug-In.

This method indicates to the system that this anim should not appear in the Track View. Note: Track View was formally referred to as Tree View. This is what parameter blocks do for example. They don't show up in track view, just their sub-anim's do. This prevents the extra level of the parameter block from appearing.

**Return Value:**

Return TRUE to not appear in the Track View. Note that if you return TRUE your children will appear in the track view regardless.

**Default Implementation:****{ return FALSE; }****Prototype:****virtual BOOL BypassTrackBar();****Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method indicates to the system that this anim should not appear in the Track Bar. The anim won't show up in the Track Bar, just its sub-anim's will. This function is similar to BypassTreeView(), but refers to the Track Bar instead of the Track View.

**Return Value:**

Return TRUE to not appear in the Track Bar. Note that if you return TRUE your children will appear in the Track Bar regardless.

**Default Implementation:****{ return BypassTreeView(); }****Prototype:****virtual BOOL BypassPropertyLevel();**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

Use this method in order to cause parameters in this Animatable (as a sub-anim) to appear to reside at the level of the parent Animatable in the scripiter. Return TRUE and this Animatable won't appear as a property in the scripiter however it's sub-anim's children will. The default implementation returns FALSE indicating it will appear normally.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual BOOL InvisibleProperty();
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method controls the visibility of this Animatable and all of its sub-anim's to appear as properties in the scripiter. Return TRUE and it won't nor will its sub-anim's. Returning FALSE (the default implementation) causes this Animatable and its sub-anim's to appear as normal.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual int NumKeys()
```

**Remarks:**

Implemented by the Plug-In.

This method returns the number of keys managed by the plug-in, or **NOT\_KEYFRAMEABLE** if it does not work with keys.

**Default Implementation:**

```
{return NOT_KEYFRAMEABLE;}
```

**Prototype:**

**virtual TimeValue GetKeyTime(int index)**

**Remarks:**

Implemented by the Plug-In.

This method returns the time of the key specified by **index**.

**Parameters:**

**int index**

Specifies the key whose time should be returned.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

**virtual int GetKeyIndex(TimeValue t)**

**Remarks:**

Implemented by the Plug-In.

Returns the index of the key at time **t** or **-1** if no key is found at the specified time.

**Parameters:**

**TimeValue t**

Specifies the time at which to retrieve the key index.

**Default Implementation:**

```
{return -1;}
```

**Prototype:**

**virtual BOOL GetNextKeyTime(TimeValue t,DWORD flags,TimeValue &nt)**

**Remarks:**

Implemented by the Plug-In.

An item should implement this method to allow the Key Mode button in 3ds max's UI to function properly. If Key Mode is set, and the user clicks the Previous Key or Next Key button, this method will be called to retrieve the next or previous key.

**Parameters:****TimeValue t**

The current time (frame slider position).

**DWORD flags**

One or more of the following value:

**NEXTKEY\_LEFT**

Search to the left.

**NEXTKEY\_RIGHT**

Search to the right.

**NEXTKEY\_POS**

Next position key.

**NEXTKEY\_ROT**

Next rotation key.

**NEXTKEY\_SCALE**

Next scale key.

**TimeValue &nt**

The time of the previous or next key is returned here.

**Return Value:**

TRUE if the key time was retrieved; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE;}
```

**Prototype:**

```
virtual void CopyKeysFromTime(TimeValue src,TimeValue  
dst,DWORD flags)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to copy or interpolate a new key from a source time to a destination time.

**Parameters:****TimeValue src**

The source time.

**TimeValue dst**

The destination time.

**DWORD flags**

These filter flags are passed to a transform (Matrix3) controller. The TM can decide what to do with them. They have obvious meaning for the PRS controller. One or more of the following values:

**COPYKEY\_POS**

Copy the position key.

**COPYKEY\_ROT**

Copy the rotation key.

**COPYKEY\_SCALE**

Copy the scale key.

**Prototype:**

**virtual void DeleteKeyAtTime(TimeValue t)**

**Remarks:**

Implemented by the Plug-In.

This method is called to delete the key at the specified time.

**Parameters:****TimeValue t**

Specifies the time to delete the key.

**Default Implementation:**

```
{}
```

**Prototype:**

**virtual BOOL IsKeyAtTime(TimeValue t,DWORD flags)**

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if there is a key of the specified type at the specified time; otherwise FALSE.

**Parameters:****TimeValue t**

Specifies the time to check for a key.

### **DWORD flags**

One or more of the following values:

**KEYAT\_POSITION**

**KEYAT\_ROTATION**

**KEYAT\_SCALE**

### **Default Implementation:**

```
{return FALSE;}
```

### **Prototype:**

```
virtual int GetKeyTimes(Tab<TimeValue> &times,Interval  
range,DWORD flags)
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to build a table of time values, one time for each key within the interval passed. The plug-in should load up the table passed with the time of each key present over the specified time range.

### **Parameters:**

**Tab<TimeValue> &times**

The table of time values to build. See [Class Tab](#).

**Interval range**

The range of time over which to retrieve the key times. See [Class Interval](#).

**DWORD flags**

One of the following values:

**KEYAT\_POSITION** - Return for Position keys only.

**KEYAT\_ROTATION** - Return for Rotation keys only.

**KEYAT\_SCALE** - Return for Scale keys only.

### **Return Value:**

The plug-in should return an offset so the system can access the keys using an index. Thus it should return the number of keys skipped because their times were before **range.Start()**. For example, say the first keyframe in the interval passed was actually the third key overall. The plug-in should return 2 (two keys preceded the first one stored). In this way, the system can access the key

as the i-th key in the table plus 2.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual int GetKeySelState(BitArray &sel,Interval range,DWORD flags)
```

**Remarks:**

Implemented by the Plug-In.

When this method is called, the plug-in should update the BitArray **sel** to indicate if its keys present in the interval passed are selected or deselected.

**Parameters:**

**BitArray &sel**

The bit array to update, one bit for each key within the interval **range**. If the key is selected, the corresponding bit should be 1, otherwise it should be 0. See [Class BitArray](#).

**Interval range**

The range of time over which to retrieve the key selected state. [Class Interval](#).

**DWORD flags**

One or more of the following values:

**KEYAT\_POSITION** - Return for Position keys only.

**KEYAT\_ROTATION** - Return for Rotation keys only.

**KEYAT\_SCALE** - Return for Scale keys only.

Note: If the flags are passed as 0, use ALL keys within the range.

**Return Value:**

The number of keys skipped because their times were before **range.Start()**.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
void OpenTreeEntry(int type, DWORD tv)
```

**Remarks:**

Implemented by the System.

This method may be called to open the specified Track View entry. The type parameter indicates if the child tree or the sub-anim (parameter) tree is opened.

**Parameters:**

**int type**

This value may be either 0 or 1. If 0, the child tree is opened. If 1, the sub-anim tree is opened.

**DWORD tv**

This parameter is available in release 2.0 and later only.

It specifies which Track View(s) are altered, one bit for each Track View. In 3ds max 2.0 the open/closed state is independent for each Track View. The low-order 16 bits represent the 16 track views.

**Prototype:**

**void CloseTreeEntry(int type, DWORD tv)**

**Remarks:**

Implemented by the System.

This method may be called to close the specified Track View entry. The type parameter indicates if the child tree or the sub-anim tree is closed.

**Parameters:**

**int type**

This value may be either 0 or 1. If 0, the child tree is closed. If 1, the sub-anim (parameter) tree is closed.

**DWORD tv**

This parameter is available in release 2.0 and later only.

It specifies which Track View(s) are altered, one bit for each Track View. In 3ds max 2.0 the open/closed state is independent for each Track View. The low-order 16 bits represent the 16 track views.

**Prototype:**

**int IsTreeEntryOpen(int type, DWORD tv);**

**Remarks:**

Implemented by the System.

Returns nonzero if the specified tree is opened for this item, and zero if it is closed.

**Parameters:**

**int type**

This value may be either 0 or 1. If 0, the child tree is checked. If 1, the sub-anim (parameter) tree is checked.

**DWORD tv**

This parameter is available in release 2.0 and later only.

Specifies which Track View to check -- one bit per Track View.

**Prototype:**

**BOOL GetSelInTrackView(DWORD tv);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Returns TRUE if this animatable is selected in the specified Track View; FALSE if not selected.

**Parameters:**

**DWORD tv**

Specifies which Track View to check -- one bit per Track View.

**Prototype:**

**void SetSelInTrackView(DWORD tv, BOOL sel);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Sets the state of this animatable to selected or deselected in the specified Track View.

**Parameters:**

**DWORD tv**

Specifies which Track View to check -- one bit per Track View.

## **BOOL sel**

TRUE to select; FALSE to deselect.

### **Prototype:**

**BOOL InTrackViewSelSet(int which);**

### **Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Returns TRUE if this animatable is in the specified selection set; otherwise FALSE.

### **Parameters:**

**int which**

Indicates the Track View selection set to check -- this should be  $\geq 0$  and  $< \text{MAX\_TRACKVIEW\_SELSETS}$

### **Prototype:**

**void SetTrackViewSelSet(int which, BOOL inOut);**

### **Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Sets the selected or deselected state of this animatable in the specified selection set.

### **Parameters:**

**int which**

Indicates the Track View selection set to modify -- this should be  $\geq 0$  and  $< \text{MAX\_TRACKVIEW\_SELSETS}$

**BOOL inOut**

TRUE for in; FALSE for out.

## **Operations to a selected block of time.**

### **Prototype:**

## **virtual Interval GetTimeRange(DWORD flags);**

### **Remarks:**

Implemented by the System.

Returns an interval representing the tracks time range, based on the flags passed.

### **Parameters:**

#### **DWORD flags**

One or more of the following values:

##### **TIMERANGE\_SELONLY**

The bounding interval of selected keys only.

##### **TIMERANGE\_ALL**

Whatever the channel's time range is - usually the bounding interval of all keys.

##### **TIMERANGE\_CHILDNODES**

The node's time range should include its child nodes.

##### **TIMERANGE\_CHILDANIMS**

A animatable's child anim ranges should be included.

### **Return Value:**

An interval representing the tracks time range.

### **Prototype:**

#### **virtual void EditTimeRange(Interval range,DWORD flags)**

### **Remarks:**

Implemented by the Plug-In.

This method is called to change the range of the anim (usually a controller) to the given range. This is the range that is used to compute the Out of Range Types. For example, this method may be called when the user is working in Position Range mode in the Track View.

Keyframe controllers generally support this method. Other controllers may or may not support this method. For example, a procedural controller may want to maintain a range upon which the animation is based. The user may then move the range bar around to move the procedural animation around.

The range passed is the range used to compute the Out of Range Types. This

may be used for example with the Loop ORT to extend the range, either past the last key or before the first key, so there is some time to loop back to the start.

The 3ds max keyframe controllers maintain an interval that is their range. It is normally defined to be the first key to the last key. If the user goes into Position Range mode and moves the range around, this method is called. The keyframe controllers set a flag to indicate that the range is no longer linked to the first key or the last key. Then the range is stored in the interval, and this is considered the 'in range' portion of the controller. If time is evaluated outside of this range it applies the ORTs to determine the value.

**Parameters:**

**Interval range**

The new range for the anim.

**DWORD flags**

**EDITRANGE\_LINKTOKEYS**

If this flag is set, the controller should re-establish the link between the start and end keys and its range. This is passed if the user presses the link to keys button in Track View. Thus, if one of the ends of the interval is at a key, link it to the key so that if the key moves, the interval moves.

**Default Implementation:**

{ }

**Prototype:**

**virtual void DeleteTime(Interval iv, DWORD flags);**

**Remarks:**

Implemented by the Plug-In.

This method is called to delete the specified interval of time (or the keys within the interval).

**Parameters:**

**Interval iv**

The interval of time to delete.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

**TIME\_INCRIGHT**

Include the right endpoint.

**TIME\_NOSLIDE**

Delete any keys in the interval but don't actually remove the block of time.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual void ReverseTime(Interval iv, DWORD flags);
```

**Remarks:**

Implemented by the Plug-In.

This method is called to reverse the data within the specified interval. For example, if the interval passed is from frame 10 to 20, and there is a key at frame 12, the key should be moved to frame 18. Considered another way, if all the times were normalized, and there was a value **n** between 0 and 1, **n** should be changed to **1-n**.

**Parameters:****Interval iv**

The interval of time over which to reverse the data.

**DWORD flags**

One or more of the following values:

**TIME\_INCLEFT**

Include the left endpoint.

**TIME\_INCRIGHT**

Include the right endpoint.

**Default Implementation:**

```
{
```

**Sample Code:**

```
INTERP_CONT_TEMPLATE
```

```
void InterpControl<INTERP_CONT_PARAMS>::ReverseTime(
```

```

Interval iv,
DWORD flags )
{
Interval test = TestInterval(iv,flags);
int n = keys.Count();
HoldTrack();
for ( int i = 0; i < n; i++ ) {
if (keys[i].TimeLocked()) continue;
if ( test.InInterval(keys[i].time) ) {
TimeValue delta = keys[i].time - iv.Start();
keys[i].time = iv.End()-delta;
}
}
keys.KeysChanged();
keys.CheckForDups();
ivalid.SetEmpty();
NotifyDependents(FOREVER, PART_ALL, REFMSG_CHANGE);
}

```

**Prototype:**

**virtual void ScaleTime(Interval iv, float s);**

**Remarks:**

Implemented by the Plug-In.

This method is called to scale an interval of time by the specified scale factor.

**Parameters:**

**Interval iv**

The interval of time to scale. The origin of the scale is at **iv.Start()**.

**float s**

The scale factor for the time.

**Default Implementation:**

**{}**

**Sample Code:**

**INTERP\_CONT\_TEMPLATE**

**void InterpControl<INTERP\_CONT\_PARAMS>::ScaleTime( Interval**

```

iv, float s)
{
int n = keys.Count();
TimeValue delta =
int(s*float(iv.End()-iv.Start())) + iv.Start()-iv.End();
HoldTrack();
for ( int i = 0; i < n; i++ ) {
if (keys[i].TimeLocked()) continue;
if ( iv.InInterval(keys[i].time) ) {
keys[i].time =
int(s*float(keys[i].time - iv.Start())) + iv.Start();
} else
if (keys[i].time > iv.End()) {
keys[i].time += delta;
}
}
keys.KeysChanged();
ivalid.SetEmpty();
NotifyDependents(FOREVER, PART_ALL, REFMSG_CHANGE);
}

```

**Prototype:**

**virtual void InsertTime(TimeValue ins, TimeValue amount)**

**Remarks:**

Implemented by the Plug-In.

This method is called to insert the specified amount of time at the specified insertion point.

**Parameters:**

**TimeValue ins**

The time to begin the insertion.

**TimeValue amount**

The amount of time to insert.

**Default Implementation:**

**{}**

**Prototype:**

**virtual BOOL SupportTimeOperations()**

**Remarks:**

Implemented by the Plug-In.

If an anim supports time operations in the track view (cut, copy, paste, etc.), it should implement this method to return TRUE. When it is FALSE the user cannot select blocks of time in the anim's track.

**Default Implementation:**

**{return FALSE;}**

**Prototype:**

**virtual void MapKeys(TimeMap \*map, DWORD flags);**

**Remarks:**

Implemented by the Plug-In.

The method is called to update the keys specified by the flags, using the TimeMap passed. The plug-in should go through the specified keys and change their **time** to **TimeMap::map(time)**. See the sample code below for how this is done.

**Parameters:**

**TimeMap \*map**

This class provides a method, **map()**, that is applied to the keys. See [Class TimeMap](#).

**DWORD flags**

The flags indicate the keys to operate on. One or more of the following values:

**TRACK\_DOSEL**

Selected keys only.

**TRACK\_DOALL**

All the keys, ignore their selection state.

**TRACK\_SLIDEUNSEL**

Slide unselected keys to the right. Keys are slid by the amount the last key was transformed.

**TRACK\_RIGHTTOLEFT**

Enumerate right to left. If **TRACK\_SLIDEUNSEL** is set, keys will

slide to the left.

### **TRACK\_DOSUBANIMS**

Sub-Animatables keys as well.

### **TRACK\_DOCHILDNODES**

Child Nodes keys as well

### **TRACK\_MAPRANGE**

The range, if not locked to first and last key, should be mapped as well.

#### **Sample Code:**

#### **INTERP\_CONT\_TEMPLATE**

```
void InterpControl<INTERP_CONT_PARAMS>::MapKeys(TimeMap  
*map,DWORD flags )
```

```
{  
int n = keys.Count();  
BOOL changed = FALSE;  
if (!n) goto doneMapKeys;  
HoldTrack();  
if (flags&TRACK_DOALL) {  
for (int i = 0; i < n; i++) {  
if (keys[i].TimeLocked()) continue;  
keys[i].time = map->map(keys[i].time);  
changed = TRUE;  
}  
} else  
if (flags&TRACK_DOSEL) {  
BOOL slide = flags&TRACK_SLIDEUNSEL;  
TimeValue delta = 0, prev;  
int start, end, inc;  
if (flags&TRACK_RIGHTTOLEFT) {  
start = n-1;  
end = -1;  
inc = -1;  
} else {  
start = 0;  
end = n;  
inc = 1;  
}  
}
```

```

for (int i = start; i != end; i += inc) {
    if (keys[i].TimeLocked()) continue;
    if (keys[i].TestKFlag(KEY_SELECTED)) {
        prev = keys[i].time;
        keys[i].time = map->map(keys[i].time);
        delta = keys[i].time - prev;
        changed = TRUE;
    } else if (slide) {
        keys[i].time += delta;
    }
}
}
if (flags&TRACK_MAPRANGE &&
keys.TestTFlag(RANGE_UNLOCKED)) {
    TimeValue t0 = map->map(keys.range.Start());
    TimeValue t1 = map->map(keys.range.End());
    keys.range.Set(t0,t1);
}
if (changed) {
    keys.KeysChanged();
    ivalid.SetEmpty();
    NotifyDependents(FOREVER, PART_ALL, REFMSG_CHANGE);
}
doneMapKeys:
Animatable::MapKeys(map,flags);
}

```

**Prototype:**

**virtual void DeleteKeys(DWORD flags)**

**Remarks:**

Implemented by the Plug-In.

This method is called to delete keys, as specified by the flags passed.

**Parameters:**

**DWORD flags**

One or more of the following values:

### **TRACK\_DOSEL**

Delete selected keys only.

### **TRACK\_DOALL**

Delete all keys (ignore selection state).

### **TRACK\_SLIDEUNSEL**

Slide unselected keys to the right.

### **TRACK\_RIGHTTOLEFT**

Enumerate right to left. If **TRACK\_SLIDEUNSEL** is set, keys will slide to the left.

#### **Default Implementation:**

```
{}
```

#### **Prototype:**

```
virtual void DeleteKeyByIndex(int index);
```

#### **Remarks:**

Implemented by the Plug-In.

Deletes the key specified by the index passed.

#### **Parameters:**

**int index**

The index of the key to delete.

#### **Default Implementation:**

```
{}
```

#### **Prototype:**

```
virtual void SelectKeys(TrackHitTab& sel, DWORD flags);
```

#### **Remarks:**

Implemented by the Plug-In.

This method is called to select or deselect a set of keys identified by the **TrackHitTab** and the specified flags.

#### **Parameters:**

**TrackHitTab& sel**

The table of track hit records. See [Class TrackHitRecord](#) and [Class Tab](#). Note

the following: **typedef Tab<TrackHitRecord> TrackHitTab;**

### **DWORD flags**

Either **SELKEYS\_SELECT**, **SELKEYS\_DESELECT**, or a combination of **SELKEYS\_CLEARKEYS** and **SELKEYS\_CLEARCURVE** will be specified.

One or more of the following values:

#### **SELKEYS\_SELECT**

The keys should be selected.

#### **SELKEYS\_DESELECT**

The keys should be deselected.

#### **SELKEYS\_CLEARKEYS**

All keys should be deselected.

#### **SELKEYS\_CLEARCURVE**

All keys on the function curve should be deselected.

#### **SELKEYS\_FCURVE**

Indicates that we are operating on the keys of a function curve, and not of a track.

### **Default Implementation:**

**{}**

### **Prototype:**

**virtual void SelectSubKeys(int subNum, TrackHitTab& sel, DWORD flags);**

### **Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

This method is called on the client when the client takes over control of an anims function curves. It's called to select or deselect a set of keys identified by the **TrackHitTab** and the specified flags.

### **Parameters:**

#### **int subNum**

The index of the sub-anim to select or deselect

## **TrackHitTab& sel**

The table of track hit records. See [Class TrackHitRecord](#) and [Class Tab](#). Note the following: **typedef Tab<TrackHitRecord> TrackHitTab;**

### **DWORD flags**

Either **SELKEYS\_SELECT**, **SELKEYS\_DESELECT**, or a combination of **SELKEYS\_CLEARKEYS** and **SELKEYS\_CLEARCURVE** will be specified.

One or more of the following values:

#### **SELKEYS\_SELECT**

The keys should be selected.

#### **SELKEYS\_DESELECT**

The keys should be deselected.

#### **SELKEYS\_CLEARKEYS**

All keys should be deselected.

#### **SELKEYS\_CLEARCURVE**

All keys on the function curve should be deselected.

#### **SELKEYS\_FCURVE**

Indicates that we are operating on the keys of a function curve, and not of a track.

### **Default Implementation:**

```
{
```

### **Prototype:**

```
virtual void SelectSubCurve(int subNum,BOOL sel);
```

### **Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

This method is called to set the selected state of the sub-curve whose index is passed.

### **Parameters:**

**int subNum**

The index of the sub-anim to select or deselect

**BOOL sel**

TRUE to select the curve; FALSE to deselect it.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual void SelectKeyByIndex(int i, BOOL sel);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

This method is called to set the selected state of the key whose index is passed.

**Parameters:****int i**

The key to select or deselect.

**BOOL sel**

TRUE to select the key; FALSE to deselect it.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual void FlagKey(TrackHitRecord hit);
```

**Remarks:**

Implemented by the Plug-In.

This method is called to have the plug-in flag or mark a specific key identified by the TrackHitRecord.

As an example, when the user goes to move a selection set of keys in the Track View, a yellow marker is drawn. To move the group of keys, the user clicks on a single one. The system needs to track this one key as it is moved, and needs a way to identify it. This method is called so the developer can flag this key as the one that was selected. This is needed because the Track View doesn't know anything about a specific controllers ordering of keys and thus

cannot refer to it by index.

The system will call **GetFlagKeyIndex()** (described below) to retrieve the index of the key that was flagged.

**Parameters:**

**TrackHitRecord hit**

The developer uses this class to identify a key. This is the hit record that the controller gave the Track View in the first place to identify the hit. Thus this is enough information to identify the key. See [Class TrackHitRecord](#).

**Default Implementation:**

```
{}
```

**Sample Code:**

```
INTERP_CONT_TEMPLATE
void
InterpControl<INTERP_CONT_PARAMS>::FlagKey(TrackHitRecord
hit)
{
    int n = keys.Count();
    for ( int i = 0; i < n; i++ ) {
        keys[i].ClearKFlag(KEY_FLAGGED);
    }
    assert(hit.hit>=0&&hit.hit<(DWORD)n);
    keys[hit.hit].SetKFlag(KEY_FLAGGED);
}
```

**Prototype:**

```
virtual int GetFlagKeyIndex();
```

**Remarks:**

Implemented by the Plug-In.

Returns the index of the key that is flagged, or -1 if no keys are flagged. See the method above.

**Default Implementation:**

```
{return -1;}
```

**Sample Code:**

## **INTERP\_CONT\_TEMPLATE**

```
int InterpControl<INTERP_CONT_PARAMS>::GetFlagKeyIndex()
{
    int n = keys.Count();
    for ( int i = 0; i < n; i++ ) {
        if (keys[i].TestKFlag(KEY_FLAGGED)) {
            return i;
        }
    }
    return -1;
}
```

### **Prototype:**

```
virtual int NumSelKeys();
```

### **Remarks:**

Implemented by the Plug-In.

Returns the number of selected keys.

### **Default Implementation:**

```
{return 0;}
```

### **Prototype:**

```
virtual void CloneSelectedKeys(BOOL offset=FALSE);
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to make a copy of the selected keys.

### **Parameters:**

**BOOL offset=FALSE**

If TRUE, set the new key time to be centered between the original key and the next key.

### **Prototype:**

```
virtual void AddNewKey(TimeValue t,DWORD flags);
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to add a new key at the specified time. The value of the key is set to the value of the previous key, or interpolated between keys, based on the flags passed.

**Parameters:**

**TimeValue t**

The time to add the key.

**DWORD flags**

One or more of the following values:

**ADDKEY\_SELECT**

Select the new key and deselect any other selected keys.

**ADDKEY\_INTERP**

If TRUE then initialize the new key to the interpolated value at that time. If FALSE, initialize the key to the value of the previous key.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void MoveKeys(ParamDimensionBase *dim,float  
delta,DWORD flags);
```

**Remarks:**

Implemented by the Plug-In.

This method is called to move selected keys vertically in the function curve editor. This moves the key values but does not alter the key times. The developer adds the delta to the selected key values, after converting them using the dimension **\*dim** passed. See the sample code below for how this may be done.

**Parameters:**

**ParamDimensionBase \*dim**

This class is used to scale the parameter's values into and out of units used in the user interface. For example, if the parameter was an angle, it would be shown in degrees, but stored in radians. Methods of this class allow the value to be converted back and forth. This is needed because the delta passed is in

user interface units. Thus the selected key values need to be converted before the delta is applied. See [Class ParamDimensBase](#).

### **float delta**

The amount to move the keys (move the values - not the times). This is in the units of the user interface. For example, if an angle has a value in the function curve editor of 100 degrees, 100 would be passed as the delta.

### **DWORD flags**

These are not currently used.

### **Default Implementation:**

```
{}
```

### **Sample Code:**

```
INTERP_CONT_TEMPLATE
void
InterpControl<INTERP_CONT_PARAMS>::MoveKeys(ParamDimensio
*dim,float delta,DWORD flags)
{
int n = keys.Count();
if (!n) return;
float m = 1.0f;
Interval valid;
BOOL changed = FALSE;
HoldTrack();
for (int i = 0; i < n; i++ ) {
for (int j=0;j<ELS;j++) {
if (keys[i].AnyElemSelected()) {
m = GetMultVal(keys[i].time,valid);
}
if (keys[i].ElemSelected(j)) {
keys[i][j] =
dim->UnConvert(dim->Convert(keys[i][j]*m)+delta)/m;
changed = TRUE;
}
}
}
if (changed) {
// FALSE indicates that key times didn't
```

```
// change so sorting isn't necessary.  
keys.KeysChanged(FALSE);  
ivalid.SetEmpty();  
NotifyDependents(FOREVER, PART_ALL, REFMSG_CHANGE);  
}  
}
```

**Prototype:**

```
virtual void ScaleKeyValues(ParamDimensionBase *dim,float  
origin, float scale,DWORD flags)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to scale selected keys values. This scales the key values but does not alter the key times. The developer scales the selected key values about the specified origin, after converting them using the dimension **\*dim** passed.

Note the following macro available for scaling about an origin:

```
#define ScaleAboutOrigin(val,origin,scale)  
(((val)-(origin))*(scale))+(origin))
```

**Parameters:**

**ParamDimensionBase \*dim**

This class is used to scale the parameter's values into and out of units used in the user interface. For example, if the parameter was an angle, it would be shown in degrees, but stored in radians. Methods of this class allow the value to be converted back and forth. See [Class ParamDimensBase](#).

**float origin**

The origin about which the keys are scaled.

**float scale**

The scale factor to apply to the key values.

**DWORD flags**

These are not currently used.

**Default Implementation:**

```
{}
```

**Sample Code:**

```
INTERP_CONT_TEMPLATE
void InterpControl<INTERP_CONT_PARAMS>::ScaleKeyValues(
    ParamDimensionBase *dim,float origin,float scale,DWORD flags)
{
    int n = keys.Count();
    if (!n) return;
    BOOL changed = FALSE;
    HoldTrack();
    for (int i = 0; i < n; i++ ) {
        for (int j=0;j<ELS;j++) {
            if (keys[i].ElemSelected(j)) {
                keys[i][j] = dim->UnConvert( ScaleAboutOrigin(
                    dim->Convert(keys[i][j]),origin,scale) );
                changed = TRUE;
            }
        }
    }
    if (changed) {
        keys.KeysChanged(FALSE);
        ivalid.SetEmpty();
        NotifyDependents(FOREVER, PART_ALL, REFMSG_CHANGE);
    }
}
```

**Prototype:**

```
virtual void SelectCurve(BOOL sel)
```

**Remarks:**

Implemented by the Plug-In.

The plug-in keeps track of whether its function curve is selected or not. This method is called to have the plug-in select or deselect its function curve.

**Parameters:**

**BOOL sel**

TRUE if the curve should be selected; FALSE if it should be deselected.

**Default Implementation:**

```
}
```

**Prototype:**

```
virtual BOOL IsCurveSelected()
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the function curve is selected; otherwise returns FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL IsKeySelected(int index)
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the key specified by the index is selected; otherwise FALSE.

**Parameters:**

**int index**

The index of the key to test.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual int GetSelKeyCoords(TimeValue &t, float &val, DWORD flags);
```

**Remarks:**

Implemented by the Plug-In.

This method is used to determine the commonality of the selected keys for display in the time/value type in fields of Track View. It is also used to retrieve the value and/or time of the selected keys (if there is only one selected, or they are common to the selected keys). The flags parameter specified which values to retrieve. The return value indicates if nothing, or several keys were selected. It also indicates if the selected keys shared a common time and/or common

value.

**Parameters:**

**TimeValue &t**

The time of the selected keys is returned here (if appropriate).

**float &val**

The value of the selected keys is returned here (if appropriate).

**DWORD flags**

One of the following values:

**KEYCOORDS\_TIMEONLY**

Only the time **t** needs to be updated.

**KEYCOORDS\_VALUEONLY**

Only the value **val** needs to be updated.

**Return Value:**

This indicates what was selected, and what these keys had in common. One or more of the following values should be set:

**KEYS\_NONESELECTED**

This indicates that no keys are selected.

**KEYS\_MULTISELECTED**

This indicates that multiple keys are selected.

Both of these last two bits could be set.

**KEYS\_COMMONTIME**

If the selected keys share the same time then this flag should be set. In this case it is appropriate to update **t** if required.

**KEYS\_COMMONVALUE**

If the selected keys share the same value then this flag should be set. In this case it is appropriate to update **val** if required.

**Default Implementation:**

```
{return KEYS_NONESELECTED;}
```

**Prototype:**

```
virtual void SetSelKeyCoords(TimeValue t, float val,DWORD flags)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to update the time and/or value of the selected keys as specified by the flags. This is called if the user uses the time/value type in fields of Track View.

**Parameters:****TimeValue t**

The time to set for the selected keys (if the flags indicate this is needed).

**float val**

The value to set for the selected keys (if the flags indicate this is needed).

**DWORD flags**

One of the following values:

**KEYCOORDS\_TIMEONLY**

Only the time needs to be updated.

**KEYCOORDS\_VALUEONLY**

Only the value needs to be updated.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual int SetSelKeyCoordsExpr(ParamDimension *dim, TCHAR *timeExpr, TCHAR *valExpr, DWORD flags);
```

**Remarks:**

Implemented by the Plug-In.

This method is available in release 2.0 and later only.

This method is similar to **SetSelKeyCoords()** above. In that case you're given a time and a value and are to update the selected keys with these values (based on the flags passed). In this case, you are instead passed time and value **expressions** (as strings). The idea is that these strings are evaluated as expressions and the resulting values are used to update the selected keys. For instance, the user could select a bunch of keys and then type in **n+45**. This would add 45 to all the values of the keys.

Developers can use the 3ds max expression parser (see [Class Expr](#)) to evaluate

the strings. Debug SDK users can see `\MAXSDKDB\SDKSRC\CTRLTEMP.H` for an example (or see the sample code below)). If a plug-in doesn't support this feature it can return `FALSE` from this method and the old `SetSelKeyCoords()` method will be called. Note that the variable names are defined as `KEYCOORDS_TIMEVAR` and `KEYCOORDS_VALVAR`.

#### **Parameters:**

**ParamDimension \*dim**

This is used to convert the parameter value once you get it.

**TCHAR \*timeExpr**

A string containing the time expression.

**TCHAR \*valExpr**

A string containing the value expression.

**DWORD flags**

One of the following values:

**KEYCOORDS\_TIMEONLY**

Only the time **t** needs to be updated.

**KEYCOORDS\_VALUEONLY**

Only the value **val** needs to be updated.

#### **Return Value:**

This indicates what was selected, and what these keys had in common. One or more of the following values should be set:

**KEYCOORDS\_EXPR\_UNSUPPORTED**

Don't implement this method

**KEYCOORDS\_EXPR\_ERROR**

Error in expression

**KEYCOORDS\_EXPR\_OK**

Expression evaluated

#### **Default Implementation:**

```
{return KEYCOORDS_EXPR_UNSUPPORTED;}
```

#### **Sample Code:**

## INTERP\_CONT\_TEMPLATE

```
int InterpControl<INTERP_CONT_PARAMS>::SetSelKeyCoordsExpr(
    ParamDimension *dim,
    TCHAR *timeExpr, TCHAR *valExpr, DWORD flags)
{
    Expr texpr, vexpr;
    float vin, vout=0.0f, tfin, tfout=0.0f;

    if (timeExpr) {
        texpr.defVar(SCALAR_VAR,KEYCOORDS_TIMEVAR);
        if (texpr.load(timeExpr)!=EXPR_NORMAL) return
KEYCOORDS_EXPR_ERROR;
    }
    if (valExpr) {
        vexpr.defVar(SCALAR_VAR,KEYCOORDS_VALVAR);
        if (vexpr.load(valExpr)!=EXPR_NORMAL) return
KEYCOORDS_EXPR_ERROR;
    }

    int n = keys.Count();
    if (!n) return KEYCOORDS_EXPR_OK;
    HoldTrack();
    for (int i = 0; i < n; i++ ) {
        if (!(flags&KEYCOORDS_VALUEONLY)) {
            if (keys[i].TimeLocked()) continue;
            if (keys[i].TestKFlag(KEY_SELECTED)) {
                tfin = float(keys[i].time)/float(GetTicksPerFrame());
                texpr.eval(&tfout, 1, &tfin);
                keys[i].time = int(tfout*GetTicksPerFrame());
            }
        }
        if (!(flags&KEYCOORDS_TIMEONLY)) {
            for (int j=0;j<ELS;j++) {
                if (keys[i].ElemSelected(j)) {
                    vin = dim->Convert(keys[i][j]);
                    vexpr.eval(&vout, 1, &vin);
                    keys[i][j] = dim->UnConvert(vout);
                }
            }
        }
    }
}
```

```

    }
    }
    }
    }
    keys.KeysChanged();
    keys.CheckForDups();
    ivalid.SetEmpty();
    NotifyDependents(FOREVER, PART_ALL, REFMSG_CHANGE);
    return KEYCOORDS_EXPR_OK;

```

### Prototype:

```

virtual void AdjustTangents(TrackHitRecord hit, ParamDimensionBase *dim, Rect& rcGraph, float tzoom, int tscroll, float vzoom, int vscroll, int dx, int dy, DWORD flags)

```

### Remarks:

Implemented by the Plug-In.

If a plug-in has tangent handles, this method is called if the user adjusts them. If a plug-in doesn't have tangent handles, this method may be ignored. This method is called if the user selects one of the handles and moves the mouse. This method is passed the dx, and dy of the mouse motion.

The plug-in may have any types of handles it wishes, and it is responsible for processing whatever needs to be done when the user adjusts them. The method is passed information about the screen space, such as the overall rectangle, and time and value scroll and zoom factors. See [List of Screen-Time-Value Macros](#) for macros to convert in and out of screen space.

### Parameters:

#### **TrackHitRecord hit**

This identifies the handle that was selected (hit).

#### **ParamDimensionBase \*dim**

The parameter dimension. See [Class ParamDimensionBase](#).

#### **Rect& rcGraph**

This is the rectangle of the graph viewport.

#### **float tzoom**

This is the time zoom factor.

**int tscroll**

This is the time scroll factor.

**float vzoom**

This is the value zoom factor.

**int vscroll**

This is the value scroll factor.

**int dx**

The mouse movement in screen coordinates in the x direction.

**int dy**

The mouse movement in screen coordinates in the y direction.

**DWORD flags**

One of the following values:

**ADJTAN\_LOCK**

Indicates the tangents are locked.

**ADJTAN\_BREAK**

Indicates the tangents have been broken.

**Default Implementation:**

```
{ }
```

**Drawing and hit testing tracks****Prototype:**

```
virtual int GetTrackVSpace(int lineHeight)
```

**Remarks:**

Implemented by the Plug-In.

Returns the vertical space occupied by the track in units of one line.

**Parameters:****int lineHeight**

The height of a single line in pixels.

**Default Implementation:**

```
{ return 1; }
```

**Prototype:**

**virtual int HitTestTrack(TrackHitTab& hits,Rect& rcHit,Rect& rcTrack,float zoom,int scroll,DWORD flags)**

**Remarks:**

Implemented by the Plug-In.

This method is called to determine which keys lie within the **rcHit** rectangle. Keys that are hit are added to the **hits** table.

**Parameters:****TrackHitTab& hits**

The table of **TrackHitRecords** to update. Each key that lies within the hit rectangle (is hit) should be added to this table. It is up to the plug-in to define a scheme that allows it to identify its hits using the data members of [Class TrackHitRecord](#). Also see [Class Tab](#) for methods to add to the table.

**Rect& rcHit**

This is the region that was selected for hit testing. This may be a small rectangle about the mouse pick point, or a larger rectangle if the user selected by window.

**Rect& rcTrack**

This is the entire rectangular region of the track.

**float zoom**

The is the time zoom factor.

**int scroll**

This is the time scroll factor.

**DWORD flags**

One or more of the following value:

**HITTRACK\_SELONLY**

Selected only.

**HITTRACK\_UNSELONLY**

Unselected only.

**HITTRACK\_ABORTONHIT**

Abort hit testing on first hit.

**HITCURVE\_TESTTANGENTS**

Hit test curve tangents.

**Return Value:**

One of the following values:

**TRACK\_DONE**

This indicates the track was hit tested.

**TRACK\_DORANGE**

This indicates that the system will handle hit testing to the range bar for the item. For example a node returns this value because it does not have any keys. Therefore it just lets the user hit test the range bar. In general, anything that is not a leaf controller will not implement this method and return the default. The system will then simply hit test the range bar.

**TRACK\_ASKCLIENT**

If a plug-in returns this value then the anim's client will be given a chance to paint the track in Track View. If a client returns this value then the method **PaintSubTrack()** will be called.

**Default Implementation:**

```
{ return TRACK_DORANGE; }
```

**Prototype:**

```
virtual int PaintTrack(ParamDimensionBase *dim, HDC  
hdc,Rect& rcTrack,Rect& rcPaint, float zoom, int scroll, DWORD  
flags)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to display the item in the track view. If an item needs to draw itself in a special fashion, it implements this method to do so. For example, a sound plug-in may draw its waveform using this method. If an item does not need to draw itself, the default implementation may be used. This draws the range bar for the item.

Note: When drawing something to appear in Track View, a developer should not do any clipping of their own. 3ds max will take care of all clipping itself.

**Parameters:**

**ParamDimensionBase \*dim**

This parameter is available in release 2.0 and later only.

The dimension for the parameter of this track.

**HDC hdc**

The handle of the [= 4\) BSPSPopupOnMouseOver\(event\);">device context](#).

**Rect& rcTrack**

The entire rectangle of the inside of the track.

**Rect& rcPaint**

This is the rectangular region that needs to be repainted - the invalid region.

**float zoom**

The time zoom factor.

**int scroll**

The time scroll factor.

**DWORD flags**

One or more of the following values. These are filters for controllers with more than one curve. NOTE: RGB controllers interpret X as red, Y as green, and Z as blue.

**DISPLAY\_XCURVE**

**DISPLAY\_YCURVE**

**DISPLAY\_ZCURVE**

**Return Value:**

One of the following values:

**TRACK\_DONE**

Indicates the track was painted.

**TRACK\_DORANGE**

Indicates the system should draw the range bars for the item.

**TRACK\_ASKCLIENT**

Indicates the anim's client will be given a chance to paint the track in Track View. See **Animatable::PaintSubTrack()** which will be called to do this.

**Default Implementation:**

```
{ return TRACK_DORANGE; }
```

**Prototype:**

```
virtual int PaintSubTrack(int subNum, ParamDimensionBase
```

**\*dim, HDC hdc, Rect& rcTrack, Rect& rcPaint, float zoom, int scroll, DWORD flags);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

This method will be called if PaintTrack returns **TRACK\_ASKCLIENT**.

This gives the anim's client a chance to paint the tracks in Track View.

**Parameters:**

**int subNum**

Specifies the sub-anim to paint.

**ParamDimensionBase \*dim**

The dimension for the parameter of this track.

**HDC hdc**

The handle of the [= 4\) BSPSPopupOnMouseOver\(event\);">device context](#).

**Rect& rcTrack**

The entire rectangle of the inside of the track.

**Rect& rcPaint**

This is the rectangular region that needs to be repainted - the invalid region.

**float zoom**

The time zoom factor.

**int scroll**

The time scroll factor.

**DWORD flags**

One or more of the following values. These are filters for controllers with more than one curve. NOTE: RGB controllers interpret X as red, Y as green, and Z as blue.

**DISPLAY\_XCURVE**

**DISPLAY\_YCURVE**

**DISPLAY\_ZCURVE**

**Return Value:**

One of the following values:

**TRACK\_DONE**

Indicates the track was painted.

## **TRACK\_DORANGE**

Indicates the system should draw the range bars for the item.

### **Default Implementation:**

```
{return TRACK_DORANGE;}
```

## **Drawing and hit testing function curves**

### **Prototype:**

```
virtual int PaintFCurves(ParamDimensionBase *dim, HDC hdc,  
Rect& rcGraph,Rect& rcPaint, float tzoom, int tscroll, float  
vzoom, int vscroll, DWORD flags);
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to draw the function curve of the anim.

### **Parameters:**

#### **ParamDimensionBase \*dim**

The parameter dimension. See [Class ParamDimensionBase](#).

#### **HDC hdc**

The handle of the [= 4\) BSPSPopupOnMouseOver\(event\);">device context](#).

#### **Rect& rcGraph**

The entire rectangle of the inside of the graph region.

#### **Rect& rcPaint**

This is the rectangular region that needs to be repainted - the invalid region.

#### **float tzoom**

The time zoom factor.

#### **int tscroll**

The time scroll factor.

#### **float vzoom**

The value zoom factor.

#### **int vscroll**

The value scroll factor.

#### **DWORD flags**

One or more of the following values:

## **PAINTCURVE\_SHOWTANGENTS**

Show the curve tangent handles.

## **PAINTCURVE\_FROZEN**

Show the curve in a frozen state.

These are filters for controllers with more than one curve. NOTE: RGB controllers interpret X as red, Y as green and Z as blue.

### **DISPLAY\_XCURVE**

### **DISPLAY\_YCURVE**

### **DISPLAY\_ZCURVE**

## **PAINTCURVE\_GENCOLOR**

This option is available in 3ds max 2.0 and later only.

Draw the curve in its standard color.

The following options are passed to float controllers indicating a suggested color for drawing.

### **PAINTCURVE\_XCOLOR**

This option is available in 3ds max 2.0 and later only.

Draw the curve in red.

### **PAINTCURVE\_YCOLOR**

This option is available in 3ds max 2.0 and later only.

Draw the curve in green.

### **PAINTCURVE\_ZCOLOR**

This option is available in 3ds max 2.0 and later only.

Draw the curve in blue.

### **Return Value:**

A plug-in should always return 0.

### **Default Implementation:**

```
{ return 0; }
```

### **Prototype:**

```
virtual int HitTestFCurves(ParamDimensionBase  
*dim,TrackHitTab& hits, Rect& rcHit, Rect& rcGraph,float  
tzoom, int tscroll,float vzoom,int vscroll, DWORD flags)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to hit test the item's function curves. It is called to determine which keys on the curve lie within the **rcHit** rectangle. Keys that are hit are added to the **hits** table.

**Parameters:****ParamDimensionBase \*dim**

The parameter dimension. See [Class ParamDimensionBase](#).

**TrackHitTab& hits**

The table of **TrackHitRecords** to update. Each key that lies within the hit rectangle (is hit) should be added to this table. It is up to the plug-in to define a scheme that allows it to identify its hits using the data members of [Class TrackHitRecord](#). Also see [Class Tab](#) for methods to add to the table.

**Rect& rcHit**

This is the region that was selected for hit testing. This may be a small rectangle about the mouse pick point, or a larger rectangle if the user selected by window.

**Rect& rcGraph**

This is the entire rectangle of the graph region.

**float tzoom**

This is the time zoom factor.

**int tscroll**

This is the time scroll factor.

**float vzoom**

This is the time zoom factor.

**int vscroll**

This is the time scroll factor.

**DWORD flags**

One or more of the following values:

**HITTRACK\_SELONLY**

Selected only.

**HITTRACK\_UNSELONLY**

Unselected only.

## **HITTRACK\_ABORTONHIT**

Abort hit testing on first hit.

## **HITCURVE\_TESTTANGENTS**

Hit Test curve tangent handles.

These are filters for controllers with more than one curve. NOTE: RGB controllers interpret X as red, Y as green and Z as blue.

### **DISPLAY\_XCURVE**

### **DISPLAY\_YCURVE**

### **DISPLAY\_ZCURVE**

#### **Return Value:**

One of the following values to indicate what was hit:

### **HITCURVE\_KEY**

Hit one or more keys.

### **HITCURVE\_WHOLE**

Hit the curve (anywhere).

### **HITCURVE\_TANGENT**

Hit a tangent handle.

### **HITCURVE\_NONE**

Nothing was hit.

### **HITCURVE\_ASKCLIENT**

Ask the client to hit test the function curve. See **HitTestSubFCurve()** below.

#### **Default Implementation:**

```
{ return HITCURVE_NONE; }
```

#### **Prototype:**

```
virtual int PaintSubFCurves(int subNum, ParamDimensionBase  
*dim, HDC hdc, Rect& rcGraph, Rect& rcPaint, float tzoom, int  
tscroll, float vzoom, int vscroll, DWORD flags);
```

#### **Remarks:**

Implemented by the Plug-In.

This method is called to draw the specified sub-anim function curve. This allows the client to paint its function curve.

## Parameters:

### **int subNum**

The sub-anim number to paint.

### **ParamDimensionBase \*dim**

The parameter dimension. See [Class ParamDimensionBase](#).

### **HDC hdc**

The handle of the [= 4\) BSPSPopupOnMouseOver\(event\);">device context](#).

### **Rect& rcGraph**

The entire rectangle of the inside of the graph region.

### **Rect& rcPaint**

This is the rectangular region that needs to be repainted - the invalid region.

### **float tzoom**

The time zoom factor.

### **int tscroll**

The time scroll factor.

### **float vzoom**

The value zoom factor.

### **int vscroll**

The value scroll factor.

### **DWORD flags**

One or more of the following values:

#### **PAINTCURVE\_SHOWTANGENTS**

Show the curve tangent handles.

#### **PAINTCURVE\_FROZEN**

Show the curve in a frozen state.

These are filters for controllers with more than one curve. NOTE: RGB controllers interpret X as red, Y as green and Z as blue.

#### **DISPLAY\_XCURVE**

#### **DISPLAY\_YCURVE**

#### **DISPLAY\_ZCURVE**

#### **PAINTCURVE\_GENCOLOR**

This option is available in 3ds max 2.0 and later only.

Draw the curve in its standard color.

The following options are passed to float controllers indicating a suggested color for drawing.

### **PAINTCURVE\_XCOLOR**

This option is available in 3ds max 2.0 and later only.

Draw the curve in red.

### **PAINTCURVE\_YCOLOR**

This option is available in 3ds max 2.0 and later only.

Draw the curve in green.

### **PAINTCURVE\_ZCOLOR**

This option is available in 3ds max 2.0 and later only.

Draw the curve in blue.

### **Return Value:**

A plug-in should always return 0.

### **Default Implementation:**

```
{ return 0; }
```

### **Prototype:**

```
virtual int HitTestSubFCurves(int subNum, ParamDimensionBase *dim, TrackHitTab& hits, Rect& rcHit, Rect& rcGraph, float tzoom, int tscroll, float vzoom, int vscroll, DWORD flags);
```

### **Remarks:**

Implemented by the Plug-In.

This method is called if **HitTestFCurves()** returns **HITCURVE\_ASKCLIENT**. It allows the client to hit test its sub-anim curves.

### **Parameters:**

**int subNum**

The sub-anim number to hit test.

**ParamDimensionBase \*dim**

The parameter dimension. See [Class ParamDimensionBase](#).

**TrackHitTab& hits**

The table of **TrackHitRecords** to update. Each key that lies within the hit rectangle (is hit) should be added to this table. It is up to the plug-in to define

a scheme that allows it to identify its hits using the data members of [Class TrackHitRecord](#). Also see [Class Tab](#) for methods to add to the table.

### **Rect& rcHit**

This is the region that was selected for hit testing. This may be a small rectangle about the mouse pick point, or a larger rectangle if the user selected by window.

### **Rect& rcGraph**

This is the entire rectangle of the graph region.

### **float tzoom**

This is the time zoom factor.

### **int tscroll**

This is the time scroll factor.

### **float vzoom**

This is the time zoom factor.

### **int vscroll**

This is the time scroll factor.

### **DWORD flags**

One or more of the following values:

#### **HITTRACK\_SELONLY**

Selected only.

#### **HITTRACK\_UNSELONLY**

Unselected only.

#### **HITTRACK\_ABORTONHIT**

Abort hit testing on first hit.

#### **HITCURVE\_TESTTANGENTS**

Hit Test curve tangent handles.

These are filters for controllers with more than one curve. NOTE: RGB controllers interpret X as red, Y as green and Z as blue.

#### **DISPLAY\_XCURVE**

#### **DISPLAY\_YCURVE**

#### **DISPLAY\_ZCURVE**

### **Return Value:**

One of the following values to indicate what was hit:

## **HITCURVE\_KEY**

Hit one or more keys.

## **HITCURVE\_WHOLE**

Hit the curve (anywhere).

## **HITCURVE\_TANGENT**

Hit a tangent handle.

## **HITCURVE\_NONE**

Nothing was hit.

### **Default Implementation:**

```
{ return HITCURVE_NONE; }
```

### **Prototype:**

```
virtual void EditTrackParams(TimeValue t, ParamDimensionBase  
*dim, TCHAR *pname, HWND hParent, IObjParam *ip,  
DWORD flags)
```

### **Remarks:**

Implemented by the Plug-In.

This method is called for the plug-in to put up a modal dialog and let the user edit the tracks parameters for the selected keys. This function should not return until the user has completed editing at which time any windows that were created should be destroyed. Unlike **BeginEditParams()** and **EndEditParams()** this interface is modal.

### **Parameters:**

#### **TimeValue t**

This time represents the horizontal position of where the user right clicked to bring up the modal edit track parameters dialog. See the flags below for when this parameter is valid.

#### **ParamDimensionBase \*dim**

The parameter dimension. See [Class ParamDimensionBase](#).

#### **TCHAR \*pname**

The name of the parameter as given by the client.

#### **HWND hParent**

This is the parent window that should be used to create any dialogs.

## **IObjParam \*ip**

An interface pointer available for calling functions in 3ds max.

## **DWORD flags**

One or more of the following values:

### **EDITTRACK\_FCURVE**

The user is in the function curve editor.

### **EDITTRACK\_TRACK**

The user is in one of the track views.

### **EDITTRACK\_SCENE**

The user is editing a path in the scene.

### **EDITTRACK\_BUTTON**

The user invoked by choosing the properties button. In this case the time parameter is NOT valid.

### **EDITTRACK\_MOUSE**

The user invoked by right clicking with the mouse. In this case the time parameter is valid.

## **Default Implementation:**

```
{}
```

## **Prototype:**

```
virtual int TrackParamsType()
```

## **Remarks:**

Implemented by the Plug-In.

This method returns a value that indicates how the track parameter editing is invoked.

## **Return Value:**

One of the following values:

### **TRACKPARAMS\_NONE**

Has no track parameters. If this is returned then **EditTrackParams()** will not be called.

### **TRACKPARAMS\_KEY**

Entered by right clicking on a selected key. This should be used if the dialog provides parameters for the entire controller (for example as the

Noise controller's dialog does).

### **TRACKPARAMS\_WHOLE**

Entered by right clicking anywhere in the track. This should be used if the dialog will represent the selection of keys (as a key info type dialog does).

#### **Default Implementation:**

```
{return TRACKPARAMS_NONE;}
```

#### **Prototype:**

```
virtual int GetFCurveExtents(ParamDimensionBase *dim,float  
&min, float &max, DWORD flags)
```

#### **Remarks:**

Implemented by the Plug-In.

This method is called to calculate the largest and smallest values of the anim.

#### **Parameters:**

##### **ParamDimensionBase \*dim**

The dimension of the anim. See [Class ParamDimensionBase](#).

##### **float &min**

The smallest value. These are in the units given by the dimension. For example, if it was an angle parameter that was displayed in degrees, the units returned through **min** should be in degrees as well.

##### **float &max**

The largest value. These are in the units given by the dimension. For example, if it was an angle parameter that was displayed in degrees, the units returned through **max** should be in degrees as well.

##### **DWORD flags**

These are filters for controllers with more than one curve. NOTE: RGB controllers interpret X as red, Y as green and Z as blue. One or more of the following values:

**DISPLAY\_XCURVE**

**DISPLAY\_YCURVE**

**DISPLAY\_ZCURVE**

#### **Return Value:**

If this method is processed, return nonzero; otherwise zero.

**Default Implementation:**

```
{return 0;}
```

**Sub-Class Indication****Prototype:**

```
virtual BOOL IsSubClassOf(Class_ID classID)
```

**Remarks:**

Implemented by the Plug-In.

If an object is a sub-class of a particular class, it will have a different **ClassID()** because it is a different class. This method allows an object to indicate that it is a sub-class of a particular class and therefore can be treated as one. For example, a class could be derived from **TriObject**. This derived class would have a different **ClassID()** than the **TriObject's** class ID however it still can be treated (cast) as a **TriObject** because it is derived from **TriObject**. Note the default implementation: a class is considered to also be a subclass of itself.

**Parameters:**

**Class\_ID classID**

The Class\_ID of the item that this class is a sub-class of.

**Return Value:**

TRUE if this class is a sub-class of **classID**; otherwise FALSE.

**Default Implementation:**

```
{return classID==ClassID();}
```

**Enumeration of Anims and Auxiliary Files****Prototype:**

```
int EnumAnimTree(AnimEnum *animEnum, Animatable *client,  
int subNum);
```

**Remarks:**

Implemented by the System.

This method recursively enumerates the Animatable hierarchy. It will call the

**AnimEnum::proc()** method passing it the anim, that anim's parent (the client), and the sub-anim index of that anim to the client, for every anim and sub-anim in the hierarchy.

**Parameters:**

**AnimEnum \*animEnum**

The callback object, called once for each sub anim from 0 to subNum-1. See [Class AnimEnum](#).

**Animatable \*client**

The client anim. This is the Animatable whose sub-anim are enumerated.

**int subNum**

The sub-anim index of the client at which to begin the enumeration. Pass 0 to do them all.

**Return Value:**

One of the following values:

**ANIM\_ENUM\_PROCEED**

**ANIM\_ENUM\_STOP**

**ANIM\_ENUM\_ABORT**

**Prototype:**

**virtual void EnumAuxFiles(NameEnumCallback& nameEnum, DWORD flags)**

**Remarks:**

Implemented by the Plug-In.

This method is used to enumerate any 'auxiliary' files maintained by the item and record the filename with the callback. Entities which actually need to load auxiliary files (for instance texture maps) must implement this method, possibly calling **ReferenceMaker::EnumAuxFiles()** also. The **ReferenceMaker** implementation simply calls itself on all its references (see below).

Class **Interface** has a method that may be used to call this on the entire system. This includes the renderer, materials editor, atmospheric effects, background, video post, lights, etc. -- everything that may have auxiliary files associated with it. After specifying the appropriate flags a list of filenames is created that matched the enumeration conditions as specified by the flags. This

is used for instance by the Archive function in 3ds max to grab a list of bitmap files used by the system.

**Parameters:**

**NameEnumCallback& nameEnum**

The callback object that may be used to record the name. See [Class NameEnumCallback](#).

**DWORD flags**

One or more of the following values. See [List of EnumAuxFiles Flags](#)

**Sample Code:**

This is the default implementation provided by **ReferenceMaker**.

```
void ReferenceMaker::EnumAuxFiles(NameEnumCallback&
nameEnum, DWORD flags) {
    for (int i=0; i<NumRefs(); i++) {
        ReferenceMaker *srm = GetReference(i);
        if (srm) srm->EnumAuxFiles(nameEnum,flags);
    }
}
```

**Prototype:**

**virtual int NumChildren()**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual Animatable\* ChildAnim(int i)**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual TSTR NodeName();**

**Remarks:**

This method is used internally.

## The following methods deal with Note Tracks.

### Prototype:

```
void AddNoteTrack(NoteTrack *note);
```

### Remarks:

Implemented by the System.

This method adds the specified note track.

### Parameters:

**NoteTrack \*note**

The note track to add. The Note Tracks provided by 3ds max are derived from [Class DefNoteTrack](#) (which is derived from **NoteTrack**). See that class for the methods and data members used to access Note Tracks.

### Prototype:

```
void DeleteNoteTrack(NoteTrack *note);
```

### Remarks:

Implemented by the System.

This method deletes the specified note track.

### Parameters:

**NoteTrack \*note**

The note track to delete. The Note Tracks provided by 3ds max are derived from [Class DefNoteTrack](#) (which is derived from **NoteTrack**). See that class for the methods and data members used to access Note Tracks.

### Prototype:

```
BOOL HasNoteTracks();
```

### Remarks:

Implemented by the System.

This method returns TRUE if the track has note tracks; otherwise FALSE.

### Prototype:

```
int NumNoteTracks();
```

**Remarks:**

Implemented by the System.

This method returns the number of note tracks.

**Prototype:**

**NoteTrack \*GetNoteTrack(int i);**

**Remarks:**

Implemented by the System.

This method retrieves the 'i-th' note track.

**Parameters:**

**int i**

Specifies the note track to retrieve.

**Return Value:**

A pointer to a Note Track. The Note Tracks provided by 3ds max are derived from [Class DefNoteTrack](#) (which is derived from **NoteTrack**). See that class for the methods and data members used to access Note Tracks.

**Parameter Block2 Methods****Prototype:**

**virtual int NumParamBlocks();**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method returns the number of ParamBlock2s in this instance.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

**virtual IParamBlock2\* GetParamBlock(int i);**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method return 'i-th' ParamBlock2 in this instance (or NULL if not available).

**Parameters:**

**int i**

The zero based index of the ParamBlock2 to return.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual IParamBlock2* GetParamBlockByID(short id);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method returns a pointer to the ParamBlock2 as specified by the ID passed (or NULL if not available).

**Parameters:**

**short id**

The BlockID of the ParamBlock2 instance.

**Default Implementation:**

```
{ return NULL; }
```

## Schematic View Methods

**Prototype:**

```
bool SvSaveData(ISave *isave, USHORT id);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

This is the save method for schematic view data. For classes derived from ReferenceMaker, there is no need to call these. However, **if** you have a class derived from Animatable **and** it appears in the schematic view **and** you want

to save schematic view properties for the object (node position, selection state, etc.) **then** you have to call this in your **Save()** method.

**Parameters:**

**ISave \*isave**

An interface for saving data. See [Class ISave](#).

**USHORT id**

The Chunk id (chosen by the developer).

**Return Value:**

Returns true if saved okay; otherwise false.

**Prototype:**

**bool SvLoadData(ILoad \*iLoad);**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

This is the load method for schematic view data. For classes derived from ReferenceMaker, there is no need to call these. However, **if** you have a class derived from Animatable **and** it appears in the schematic view **and** you want to load schematic view properties for the object (node position, selection state, etc.) **then** you have to call this in your **Load()** method.

**Parameters:**

**ILoad \*iLoad**

An interface for loading data. See [Class ILoad](#).

**Return Value:**

Returns true if loaded okay; otherwise false.

**Prototype:**

**virtual SvGraphNodeReference**

**SvTraverseAnimGraph(IGraphObjectManager \*gom, Animatable \*owner, int id, DWORD flags);**

**Remarks:**

This method is available in release 3.0 and later only.

This method traverses the graph of objects in the 3ds max scene, adding desired objects to the schematic view. Developers can specialize this behaviour by overriding this method and adding whatever objects are interesting to the schematic view. Objects are added to the schematic view by calling **IGraphObjectManager::AddAnimatable(...)**. Reference lines are added to the schematic view by calling **IGraphObjectManager::AddReference(...)**. Implementers of this method should call it recursively to process other objects in the scene. See [Class IGraphObjectManager](#).

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**Animatable \*owner**

The owning animatable.

**int id**

This is usually the sub-anim number (but can actually be any value the developer chooses).

**DWORD flags**

See [List of Schematic View AddAnimatable Flags](#).

**Return Value:**

A SvGraphNodeReference object.

**Prototype:**

**SvGraphNodeReference**

**SvStdTraverseAnimGraph(IGraphObjectManager \*gom,  
Animatable \*owner, int id);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is a default graph traversal function which can be called from **SvTraverseAnimGraph(...)** above to handle graph traversal in simple cases. This traversal follows the sub-anim and child references. See the code below.

**Parameters:**

## **IGraphObjectManager \*gom**

Points to the schematic view window manager.

## **Animatable \*owner**

The owning animatable.

## **int id**

This is usually the sub-anim number (but can actually be any value the developer chooses).

## **DWORD flags**

See [List of Schematic View AddAnimatable Flags](#).

### **Return Value:**

A SvGraphNodeReference object.

### **Default Implementation:**

```
// A default graph traversal function which can be
// called from SvTraverseAnimGraph(...) to handle
// graph traversal in simple cases. Follows sub-anim
// and child references...
```

### **SvGraphNodeReference**

```
Animatable::SvStdTraverseAnimGraph(IGraphObjectManager
```

```
*gom, Animatable *owner, int id, DWORD flags) {
```

```
int i;
```

```
SvGraphNodeReference nodeRef;
```

```
SvGraphNodeReference childNodeRef;
```

```
gom->PushLevel(this);
```

```
nodeRef = gom->AddAnimatable(this, owner, id, flags);
```

```
if (nodeRef.stat == SVT_PROCEED) {
```

```
for (i = 0; i < NumSubs(); i++) {
```

```
if (SubAnim(i)) {
```

```
childNodeRef = SubAnim(i)->SvTraverseAnimGraph(gom, this, i,
flags);
```

```
if (childNodeRef.stat != SVT_DO_NOT_PROCEED)
```

```
gom->AddReference(nodeRef.gNode, childNodeRef.gNode,
REFTYPE_SUBANIM);
```

```
}
```

```
}
```

```
}  
gom->PopLevel();  
return nodeRef;  
}
```

**Prototype:**

```
virtual bool SvCanInitiateLink(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns true if this animatable can be the initiator of a link operation in the schematic view.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to this node in the schematic view.

**Prototype:**

```
virtual bool SvCanConcludeLink(IGraphObjectManager *gom,  
IGraphNode *gNode, IGraphNode *initiatingGNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns true if this animatable can be the receiver (parent) of a link operation in the schematic view.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to this node in the schematic view.

**IGraphNode \*initiatingGNode**

Points to the child node in the schematic view.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual TSTR SvGetName(IGraphObjectManager *gom,  
IGraphNode *gNode, bool isBeingEdited);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the name of the object as it appears in the schematic view.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to this node in the schematic view.

**bool isBeingEdited**

true if the item is being edited; false if not.

**Default Implementation:**

```
{  
Animatable *owner;  
int subNum;  
TSTR name;  
owner = gNode->GetOwner();  
subNum = gNode->GetID();  
name = owner->SubAnimName(subNum);  
return name;  
}
```

**Prototype:**

```
virtual bool SvCanSetName(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

Return true to permit the object's name to be edited in the schematic view.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to this node in the schematic view.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool SvSetName(IGraphObjectManager *gom,  
IGraphNode *gNode, TSTR &name);
```

**Remarks:**

This method is available in release 3.0 and later only.

Called when the user changes the name of the object in the schematic view.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to this node in the schematic view.

**TSTR &name**

The new name to set.

**Return Value:**

true if the name was changed; false if not.

**Prototype:**

```
virtual bool SvCanRemoveThis(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

Return true if this object can be removed in the schematic view; false if not.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to this node in the schematic view.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool SvRemoveThis(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

This is called when the user deletes this object in the schematic view.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to this node in the schematic view.

**Return Value:**

true if deleted; false if not.

**Prototype:**

```
virtual bool SvIsHighlighted(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns true if the object is to be highlighted in the schematic view; otherwise false.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

## **IGraphNode \*gNode**

Points to this node in the schematic view.

### **Default Implementation:**

```
{ return false; }
```

### **Prototype:**

```
virtual COLORREF SvHighlightColor(IGraphObjectManager  
*gom, IGraphNode *gNode);
```

### **Remarks:**

This method is available in release 3.0 and later only.

Returns the highlight color for this node. The highlight color is used to outline nodes in the schematic view when **SvIsHighlighted(...)** returns true.

### **Parameters:**

#### **IGraphObjectManager \*gom**

Points to the schematic view window manager.

#### **IGraphNode \*gNode**

Points to this node in the schematic view.

### **Return Value:**

See [COLORREF-DWORD format](#).

### **Default Implementation:**

```
{ return gom-  
>SvGetUIColor(SV_UICLR_PLUGIN_HIGHLIGHT); }
```

### **Prototype:**

```
virtual COLORREF SvGetSwatchColor(IGraphObjectManager  
*gom, IGraphNode *gNode);
```

### **Remarks:**

This method is available in release 3.0 and later only.

Returns a color which is used to paint the triangular color swatch that appears in the upper-right hand corner of the node in the schematic view. One can return **SV\_NO\_SWATCH** to indicate that no swatch is to be drawn.

### **Parameters:**

### **IGraphObjectManager \*gom**

Points to the schematic view window manager.

### **IGraphNode \*gNode**

Points to this node in the schematic view.

#### **Return Value:**

See [COLORREF-DWORD format](#).

#### **Default Implementation:**

```
{ return SV_NO_SWATCH; }
```

#### **Prototype:**

```
virtual bool SvIsInactive(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

#### **Remarks:**

This method is available in release 3.0 and later only.

Returns true if this object is inactive; false is active. The schematic view draws inactive nodes in a grayed-out state.

#### **Parameters:**

### **IGraphObjectManager \*gom**

Points to the schematic view window manager.

### **IGraphNode \*gNode**

Points to this node in the schematic view.

#### **Default Implementation:**

```
{ return false; }
```

#### **Prototype:**

```
virtual bool SvLinkChild(IGraphObjectManager *gom,  
IGraphNode *gNodeThis, IGraphNode *gNodeChild);
```

#### **Remarks:**

This method is available in release 3.0 and later only.

This method is called to link this object to the **gNodeChild** passed.

#### **Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNodeThis**

Points to this node in the schematic view.

**IGraphNode \*gNodeChild**

Points to the child node in the schematic view.

**Return Value:**

true if linked; false if not.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool SvHandleDoubleClick(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is called when this node is double-clicked in the schematic view.

**Parameters:****IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to the node in the schematic view.

**Return Value:**

true is handled; false if not interested in the event.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual MultiSelectCallback*  
SvGetMultiSelectCallback(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is called before a multiple select/deselect operation in the schematic view. Returns a callback used to perform the (de)selection. May return NULL if this object cannot be selected in some principle editor outside the schematic view.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to the node in the schematic view.

**Return Value:**

A pointer to the callback object. See [Class MultiSelectCallback](#).

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual bool SvCanSelect(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

Returns true if this object can be selected in some editor (viewport, material editor, plug-in specific editor, etc.). Selection is actually accomplished by via the **SvGetMultiSelectCallback(...)** mechanism described above.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to the node in the schematic view.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool SvEditProperties(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is reserved for future use.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual TSTR SvGetTip(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a string to be displayed in the tip window for this object in the schematic view.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to the node in the schematic view.

**Default Implementation:**

```
{ return SvGetName(gom, gNode, false); }
```

**Prototype:**

```
virtual TSTR SvGetRefTip(IGraphObjectManager *gom,  
IGraphNode *gNode, IGraphNode *gNodeMaker);
```

**Remarks:**

Returns a string to be displayed in the tip window in the schematic view for a reference from "gNodeMaker" to this.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to the node in the schematic view.

## **IGraphNode \*gNodeMaker**

Points to the 'maker' node in the schematic view.

### **Sample Code:**

```
{ return gNodeMaker->GetAnim()->SvGetName(gom,  
gNodeMaker, false) + " -> " + SvGetName(gom, gNode, false); }
```

### **Prototype:**

```
virtual bool SvCanDetach(IGraphObjectManager *gom,  
IGraphNode *gNode);
```

### **Remarks:**

This method is available in release 3.0 and later only.

Returns true if this object can respond to the **SvDetach(...)** method; false if not.

### **Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to the node in the schematic view.

### **Default Implementation:**

```
{ return false; }
```

### **Prototype:**

```
virtual bool SvDetach(IGraphObjectManager *gom, IGraphNode  
*gNode);
```

### **Remarks:**

This method is available in release 3.0 and later only.

This method is called to detach this object from its owner.

### **Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to the node in the schematic view.

**Return Value:**

Returns true if detached; otherwise false.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
DWORD SvGetRefIndex();
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is for internal use only.

**Prototype:**

```
void SvSetRefIndex(DWORD i);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is for internal use only.

**Prototype:**

```
bool SvDeleteRefIndex();
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is for internal use only.

**Operators:****Prototype:**

```
Animatable& operator=(const Animatable& an)
```

**Remarks:**

Implemented by the System.

Assignment operator.

**The following function is not a part of class Animatable but is available for use with note tracks:**

**Prototype:**

**NoteTrack \*NewDefaultNoteTrack();**

**Remarks:**

Implemented by the System.

This function is used internally. It returns a new default note track.

## **Custom Attributes**

**Prototype:**

**ICustAttribContainer \*GetCustAttribContainer();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns a pointer to the custom attributes container interface class.

See [Class ICustAttribContainer](#) for more information.

**Prototype:**

**void AllocCustAttribContainer();**

**Remarks:**

This method is available in release 4.0 and later only.

This method allocates space for a custom attributes container.

**Prototype:**

**void DeleteCustAttribContainer();**

**Remarks:**

This method deletes space used by a custom attributes container.

# Animatable

See Also: [Class Animatable](#).

This is the base class for almost all classes related to animation. Most of the track view-related methods are implemented here. Plug-ins implement virtual methods of Animatable to handle things like the editing of an item's parameters in the command panel, and the access to an item's animatable parameters.

## **NumSubs(), SubAnim(i), SubAnimName(i)**

These methods access the plug-ins animatable parameters.

## **BeginEditParams(), EndEditParams()**

These methods are called by the system when the plug-ins parameters are to be edited in the command panel.

## **SuperClassID(), ClassID()**

These methods identify the type of plug-in and its unique class ID.

## **DeleteThis()**

This method is used to delete an instance of a plug-in class.

# BaseObject

See Also: [Class BaseObject](#).

This is the base class for objects. Anything with a representation in the 3ds max 3D viewports is derived from BaseObject (including Modifiers, whose gizmos appear in the viewports).

The main methods of this class are:

## **Display()**

This is called by the system to have the item display itself.

## **HitTest()**

This allows the system to determine if a mouse point intersects the item.

## **Snap()**

This method allows the item to participate in the 3ds max snapping system.

There are several methods to retrieve the bounding box for the object.

## **GetWorldBoundingBox()**

This is the world space bounding box.

## **GetLocalBoundingBox()**

This is the object space bounding box.

There are also methods for sub-object selection. Both procedural objects and modifiers can participate in sub-object selection.

## Class BaseObject

See Also: [Class ReferenceTarget](#), [Class INode](#), [Class ViewExp](#), [Class Box3](#), [Class IPoint2](#), [Class Matrix3](#), [Structure SnapInfo](#), [Class Point3](#), [Class CreateMouseCallBack](#), [Template Class Tab](#), [Class Interface](#).

```
class BaseObject : public ReferenceTarget
```

### Description:

This is the base class for objects and modifiers. Anything with a representation in the 3D viewports is derived from BaseObject (including modifiers and controllers whose gizmos appear in the viewports). The methods here are things such as displaying the object in the viewports, checking for intersection of the object and points the user clicks with the mouse, snapping to the object, and computing various bounding boxes for the object. Also there are methods for returning the name of the object to appear in the modifier stack, a method to deal with creating the object in the viewports (if appropriate), and named selection set related methods. There are also methods that allow other plug-ins to access the changeable parameters of the object. Finally there are several method that deal with sub-object selection, sub-object display, sub-object hit testing, and moving/rotating/scaling sub-object components of the object.

### Method Groups:

The hyperlinks below jump to the start of groups of related methods within the class:

[Object Name and Create Callback](#)

[Object Level Display, Hit Test, Snap, and Bounding Box Methods](#)

[Parameter Block Access](#)

[Named Selection Sets](#)

[Topology Change](#)

[Sub-Object Selection](#)

[Sub-Object Cloning](#)

[Sub-Object Display, Hit Testing, and Bound Box Methods](#)

[Sub-Object Center and TM Methods](#)

### Methods:

#### Object Name and Create Callback

**Prototype:**

```
virtual TCHAR *GetObjectName()
```

**Remarks:**

Implemented by the Plug-In.

Returns the name that will appear in the history browser (modifier stack).

**Default Implementation:**

```
{ return _T("Object"); }
```

**Prototype:**

```
virtual CreateMouseButton* GetCreateMouseButton();
```

**Remarks:**

Implemented by the Plug-In.

This method allows the system to retrieve a callback object used in creating an object in the 3D viewports. This method returns a pointer to an instance of a class derived from **CreateMouseButton**. This class has a method **proc()** which is where the programmer defines the user/mouse interaction during the object creation phase.

**Return Value:**

A pointer to an instance of a class derived from **CreateMouseButton**.

**Named Selection Sets**

A modifier that supports sub-object selection can choose to support named sub-object selection sets. Methods in the interface passed to objects allow them to add items to the sub-object selection set drop down list in the 3ds max toolbar. A modifier that wishes to support this capability maintains its list of named sub-object selections. When the user enters sub-object selection mode the modifier adds its named selection sets into the drop down (using **Interface::**

**AppendSubObjectNamedSelSet()**). See the Advanced Topics section on [Sub-Object Selection](#) for details.

**Prototype:**

```
virtual BOOL SupportsNamedSubSels();
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if the plug-in supports named sub-object selection sets; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**The following three methods are called when the user picks items from the drop down list.**

**Prototype:**

```
virtual void ActivateSubSelSet(TSTR &setName);
```

**Remarks:**

Implemented by the Plug-In.

When the user chooses a name from the drop down list this method is called.

The plug-in should respond by selecting the set identified by the name passed.

**Parameters:**

**TSTR &setName**

The name of the set to select.

**Prototype:**

```
virtual void NewSetFromCurSel(TSTR &setName);
```

**Remarks:**

Implemented by the Plug-In.

If the user types a new name into the named selection set drop down then this method is called. The plug-in should respond by creating a new set and give it the specified name.

**Parameters:**

**TSTR &setName**

The name for the selection set.

**Prototype:**

```
virtual void RemoveSubSelSet(TSTR &setName);
```

**Remarks:**

Implemented by the Plug-In.

If the user selects a set from the drop down and then chooses Remove Named Selections from the Edit menu this method is called. The plug-in should respond by removing the specified selection set.

**Parameters:****TSTR &setName**

The selection set to remove.

**Prototype:**

```
virtual void SetupNamedSelDropDown();
```

**Remarks:**

This method is available in release 2.0 and later only.

To support the new Edit Named Selections dialog, plug-ins must implement this method.

This method is called to rebuild the named selection set drop down list. This is usually done by calling **Interface::ClearSubObjectNamedSelSets()** followed by calls to **Interface::AppendSubObjectNamedSelSet()**.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual int NumNamedSelSets();
```

**Remarks:**

This method is available in release 2.0 and later only.

To support the new Edit Named Selections dialog, plug-ins must implement this method.

Returns the number of named selection sets.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

**virtual TSTR GetNamedSelSetName(int i);**

**Remarks:**

This method is available in release 2.0 and later only.

To support the new Edit Named Selections dialog, plug-ins must implement this method.

Returns the name of the 'i-th' named selection set.

**Parameters:**

**int i**

The index of the selection set whose name is returned.

**Default Implementation:**

```
{return _T("");}
```

**Prototype:**

**virtual void SetNamedSelSetName(int i,TSTR &newName);**

**Remarks:**

This method is available in release 2.0 and later only.

To support the new Edit Named Selections dialog, plug-ins must implement this method.

This methods sets the name of the selection set whose index is passed to the name passed.

Note: Developers need to implement Undo / Redo for modifications to their named selection sets.

**Parameters:**

**int i**

The index of the selection set whose name is to be set.

**TSTR &newName**

The new name for the selection set the plug-in should store.

**Default Implementation:**

```
{}
```

**Prototype:**

**virtual void NewSetByOperator(TSTR &newName,Tab<int>**

**&sets,int op);**

**Remarks:**

This method is available in release 2.0 and later only.

To support the new Edit Named Selections dialog, plug-ins must implement this method.

The user may bring up the Edit Named Selections dialog via the Edit / Edit Named Selection ... command. This dialog allows the user to create new selection sets using 'boolean' operations to the sets including 'Combine', 'Subtract (A-B)', 'Subtract (B-A)' and 'Intersection'. This method is called on the plug-in to generate a new selection set via one of these operations.

This method assumes the developer will append a **new** selection set with the name passed. This will result in two sets with identical names. Then the system will call **RemoveSubSelSet()** afterwards, so that the first one that is found (the old one, since the new one was appended) will be deleted.

Note: Developers need to implement Undo / Redo for modifications to their named selection sets. See

**\MAXSDK\SAMPLES\MODIFIERS\MESHSEL.CPP** for an example.

**Parameters:**

**TSTR &newName**

The new name for the selection set is passed here.

**Tab<int> &sets**

A table of the selection sets to operate on. There are **sets.Count()** sets in the table.

**int op**

One of the following values:

**NEWSET\_MERGE**

The sets should be merged.

**NEWSET\_INTERSECTION**

The sets should be intersected -- that is the items common to both sets should appear in the new set.

**NEWSET\_SUBTRACT**

The new set should be the result of subtracting the 1st thru nth set from the

0th set.

**Default Implementation:**

```
{}
```

## **Object Level Display, Hit Test, Snap, and Bounding Box Methods**

**Prototype:**

```
virtual int Display(TimeValue t, INode* inode, ViewExp *vpt, int flags)
```

**Remarks:**

Implemented by the Plug-In.

This is called by the system to have the item display itself (perform a quick render in viewport, using the current TM). Note: For this method to be called the object's validity interval must be invalid at the specified time **t**. If the interval is valid, the system may not call this method since it thinks the display is already valid.

**Parameters:**

**TimeValue t**

The time to display the object.

**INode\* inode**

The node to display.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**int flags**

See [List of Display Flags](#).

**Return Value:**

The return value is not currently used.

**Default Implementation:**

```
{ return 0; };
```

**Prototype:**

```
virtual void SetExtendedDisplay(int flags);
```

**Remarks:**

This method is used for storing mode-dependent display attributes.

Before an object's Display() method is called, the appropriate bits of the extended display flag variable are set and this method is called. After that, the Display() method is called. If the object must display itself differently based on the settings of the extended display bit fields, then the object must save the flags passed into the this method. Otherwise, there is no need for the object to store the flags.

**Parameters:****int flags**

The flags to store.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual int HitTest(TimeValue t, INode* inode, int type, int crossing, int flags, IPoint2 *p, ViewExp *vpt);
```

**Remarks:**

Implemented by the Plug-In.

This method is called to determine if the specified screen point intersects the item. The method returns nonzero if the item was hit; otherwise 0.

**Parameters:****TimeValue t**

The time to perform the hit test.

**INode\* inode**

A pointer to the node to test.

**int type**

The type of hit testing to perform. See [Hit Test Types](#) for details.

**int crossing**

The state of the crossing setting. If TRUE crossing selection is on.

**int flags**

The hit test flags. See [Hit Test Flags](#) for details.

**IPoint2 \*p**

The screen point to test.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**Return Value:**

Nonzero if the item was hit; otherwise 0.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void Snap(TimeValue t, INode* inode, SnapInfo *snap,  
IPoint2 *p, ViewExp *vpt);
```

**Remarks:**

Implemented by the Plug-In.

Checks the point passed for a snap and updates the **SnapInfo** structure.

Note: The new snapping system introduced in release 2.0 causes this method to no longer be called. Developers wanting to find snap points on an Editable Mesh object should see the method **XmeshSnap::Snap()** in **\MAXSDK\SAMPLES\SNAPS\XMESH\XMESH.CPP**.

**Parameters:**

**TimeValue t**

The time to check.

**INode\* inode**

The node to check.

**SnapInfo \*snap**

The snap info structure to update.

**IPoint2 \*p**

The screen point to check.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**Prototype:**

**virtual void GetWorldBoundingBox(TimeValue t, INode \* inode, ViewExp\* vp, Box3& box )**

**Remarks:**

Implemented by the Plug-In.

This method returns the world space bounding box for Objects (see below for the Sub-object gizmo or Modifiers gizmo version). The bounding box returned by this method does not need to be precise. It should however be calculated rapidly. The object can handle this by transforming the 8 points of its local bounding box into world space and take the minimums and maximums of the result. Although this isn't necessarily the tightest bounding box of the objects points in world space, it is close enough.

**Parameters:****TimeValue t**

The time to compute the bounding box.

**INode\* inode**

The node to calculate the bounding box for.

**ViewExp\* vp**

An interface pointer that may be used to call methods associated with the viewports.

**Box3& box**

The bounding box is returned through **box**.

**Prototype:**

**virtual void GetLocalBoundingBox(TimeValue t, INode\* inode, ViewExp\* vp, Box3& box )**

**Remarks:**

Implemented by the Plug-In.

This is the object space bounding box, the box in the object's local coordinates. The system expects that requesting the object space bounding box will be fast.

**Parameters:****TimeValue t**

The time to retrieve the bounding box.

**INode\* inode**

The node to calculate the bounding box for.

**ViewExp\* vp**

An interface pointer that may be used to call methods associated with the viewports.

**Box3& box**

The bounding box is returned through **box**.

## Topology Methods

**Prototype:**

**virtual BOOL OKToChangeTopology(TSTR &modName)**

**Remarks:**

Implemented by the System.

This method is called to see if any modifiers down in the pipeline depend on topology. It sends the message

**REFMSG\_IS\_OK\_TO\_CHANGE\_TOPOLOGY** to see if any modifiers or objects down the pipeline depend on topology.

**Parameters:**

**TSTR &modName**

This parameter is set to the dependent modifier's name if there is an item that depends on topology.

**Return Value:**

Returns TRUE if it is okay to change the topology; FALSE if it is not okay to change the topology.

**Prototype:**

**virtual BOOL ChangeTopology()**

**Remarks:**

Implemented by the Plug-In.

This method asks the question of an object or modifier "Do you change topology"? An object or modifier returns TRUE if it is capable of changing

topology when its parameters are being edited; otherwise FALSE.

When an item is selected for editing, and there is a modifier in the pipeline that depends on topology, the system calls this method to see if it may potentially change the topology. If this method returns TRUE the system will put up a warning message indicating that a modifier exists in the stack that depends on topology.

**Default Implementation:**

```
{return TRUE;}
```

**Prototype:**

```
virtual void ForceNotify(Interval& i)
```

**Remarks:**

This method is no longer used.

**Parameter Block Access**

**Prototype:**

```
virtual IParamArray *GetParamBlock();
```

**Remarks:**

Implemented by the Plug-In.

An object or modifier should implement this method if it wishes to make its parameter block available for other plug-ins to access it. The system itself doesn't actually call this method. This method is optional.

**Return Value:**

A pointer to the item's parameter block. See [Class IParamArray](#).

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual int GetParamBlockIndex(int id);
```

**Remarks:**

Implemented by the Plug-In.

If a plug-in makes its parameter block available (using **GetParamBlock()**) then it will need to provide #defines for indices into the parameter block. These defines should not be directly used with the parameter block but instead converted by this function that the plug-in implements. This way if a parameter moves around in a future version of the plug-in the #define can be remapped. A return value of -1 indicates an invalid parameter id.

**Parameters:**

**int id**

The parameter block id. See: [List of Parameter Block IDs](#).

**Return Value:**

The parameter block index or -1 if it is invalid.

**Default Implementation:**

```
{return -1;}
```

## Sub-Object Selection

Affine transformation methods

**Prototype:**

```
virtual void Move(TimeValue t, Matrix3& partm, Matrix3& tmAxis, Point3& val, BOOL localOrigin=FALSE )
```

**Remarks:**

Implemented by the Plug-In.

When this method is called the plug-in should respond by moving its selected sub-object components.

**Parameters:**

**TimeValue t**

The time of the transformation.

**Matrix3& partm**

The 'parent' transformation matrix. This matrix represents a transformation that would take points in the modifier's space and convert them into world space points. This is constructed as the node's transformation matrix times the inverse of the ModContext's transformation matrix. The node whose transformation is used is the node the user clicked on in the scene - modifiers can be instanced so there could be more than one node.

### **Matrix3& tmAxis**

The matrix that represents the axis system. This is the space in which the transformation is taking place.

### **Point3& val**

This value is a vector with X, Y, and Z representing the movement along each axis.

### **BOOL localOrigin=FALSE**

When TRUE the transformation is occurring about the sub-object's local origin.

### **Prototype:**

```
virtual void Rotate(TimeValue t, Matrix3& partm, Matrix3& tmAxis, Quat& val, BOOL localOrigin=FALSE )
```

### **Remarks:**

Implemented by the Plug-In.

When this method is called the plug-in should respond by rotating its selected sub-object components.

### **Parameters:**

#### **TimeValue t**

The time of the transformation.

#### **Matrix3& partm**

The 'parent' transformation matrix. This matrix represents a transformation that would take points in the modifier's space and convert them into world space points. This is constructed as the node's transformation matrix times the inverse of the ModContext's transformation matrix. The node whose transformation is used is the node the user clicked on in the scene - modifiers can be instanced so there could be more than one node.

#### **Matrix3& tmAxis**

The matrix that represents the axis system. This is the space in which the transformation is taking place.

#### **Quat& val**

The amount to rotate the selected components.

#### **BOOL localOrigin=FALSE**

When TRUE the transformation is occurring about the sub-object's local

origin. Note: This information may be passed onto a transform controller (if there is one) so they may avoid generating 0 valued position keys for rotation and scales. For example if the user is rotating an item about anything other than its local origin then it will have to translate in addition to rotating to achieve the result. If a user creates an object, turns on the animate button, and rotates the object about the world origin, and then plays back the animation, the object does not do what the was done interactively. The object ends up in the same position, but it does so by both moving and rotating. Therefore both a position and a rotation key are created. If the user performs a rotation about the local origin however there is no need to create a position key since the object didn't move (it only rotated). So a transform controller can use this information to avoid generating 0 valued position keys for rotation and scales.

**Prototype:**

**virtual void Scale(TimeValue t, Matrix3& partm, Matrix3& tmAxis, Point3& val, BOOL localOrigin=FALSE )**

**Remarks:**

Implemented by the Plug-In.

When this method is called the plug-in should respond by scaling its selected sub-object components.

**Parameters:**

**TimeValue t**

The time of the transformation.

**Matrix3& partm**

The 'parent' transformation matrix. This matrix represents a transformation that would take points in the modifier's space and convert them into world space points. This is constructed as the node's transformation matrix times the inverse of the ModContext's transformation matrix. The node whose transformation is used is the node the user clicked on in the scene - modifiers can be instanced so there could be more than one node.

**Matrix3& tmAxis**

The matrix that represents the axis system. This is the space in which the transformation is taking place.

**Point3& val**

This value is a vector with X, Y, and Z representing the scale along X, Y, and

Z respectively.

**BOOL localOrigin=FALSE**

When TRUE the transformation is occurring about the sub-object's local origin. See the note above in the Rotate method.

The following methods may be used to receive notification about the starting and ending phases of transforming the item when in sub-object selection.

**Prototype:**

**virtual void TransformStart(TimeValue t);**

**Remarks:**

Implemented by the Plug-In.

This method is called before the first **Move()**, **Rotate()** or **Scale()** call and before a hold is in effect.

**Parameters:**

**TimeValue t**

The current time when this method is called.

**Prototype:**

**virtual void TransformHoldingStart(TimeValue t);**

**Remarks:**

Implemented by the Plug-In.

This method is called before the first **Move()**, **Rotate()** or **Scale()** call and after a hold is in effect.

**Parameters:**

**TimeValue t**

The current time when this method is called.

**Prototype:**

**virtual void TransformHoldingFinish(TimeValue t);**

**Remarks:**

Implemented by the Plug-In.

This method is called after the user has completed the **Move()**, **Rotate()** or

**Scale()** operation and before the undo object has been accepted.

**Parameters:**

**TimeValue t**

The current time when this method is called.

**Prototype:**

**virtual void TransformFinish(TimeValue t);**

**Remarks:**

Implemented by the Plug-In.

This method is called after the user has completed the **Move()**, **Rotate()** or **Scale()** operation and the undo object has been accepted.

**Parameters:**

**TimeValue t**

The current time when this method is called.

**Prototype:**

**virtual void TransformCancel(TimeValue t);**

**Remarks:**

This method is called when the transform operation is canceled by a right-click and the undo has been canceled.

**Parameters:**

**TimeValue t**

The current time when this method is called.

## **Sub-Object Display, Hit Test, and Bounding Box Methods**

The following methods are for sub-object selection. If the derived class is NOT a modifier, the modContext pointer passed to some of these methods will be NULL.

**Prototype:**

**virtual int Display(TimeValue t, INode\* inode, ViewExp \*vpt, int flags, ModContext\* mc)**

**Remarks:**

Implemented by the Plug-In.

When this method is called the plug-in should respond by performing a quick render of the modifier gizmo in viewport using the current TM.

Note for Modifiers: For this method to be called properly, one must send two reference messages using NotifyDependents.

In BeginEditParams send:

```
NotifyDependents(Interval(t,t), PART_ALL,  
REFMSG_MOD_DISPLAY_ON);
```

In EndEditParams send:

```
NotifyDependents(Interval(t,t), PART_ALL,  
REFMSG_MOD_DISPLAY_OFF);
```

**Parameters:****TimeValue t**

The time to display the item.

**INode\* inode**

The node to render.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**int flags**

See [List of Display Flags](#).

**ModContext\* mc**

A pointer to the modifiers ModContext.

**Return Value:**

Nonzero if the item was displayed; otherwise 0.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual int HitTest(TimeValue t, INode* inode, int type, int  
crossing, int flags, IPoint2 *p, ViewExp *vpt, ModContext* mc)
```

**Remarks:**

Implemented by the Plug-In.

This method is used in modifier gizmo hit testing. It is called to determine if the specified screen point intersects the gizmo. The method returns nonzero if the item was hit; otherwise 0.

**Parameters:****TimeValue t**

The time to perform the hit test.

**INode\* inode**

A pointer to the node to test.

**int type**

The type of hit testing to perform. See [Hit Test Types](#) for details.

**int crossing**

The state of the crossing setting. If TRUE crossing selection is on.

**int flags**

The hit test flags. See [Hit Test Flags](#) for details.

**IPoint2 \*p**

The screen point to test.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**ModContext\* mc**

A pointer to the modifiers ModContext.

**Return Value:**

Nonzero if the item was hit; otherwise 0.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual void GetWorldBoundingBox(TimeValue t, INode* inode,  
ViewExp *vpt, Box3& box, ModContext *mc)
```

**Remarks:**

Implemented by the Plug-In.

This method computes the world space bounding box of the modifier gizmo (or any object that when in sub-object mode has a gizmo).

**Parameters:**

**TimeValue t**

The time to compute the bounding box.

**INode\* inode**

The node to calculate the bounding box for.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

**Box3& box**

The bounding box is returned through **box**.

**ModContext \*mc**

A pointer to the modifiers ModContext.

**Prototype:**

**virtual BOOL HasUVW();**

**Remarks:**

Implemented by the Plug-In.

This method is available in release 2.0 and later only.

It is called to find out if the object is has UVW coordinates. This method returns TRUE if the object has UVW coordinates; otherwise FALSE. In 3ds max 2.0 and later there is code in the renderer that will automatically turn on the UVW coordinates of the base object if UV's are missing (and needed). The base object has to implement two simple methods to make this work:

**HasUVW()** and **SetGenUVW()**.

Developers are encouraged to put these methods in their objects: it makes using the program easier for the user. If they are not implemented, it doesn't cause any real harm: it will just operate as before and put up the missing UVW's message.

Here is how the procedural sphere implements these methods:

```
BOOL SphereObject::GetGenUVW() {
```

```

BOOL genUVs;
Interval v;
pblock->GetValue(PB_GENUVS, 0, genUVs, v);
return genUVs;
}
void SphereObject::SetGenUVW(BOOL sw) {
if (sw==GetGenUVW()) return;
pblock->SetValue(PB_GENUVS,0, sw);
}

```

Important Note: The **pblock->SetValue()** will cause a call to **NotifyDependents(FOREVER, PART\_TEXMAP, REFMSG\_CHANGE)**, which will invalidate the UVW cache. It is essential that this call be made, so if the 'generate UVW' boolean is not handled by a parameter block, then **NotifyDependents()** needs to be called explicitly.

Also Note: For "modifiable objects" that pass up the pipeline getting modified, such as **TriObject**, **EditTriObject**, etc., which cannot generate their own UVWs, but can carry them in their data structures, only this **HasUVW()** method needs to be implemented. For example, here is the implementation for **TriObject**:

```

BOOL TriObject::HasUVW() { return mesh.tvFace?1:0; }

```

**Default Implementation:**

```

{ return 1; }

```

**Prototype:**

```

virtual BOOL HasUVW(int mapChannel);

```

**Remarks:**

This method is available in release 3.0 and later only.

It is called to find out if the object is has UVW coordinates for the specified mapping channel. This method returns TRUE if the object has UVW coordinates; otherwise FALSE. See the method **HasUVW()** above for more details.

**Parameters:**

**int mapChannel**

See [List of Mapping Channels Values](#).

**Default Implementation:**

```
{ return (mapChannel==1) ? HasUVW() : FALSE; }
```

**Prototype:**

```
virtual void SetGenUVW(BOOL sw);
```

**Remarks:**

Implemented by the Plug-In.

This method is available in release 2.0 and later only.

This method is called to change the state of its Generate UVW boolean. If the state changes, the object must send a **REFMSG\_CHANGE** up the pipeline by calling **NotifyDependents()**. This applies to map channel 1.

**Parameters:**

**BOOL sw**

The new state for the generate UVW flag.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void SetGenUVW(int mapChannel, BOOL sw);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method is called to change the state of its Generate UVW boolean for the specified mapping channel. If the state changes, the object must send a **REFMSG\_CHANGE** up the pipeline by calling **NotifyDependents()**.

**Parameters:**

**int mapChannel**

The mapping channel index. See [List of Mapping Channel Index Values](#).

**BOOL sw**

The new state for the generate UVW flag.

**Default Implementation:**

```
{ if (mapChannel==1) SetGenUVW(sw); }
```

**Prototype:**

```
virtual void ShowEndResultChanged(BOOL showEndResult);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method notifies the BaseObject that the end result display has been switched (the "Show End Result" button has been toggled). Sometimes this is needed for display changes.

This method is implemented in Edit Mesh, which uses it as shown below:

```
void EditMeshMod::ShowEndResultChanged(BOOL  
showEndResult) {  
    NotifyDependents(FOREVER, PART_DISPLAY,  
REFMSG_CHANGE);  
}
```

This allows the Edit Mesh modifier to update itself in response to a user click of the "Show End Result" button in the modifier panel.

**Parameters:**

**BOOL showEndResult**

TRUE if Show End Result is on; FALSE if off.

**Default Implementation:**

```
{ }
```

## Sub-Object Cloning

**Prototype:**

```
virtual void CloneSelSubComponents(TimeValue t)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to make a copy of the selected sub-object components of

the item. If this is called on an object, the selection level of the object is used to determine which type of sub-objects are cloned. For instance in a Mesh, the selection level determines if selected vertices, edges or faces are cloned. If this is called on a Modifier then the selection level of the modifier is used. Modifiers call **Interface::GetModContexts()** to get a list of ModContexts, one for each object the modifier is applied to. Then the selected sub-objects are cloned for each object in the list.

**Parameters:**

**TimeValue t**

The time at which to clone the selected sub-object components.

**Prototype:**

**virtual void AcceptCloneSelSubComponents(TimeValue t)**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called when the user mouses up after shift-cloning a sub-object selection.

**Parameters:**

**TimeValue t**

The time at which the clone of the selected components is being done.

**Default Implementation:**

**{}**

## **Sub-Object Selection**

**Prototype:**

**virtual void SelectSubComponent(HitRecord \*hitRec, BOOL selected, BOOL all, BOOL invert=FALSE)**

**Remarks:**

Implemented by the Plug-In.

This method is called to change the selection state of the component identified by **hitRec**.

**Parameters:**

## **HitRecord \*hitRec**

Identifies the component whose selected state should be set. See [Class HitRecord](#) .

## **BOOL selected**

TRUE if the item should be selected; FALSE if the item should be de-selected.

## **BOOL all**

TRUE if all components in the HitRecord chain should be selected; FALSE if only the top-level HitRecord should be selected. (A HitRecord contains a Next() pointer; typically you want to do whatever you're doing to all the Next()'s until Next() returns NULL).

## **BOOL invert=FALSE**

This is set to TRUE when **all** is also set to TRUE and the user is holding down the Shift key while region selecting in select mode. This indicates the items hit in the region should have their selection state inverted

### **Prototype:**

```
virtual void ClearSelection(int selLevel);
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to clear the selection for the given sub-object level. All sub-object elements of this type should be deselected. This will be called when the user chooses Select None from the 3ds max Edit menu.

### **Parameters:**

**int selLevel**

Specifies the selection level to clear.

### **Default Implementation:**

```
{
```

### **Prototype:**

```
virtual void SelectAll(int selLevel);
```

### **Remarks:**

Implemented by the Plug-In.

This method is called to select every element of the given sub-object level.

This will be called when the user chooses Select All from the 3ds max Edit menu.

**Parameters:**

**int selLevel**

Specifies the selection level to select.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void InvertSelection(int selLevel);
```

**Remarks:**

This method is called to invert the specified sub-object level. If the element is selected it should be deselected. If it's deselected it should be selected. This will be called when the user chooses Select Invert from the 3ds max Edit menu.

**Parameters:**

**int selLevel**

Specifies the selection level to invert.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual int SubObjectIndex(HitRecord *hitRec);
```

**Remarks:**

Implemented by the Plug-In.

Returns the index of the sub-object element identified by the HitRecord **hitRec**. See [Class HitRecord](#). The sub-object index identifies a sub-object component. The relationship between the index and the component is established by the modifier. For example an edit modifier may allow the user to select a group of faces and these groups of faces may be identified as group 0, group 1, group 2, etc. Given a hit record that identifies a face, the edit modifier's implementation of this method would return the group index that the face belonged to.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void ActivateSubobjSel(int level, XFormModes& modes )
```

**Remarks:**

Implemented by the Plug-In.

When the user changes the selection of the sub-object drop down, this method is called to notify the plug-in. This method should provide instances of a class derived from [CommandMode](#) to support move, rotate, non-uniform scale, uniform scale, and squash modes. These modes replace their object mode counterparts however the user still uses the move/rotate/scale tool buttons in the toolbar to activate them. If a certain level of sub-object selection does not support one or more of the modes NULL may be passed. If NULL is specified the corresponding toolbar button will be grayed out.

**Parameters:****int level**

The sub-object selection level the command modes should be set to support. A **level** of 0 indicates object level selection. If **level** is greater than or equal to 1 the index refers to the types registered by the object in the order they appeared in the list when registered by **Interface::RegisterSubObjectTypes()**. See [Class Interface](#).

**XFormModes& modes**

The command modes to support

**Sample Code:**

```
void SimpleMod::ActivateSubobjSel(int level, XFormModes& modes )
{
    switch ( level ) {
        case 1: // Modifier box
            modes = XFormModes(moveMode,rotMode,nuscaleMode,
                uscaleMode,squashMode,NULL);
            break;
        case 2: // Modifier Center
            modes =
```

```
XFormModes(moveMode,NULL,NULL,NULL,NULL,NULL);
    break;
}
NotifyDependents(FOREVER,PART_DISPLAY,REFMSG_CHANGE);
}
```

See Also: [Class XFormModes](#).

## Sub-Object Centers and TMs

### Prototype:

```
virtual void GetSubObjectCenters(SubObjAxisCallback
*cb,TimeValue t,INode *node, ModContext *mc)
```

### Remarks:

Implemented by the Plug-In.

When the user is in a sub-object selection level, the system needs to get the reference coordinate system definition from the current modifier being edited so that it can display the axis. This method specifies the position of the center. The plug-in enumerates its centers and calls the callback **cb** once for each.

### Parameters:

**SubObjAxisCallback \*cb**

The callback object whose methods may be called. See [Class SubObjAxisCallback](#).

**TimeValue t**

The time to enumerate the centers.

**INode \*node**

A pointer to the node.

**ModContext \*mc**

A pointer to the ModContext.

### Prototype:

```
virtual void GetSubObjectTMs(SubObjAxisCallback *cb,
TimeValue t,INode *node, ModContext *mc)
```

### Remarks:

Implemented by the Plug-In.

When the user is in a sub-object selection level, the system needs to get the reference coordinate system definition from the current modifier being edited so that it can display the axis. This method returns the axis system of the reference coordinate system. The plug-in enumerates its TMs and calls the callback **cb** once for each. See [Sub-Object Coordinate Systems](#).

**Parameters:**

**SubObjAxisCallback \*cb**

The callback object whose methods may be called.

**TimeValue t**

The time to enumerate the TMs.

**INode \*node**

A pointer to the node.

**ModContext \*mc**

A pointer to the ModContext.

**Prototype:**

```
virtual void NotifyPreCollapse(INode *node, IDerivedObject  
*derObj, int index);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is called before a modifier or object is collapsed. See also [Class NotifyCollapseEnumProc](#).

**Parameters:**

**INode \*node**

Points to the node for the object being collapsed.

**IDerivedObject \*derObj**

If the object associated with **node** above is a Modifier this points to the derived object. If it's an object then this is NULL.

**int index**

If the object associated with **node** above is a Modifier this is the index of this modifier in the DerivedObject.. If it's an object then this is 0.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void NotifyPostCollapse(INode *node, Object *obj,  
IDerivedObject *derObj, int index);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is called after a modifier or object is collapsed. See also [Class NotifyCollapseEnumProc](#).

**Parameters:**

**INode \*node**

Points to the node for the object being collapsed.

**IDerivedObject \*derObj**

If the object associated with **node** above is a Modifier this points to the derived object. If it's an object then this is NULL.

**int index**

If the object associated with **node** above is a Modifier this is the index of this modifier in the DerivedObject. If it's an object then this is 0.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual int NumSubObjTypes();
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns the number of sub-object types supported by the object or modifier. In R4 objects or modifier must override this method and GetSubObjType() below.

**Default Implementation:**

```
{ return 0;}
```

**Prototype:**

**virtual ISubObjType \*GetSubObjType(int i);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the sub-object type for the sub-object whose index is passed.

If the parameter **i** passed is -1 the system requests an **ISubObjType** for the current SubObjectLevel that flows up the modifier stack. If the subobject selection of the modifier or base object does not affect the sub-object selection that flows up the stack NULL must be returned. See the sample code in **\MAXSDK\SAMPLES\MODIFIERS\MESHSEL.CPP**.

**Parameters:**

**int i**

The zero based index of the sub-object type to get. See the remarks above.

**Return Value:**

The sub-object type. See [Class ISubObjType](#).

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

**virtual int GetSubObjectLevel();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns an integer which indicates the current sub-object level of the modifier or base object.

**Return Value:**

A value of 0 indicates object level. A value of 1 through

**NumSubObjTypes()** are the sub-object levels in the same order as they are returned by **GetSubObjType(int i)** (with an offset of 1 of course).

# CameraObject

See Also: [Class CameraObject](#).

This is the class from which plug-in cameras are derived. Camera objects implement methods that return the properties of the camera. Some of these methods are:

## **EvalCameraState()**

This returns a data structure containing the FOV and distance to the target.

These can also be accessed by:

## **Get/SetFOV()**

Provides access to the field of view of the camera.

## **Get/SetTDist()**

Provides access to the target distance of the camera.

# ConstObject

See Also: [Class ConstObject](#).

Plug-In construction objects such as the Grid Helper Object are subclassed off ConstObject.

# Control

See Also: [Class Control](#).

Controllers are the objects in 3ds max that control animation. Controllers come in different types based on the type of data they control. For example, transform controllers control the 4x3 matrices used to define the position of nodes in the scene while float controllers control simple floating point values.

Control is the class from which you may derived controller objects. Shown below are several of the methods of Control:

## **GetValue()**

Retrieves the value of the controller at the specified time, and updates the validity interval passed in to reflect the interval of the controller.

## **SetValue()**

Sets the value of the controller at the specified time.

## **IsLeaf()**

The controller should return TRUE if it has no sub controllers. For example, a PRS controller is not a leaf controller (because it has sub-controllers for Position, Rotation and Scale), but a simple keyframed float controller is a leaf controller.

## **Copy()**

When a controller is assigned to a track in the track view the new controller is plugged into the parameter and then this method is called on the new controller with a pointer to the old controller passed in. The new controller can then attempt to copy any data that it can from the old controller. At the very least it should initialize itself to the value of the old controller at frame 0.

# GeomObject

See Also: [Class GeomObject](#).

GeomObjects are objects that have geometry and are renderable. The procedural sphere, cylinder and box are examples of GeomObjects. In addition to the regular object methods, objects of this class implement these methods:

## **IsInstanceDependent()**

If an object creates different meshes depending on the particular instance (view-dependent) it should return nonzero; otherwise 0.

## **GetRenderMesh();**

This method provides a mesh representation of the object for use by the renderer.

## **GetRenderPatchMesh()**

This method provides a patch mesh representation of the object for use by the renderer. If this method returns NULL, then **GetRenderMesh()** will be called.

# HelperObject

See Also: [Class HelperObject](#).

This is the class from which plug-in helper objects are derived. These are objects that are used as aids in the scene such as points, tape measurers, dummy objects, etc.

# **INode**

See Also: [Class INode](#).

The INode class is the interface to nodes in the scene. It provides methods to access various properties of a node such as its name, wire-frame color, transformation matrix, etc.

# Class INode

See Also: [Class ReferenceTarget](#), [Class FPMixinInterface](#), [Class Interface](#), [Class INodeTransformed](#), [Class Material](#), [Class Matrix3](#), [Class Quat](#), [Class AngAxis](#), [Class Control](#), [Class Object](#), [Class INodeTab](#).

```
class INode : public ReferenceTarget, public FPMixinInterface
```

## Description:

The INode class is the interface to nodes in the scene. It provides methods to access various parts of a node such as its name, transformation matrices, parents and children, display status, etc. All methods of this class are implemented by the system.

## Method Groups:

These hyperlinks take you to the start of groups of related methods within the class.

[Pipeline Evaluation](#)

[Node Naming](#)

[Parent-Child/Hierarchy / Groups](#)

[Target / Node Selection](#)

[Object Reference](#)

[Display Attributes](#)

[Rendering Attributes](#)

[Vertex Color Attributes](#)

[Access to Node Transformation Matrices](#)

[Object Offset Adjustment](#)

[Access to Controllers](#)

[Access to Visibility Properties](#)

[Access to Render Data](#)

[Materials](#)

[Put Nodes in Foreground Plane](#)

[Active Grid Node](#)

[Temporary Storage of Data with Nodes](#)

[User Properties](#)

[Geometry/Graphics \(G\) Buffer Access](#)

[INodeTransformed methods](#)

[IK Related Methods](#)

[Deleting the Node](#)

[WSMDerivedObject Access](#)

[XRef Access](#)

[Bone Methods](#)

## Methods:

### Prototype:

```
virtual const ObjectState& EvalWorldState(TimeValue  
time,BOOL evalHidden=TRUE)=0;
```

### Remarks:

This method should be called when a developer needs to work with an object that is the result of the node's pipeline. This is the object that appears in the scene.

This may not be an object that anyone has a reference to - it may just be an object that has flowed down the pipeline. For example, if there is a Sphere in the scene that has a Bend and Taper applied, **EvalWorldState()** would return an **ObjectState** containing a TriObject. This is the result of the sphere turning into a TriObject and being bent and tapered (just as it appeared in the scene).

If a developer needs to access the object that the node in the scene references, then the method **INode::GetObjectRef()** should be used instead. See [Object Reference Methods](#)

### Parameters:

#### **TimeValue time**

Specifies the time to retrieve the object state.

#### **BOOL evalHidden=TRUE**

If FALSE and the node is hidden, the pipeline will not actually be evaluated (however the TM will).

### Return Value:

The ObjectState that is the result of the pipeline. See [Class ObjectState](#).

### Sample Code:

```
// Get the object from the node  
ObjectState os = node->EvalWorldState(ip->GetTime());
```

```

if (os.obj->SuperClassID()==GEOMOBJECT_CLASS_ID) {
    obj = (GeomObject*)os.obj;
    ...
}

```

The following code shows how a **TriObject** can be retrieved from a node. Note on the code that if you call **ConvertToType()** on an object and it returns a pointer other than itself, you are responsible for deleting that object.

```

// Retrieve the TriObject from the node
int deleteIt;
TriObject *triObject = GetTriObjectFromNode(ip->GetSelNode(0),
    deleteIt);
// Use the TriObject if available
if (!triObject) return;
// ...
// Delete it when done...
if (deleteIt) triObject->DeleteMe();

// Return a pointer to a TriObject given an INode or return NULL
// if the node cannot be converted to a TriObject
TriObject *Utility::GetTriObjectFromNode(INode *node, int
&deleteIt) {
    deleteIt = FALSE;
    Object *obj = node->EvalWorldState(ip->GetTime()).obj;
    if (obj->CanConvertToType(Class_ID(TRIOBJ_CLASS_ID, 0))) {
        TriObject *tri = (TriObject *) obj->ConvertToType(ip->GetTime(),
            Class_ID(TRIOBJ_CLASS_ID, 0));
        // Note that the TriObject should only be deleted
        // if the pointer to it is not equal to the object
        // pointer that called ConvertToType()
        if (obj != tri) deleteIt = TRUE;
        return tri;
    }
    else {
        return NULL;
    }
}

```

## Node Naming

### Prototype:

```
virtual TCHAR* GetName()=0;
```

### Remarks:

Returns the name of the node.

### Prototype:

```
virtual void SetName(TCHAR *s)=0;
```

### Remarks:

Sets the name of the node.

### Parameters:

```
TCHAR *s
```

The name of the node.

### Prototype:

```
virtual ULONG GetHandle();
```

### Remarks:

This method returns the unique node handle. Each node is assigned a unique node handle.

### Return Value:

```
{ return 0; }
```

## Transformation Matrices.

For additional information regarding transformation matrices, see the Advanced Topics section on [Node and Object Offset Transformations](#).

### Prototype:

```
virtual Matrix3 GetNodeTM(TimeValue t, Interval*  
valid=NULL)=0;
```

### Remarks:

This method returns the world space transformation matrix of the node at the

specified time. This matrix contains its parents transformation. This matrix does not include the object-offset transformation, or any world space modifier (Space Warp) affects.

If you select a single node and change the reference coordinate system to 'Local', you'll see the node's axes tripod displayed. This tripod graphically depicts the nodes transformation matrix.

The Node TM may be considered the world space transformation as far as kinematics is concerned. This is almost the complete world space transformation as far as the geometry of the object is concerned, except it does not include the object-offset transformation. See the method **INode::GetObjectTM()** for the complete world space transformation of the geometry of the object.

The Node TM is inherited. When a node asks to retrieve its parents TM, it gets its parents Node TM. It does not get its parents Object TM. The object-offset transformation is not inherited.

**Parameters:**

**TimeValue t**

Specifies the time to retrieve the TM.

**Interval\* valid=NULL**

Points to storage for the validity interval of the transformation matrix. The interval, if passed, is intersected with the validity interval of the NodeTM.

**Return Value:**

The node's world space transform matrix.

**Prototype:**

**virtual void SetNodeTM(TimeValue t, Matrix3& tm)=0;**

**Remarks:**

This methods sets the node's world space transformation matrix (without the object-offset transformation or world space modifier affect). This method will perform the appropriate operation to the node's transform controller. For example, if the node has a parent, this method will take the parent's transformation into consideration when it calls **SetValue()** on the controller.

This method can be used to set the world space position of the node.

**Parameters:**

**TimeValue t**

Specifies the time to set the transformation matrix.

**Matrix3& tm**

The node's world space transformation matrix.

**Prototype:**

```
virtual void Move(TimeValue t, const Matrix3& tmAxis, const Point3& val, BOOL localOrigin=FALSE, BOOL affectKids=TRUE, int pivMode=PIV_NONE, BOOL ignoreLocks=FALSE)=0;
```

**Remarks:**

This method may be called to move the node about the specified axis system. Either the pivot point, or the geometry of the object, or both the pivot and the object may be transformed. Optionally, any children of the node can be counter transformed so they don't move.

**Parameters:****TimeValue t**

The time to transform the node.

**const Matrix3& tmAxis**

The axis system about which the node is transformed.

**const Point3& val**

The amount of the transformation relative to the axis system.

**BOOL localOrigin=FALSE**

If TRUE the transformation takes place about the nodes local origin; otherwise about the world origin.

**BOOL affectKids=TRUE**

If TRUE any child nodes are transformed along with the parent node. If FALSE any children of the node are counter transformed so they don't move.

**int pivMode=PIV\_NONE**

One of the following values:

**PIV\_NONE**

Move both the pivot point and the geometry of the object.

**PIV\_PIVOT\_ONLY**

Move the pivot point only.

## **PIV\_OBJECT\_ONLY**

Move the geometry of the object only.

## **BOOL ignoreLocks=FALSE**

If TRUE any transform locks associated with the node are ignored; otherwise the locks govern the application of the transformation.

## **Prototype:**

```
virtual void Rotate(TimeValue t, const Matrix3& tmAxis, const  
AngAxis& val, BOOL localOrigin=FALSE, BOOL  
affectKids=TRUE, int pivMode=PIV_NONE, BOOL  
ignoreLocks=FALSE)=0;
```

## **Remarks:**

This method may be called to rotate the node about the specified axis system. Either the pivot point, or the geometry of the object, or both the pivot and the object may be transformed. Optionally, any children of the node can be counter transformed so they don't rotate.

## **Parameters:**

### **TimeValue t**

The time to transform the node.

### **const Matrix3& tmAxis**

The axis system about which the node is transformed.

### **const AngAxis& val**

The amount of the transformation.

### **BOOL localOrigin=FALSE**

If TRUE the transformation takes place about the nodes local origin; otherwise about the world origin.

### **BOOL affectKids=TRUE**

If TRUE any child nodes are transformed along with the parent node. If FALSE any children of the node are counter transformed so they don't rotate.

### **int pivMode=PIV\_NONE**

One of the following values:

#### **PIV\_NONE**

Move both the pivot point and the geometry of the object.

**PIV\_PIVOT\_ONLY**

Move the pivot point only.

**PIV\_OBJECT\_ONLY**

Move the geometry of the object only.

**BOOL ignoreLocks=FALSE**

If TRUE any transform locks associated with the node are ignored; otherwise the locks govern the application of the transformation.

**Prototype:**

```
virtual void Rotate(TimeValue t, const Matrix3& tmAxis, const  
Quat& val, BOOL localOrigin=FALSE, BOOL affectKids=TRUE,  
int pivMode=PIV_NONE, BOOL ignoreLocks=FALSE)=0;
```

**Remarks:**

This method may be called to rotate the node about the specified axis system. Either the pivot point, or the geometry of the object, or both the pivot and the object may be transformed. Optionally, any children of the node can be counter transformed so they don't rotate.

**Parameters:****TimeValue t**

The time to transform the node.

**const Matrix3& tmAxis**

The axis system about which the node is transformed.

**const Quat& val**

The amount of the transformation.

**BOOL localOrigin=FALSE**

If TRUE the transformation takes place about the nodes local origin; otherwise about the world origin.

**BOOL affectKids=TRUE**

If TRUE any child nodes are transformed along with the parent node. If FALSE any children of the node are counter transformed so they don't rotate.

**int pivMode=PIV\_NONE**

One of the following values:

**PIV\_NONE**

Move both the pivot point and the geometry of the object.

**PIV\_PIVOT\_ONLY**

Move the pivot point only.

**PIV\_OBJECT\_ONLY**

Move the geometry of the object only.

**BOOL ignoreLocks=FALSE**

If TRUE any transform locks associated with the node are ignored; otherwise the locks govern the application of the transformation.

**Prototype:**

```
virtual void Scale(TimeValue t, const Matrix3& tmAxis, const Point3& val, BOOL localOrigin=FALSE, BOOL affectKids=TRUE, int pivMode=PIV_NONE, BOOL ignoreLocks=FALSE)=0;
```

**Remarks:**

This method may be called to scale the node about the specified axis system. Either the pivot point, or the geometry of the object, or both the pivot and the object may be transformed. Optionally, any children of the node can be counter transformed so they don't scale.

**Parameters:**

**TimeValue t**

The time to transform the node.

**const Matrix3& tmAxis**

The axis system about which the node is transformed.

**const Point3& val**

The amount of the transformation.

**BOOL localOrigin=FALSE**

If TRUE the transformation takes place about the nodes local origin; otherwise about the world origin.

**BOOL affectKids=TRUE**

If TRUE any child nodes are transformed along with the parent node. If FALSE any children of the node are counter transformed so they don't scale.

**int pivMode=PIV\_NONE**

One of the following values:

**PIV\_NONE**

Move both the pivot point and the geometry of the object.

**PIV\_PIVOT\_ONLY**

Move the pivot point only.

**PIV\_OBJECT\_ONLY**

Move the geometry of the object only.

**BOOL ignoreLocks=FALSE**

If TRUE any transform locks associated with the node are ignored; otherwise the locks govern the application of the transformation.

**Prototype:**

```
virtual Matrix3 GetObjectTM(TimeValue time, Interval*  
valid=NULL)=0;
```

**Remarks:**

This method returns the transformation matrix the object needs to be multiplied by to transform it into world space.

At times, this matrix may be the identity. For example, a deformable object that has a Space Warp applied to it would already have been translated into world space by the space warp. Since the object is already in world space the matrix needed to get it there is the identity.

This matrix would **not** be the identity for a deformable object with only object space modifiers applied. This object would indeed need to be transformed. In this case the TM returned would include the NodeTM plus the object-offset transformation. So, **GetObjectTM()** is dependent on the context when it is called -- it will either be equal to **GetObjectTMAfterWSM()** or **GetObjectTMBeforeWSM()**.

Developers should use **GetObjectTMBeforeWSM()** if what is wanted is the object TM and not the identity matrix.

For non-deformable objects this matrix may include the NodeTM, the object-offset transformation and the world space modifier affect.

This matrix could be used, for example, if you have a TriObject and wanted to get the world space coordinate of one of its vertices. You could do this by taking the vertex coordinate in object space and multiplying it by the matrix

returned from this method.

This matrix is also often used inside an object's **Display()** and **HitTest()** methods. When an object goes to draw itself (in its **BaseObject::Display()** method) it is given a node pointer. What the object normally does is use the node pointer and calls **GetObjectTM()**. It then takes the matrix returned and sets it into the graphics window (using **GraphicsWindow::setTransform()**). In this way, when the object starts drawing points in object space, they will be transformed with this matrix. This will transform the points into world space when they are drawn.

The Object TM is not inherited.

**Parameters:**

**TimeValue time**

Specifies the time to retrieve the object's transform matrix.

**Interval\* valid=NULL**

Points to storage for the validity interval of the transformation matrix.

**Return Value:**

The object's transformation matrix.

**Prototype:**

```
virtual Matrix3 GetObjTMBeforeWSM(TimeValue time,  
Interval* valid=NULL)=0;
```

**Remarks:**

This method explicitly retrieves the pipeline ObjectState TM before any world space modifiers have been applied. This includes the node's TM and the object-offset transformation (but not any world space modifier affect). See the section on the [Geometry Pipeline](#) for additional details on this method.

**Parameters:**

**TimeValue time**

Specifies the time to retrieve the transform matrix.

**Interval\* valid=NULL**

Points to storage for the validity interval of the transformation matrix.

**Prototype:**

**virtual Matrix3 GetObjTMAfterWSM(TimeValue time, Interval\* valid=NULL)=0;**

**Remarks:**

This method explicitly retrieves the pipeline ObjectState TM after any world space modifiers have been applied. This includes the Node TM, the object-offset transformation, and any world space modifier affects. In some cases a world space modifier can actually deform the TM itself if it cannot deform the object. Examples of this are cameras and lights. These objects do not have any 'object' to deform, so the space warp deforms the TM instead. See the section on the [Geometry Pipeline](#) for additional details on this method.

Note: Beware of calling this method from inside a function that performs a mesh traversal as doing so can invalidate the mesh.

**Parameters:**

**TimeValue time**

Specifies the time to retrieve the object's transform matrix.

**Interval\* valid=NULL**

Points to storage for the validity interval of the transformation matrix.

**Prototype:**

**virtual Matrix3 GetParentTM(TimeValue t)=0;**

**Remarks:**

Retrieves the parent node's transformation matrix. This is simply for convenience. It is the equivalent to the following code:

**node->GetParentNode()->GetNodeTM();**

**Parameters:**

**TimeValue t**

Specifies the time to retrieve the transformation matrix.

**Return Value:**

The parent node's transformation matrix.

**Prototype:**

**virtual int GetTargetTM(TimeValue t, Matrix3& m)=0;**

**Remarks:**

Retrieves the target node's transformation matrix. This is simply for convenience. It is the equivalent to the following code:

```
node->GetTarget()->GetNodeTM();
```

**Parameters:**

**TimeValue t**

Specifies the time to retrieve the transformation matrix.

**Matrix3& m**

The result is stored here.

**Return Value:**

Nonzero if the target matrix was retrieved (the node had a target); otherwise 0.

**Prototype:**

```
virtual int IsTarget()=0;
```

**Remarks:**

Determines if the node is a target node of a lookat controller.

**Return Value:**

Nonzero if the node is a target; otherwise 0.

**Prototype:**

```
virtual void SetIsTarget(BOOL b)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

This method controls the property of the node indicating if it's a target or not. Calling this is necessary when hooking up targets as the target node must have its **IsTarget()** property set.

**Parameters:**

**BOOL b**

TRUE for set; FALSE for off.

**Prototype:**

```
virtual INode* GetTarget()=0;
```

**Remarks:**

Retrieves this node's target node if any. Returns NULL if this node has no target.

**Prototype:**

**virtual INode\* GetLookatNode()=0;**

**Remarks:**

If this node is a target of a lookat controller, this method finds the node that looks at it.

**Return Value:**

The node that looks at this node or NULL if the node is not a target.

**Prototype:**

**virtual BOOL GetTransformLock(int type, int axis)=0;**

**Remarks:**

Retrieves the specified transform lock state of the node. When the user is doing interactive Moving / Rotating / Scaling these locks simply remove one of the components.

**Parameters:**

**int type**

See [Transform Lock Types](#).

**int axis**

See [Transform Lock Axis](#).

**Return Value:**

TRUE if the lock is set; otherwise FALSE.

**Prototype:**

**virtual void SetTransformLock(int type, int axis, BOOL onOff)=0;**

**Remarks:**

Sets the specified transform lock state of the node. When the user is doing interactive Moving / Rotating / Scaling these locks simply remove one of the components.

**Parameters:**

**int type**

See [Transform Lock Types](#).

**int axis**

See [Transform Lock Axis](#).

**BOOL onOff**

TRUE sets the lock on; FALSE sets the lock off.

## The following methods invalidate the node's caches

**Prototype:**

**virtual void InvalidateTreeTM()=0;**

**Remarks:**

This method will notify the node's subtree that the transformation matrix has changed. **NotifyDependents()** is called with the message **REFMSG\_CHANGE**.

**Prototype:**

**virtual void InvalidateTM()=0;**

**Remarks:**

This method will invalidate the node's transformation matrix cache.

**Prototype:**

**virtual void InvalidateWS()=0;**

**Remarks:**

This method will invalidate the node's world space cache.

## Parent-Child Hierarchy manipulation / Groups

**Prototype:**

**virtual INode\* GetParentNode()=0;**

**Remarks:**

Retrieves the parent node of this node. If the node is not linked, its parent is the root node. This may be checked using **INode::IsRootNode()**.

**Prototype:**

**virtual int IsRootNode()=0;**

**Remarks:**

Determines if this node is the root node (does not have a parent node).

**Return Value:**

Nonzero if the node is the root node; otherwise 0.

**Prototype:**

**virtual void AttachChild(INode\* node, int keepPos=1)=0;**

**Remarks:**

Makes the specified node a child of this node.

**Parameters:**

**INode\* node**

Specifies the node to attach.

**int keepPos=1**

If nonzero, the position of the child node is retained (not moved); otherwise it may be moved.

**Prototype:**

**virtual void Detach(TimeValue t, int keepPos=1)=0;**

**Remarks:**

Detaches this node from its parent.

**Parameters:**

**TimeValue t**

Specifies the time at which to detach the node.

**int keepPos=1**

If nonzero, the position of the detached node is retained (not moved); otherwise it may be moved.

**Prototype:**

**virtual int NumberOfChildren()=0;**

**Remarks:**

Returns the number of children of this node.

**Prototype:**

**virtual INode\* GetChildNode(int i)=0;**

**Remarks:**

Retrieves the 'i-th' child node of this node.

**Parameters:**

**int i**

Specifies the child node to retrieve.

**Prototype:**

**virtual BOOL IsGroupMember()=0;**

**Remarks:**

Returns TRUE if this node is a member of a group; otherwise FALSE.

**Prototype:**

**virtual BOOL IsGroupHead()=0;**

**Remarks:**

Returns TRUE if this node is the head of a group; otherwise FALSE.

**Prototype:**

**virtual BOOL IsOpenGroupMember();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if this node is a member of an open group; otherwise FALSE.

**Default Implementation:**

**{return 0; }**

**Prototype:**

**virtual BOOL IsOpenGroupHead();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if this node is the head of a group and that group is open; otherwise FALSE.

**Default Implementation:**

```
{return 0; }
```

**Prototype:**

```
virtual void SetGroupMember(BOOL b);
```

**Remarks:**

This method is available in release 2.0 and later only.

A node is marked as a group member or not. This method sets this state.

**Parameters:**

**BOOL b**

TRUE to mark the node as a group member; FALSE to indicate it's not in a group.

**Prototype:**

```
virtual void SetGroupHead(BOOL b);
```

**Remarks:**

This method is available in release 2.0 and later only.

A node is marked as the group head or not. This method sets this state.

**Parameters:**

**BOOL b**

TRUE to mark the node as a group head; FALSE to indicate it's not a group head.

**Prototype:**

```
virtual void SetGroupMemberOpen(BOOL b);
```

**Remarks:**

This method is available in release 2.0 and later only.

A node is marked as an open group member or not. This method sets this state.

**Parameters:**

## **BOOL b**

TRUE to mark the node as a open; FALSE to indicate it's not open.

### **Prototype:**

**virtual void SetGroupHeadOpen(BOOL b);**

### **Remarks:**

This method is available in release 2.0 and later only.

A node is marked as being the head of a group and being open or not. This method sets this state.

### **Parameters:**

#### **BOOL b**

TRUE to mark the node as an open group head; FALSE to indicate it's not an open group head.

## **Target / Node Selection**

### **Prototype:**

**virtual void SetTargetNodePair(int onOff);**

### **Remarks:**

This method is available in release 2.0 and later only.

This method is used for hit-testing and selecting node and target as a single unit.

In 3ds max 2.0 and later you can click on the line connecting, say, a camera to its target and drag that around in the viewports. Doing so moves both the camera and its target as a locked pair.

To accomplish this, the camera, light, and tape measure objects (those with two nodes linked by a Look At controller) check for a hit on the object-target line. If they get there (but not at the object or target itself), then they call this method passing TRUE. Then, when a hit is registered, 3ds max checks the value of this variable (by calling **GetTargetNodePair()**), and, if it is TRUE, selects both the target and the node. If it's FALSE, then either the target or the node, but not both, gets selected, as with 3ds max 1.x.

For sample code see **\MAXSDK\SAMPLES\OBJECTS\LIGHT.CPP** or **TAPEHELP.CPP**.

**Parameters:****int onOff**

TRUE for on; FALSE for off.

**Default Implementation:****{ }****Prototype:****virtual int GetTargetNodePair();****Remarks:**

This method is available in release 2.0 and later only.

Returns the target/node pair setting stored by 3ds max. See

**SetTargetNodePair()** above for details.**Default Implementation:****{ return 0; }****Display attributes****Prototype:****virtual void Hide(BOOL onOff)=0;****Remarks:**

Controls the hidden state of the node in the scene.

**Parameters:****BOOL onOff**

Pass TRUE to hide the node in the scene; pass FALSE to make the node visible.

**Prototype:****virtual int IsHidden(DWORD hFlags=0)=0;****Remarks:**

Determines if the node is hidden in the scene.

**Parameters:****DWORD hFlags=0**

If you pass 0, you will get the hidden state of the node. If you pass one or more of the flags shown below, the method checks the Class\_ID of the node to see if it's hidden by the specified category. You may specify one or more of the following values:

**HIDE\_OBJECTS**  
**HIDE\_SHAPES**  
**HIDE\_LIGHTS**  
**HIDE\_CAMERAS**  
**HIDE\_HELPERS**  
**HIDE\_WSMS**  
**HIDE\_SYSTEMS**  
**HIDE\_PARTICLES**  
**HIDE\_ALL**  
**HIDE\_NONE**

See the method **Interface::GetHideByCategoryFlags()** for how you can retrieve the currently set values to use as the flags for this method.

**Return Value:**

Nonzero if the node is hidden; otherwise 0.

**Prototype:**

**virtual int IsNodeHidden()=0;**

**Remarks:**

Returns nonzero if the node is hidden in any way; otherwise returns zero.

**Prototype:**

**virtual int IsFrozen()=0;**

**Remarks:**

Determines if the node is frozen in the scene.

**Return Value:**

Nonzero if the node is frozen; otherwise 0.

**Prototype:**

**virtual void Freeze(BOOL onOff)= 0;**

**Remarks:**

Controls the frozen state of the node in the scene. A frozen node is visible but cannot be picked.

**Parameters:****BOOL onOff**

TRUE if the node should be frozen; FALSE if the node should not be frozen.

**Prototype:**

```
virtual void SetShowFrozenWithMtl(BOOL onOff)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the **NODE\_SHOW\_FRZN\_WITH\_MTL** flag in the node so that the node will be displayed in a frozen state with materials applied.

**Parameters:****BOOL onOff**

TRUE to set the flag; FALSE to disable.

**Prototype:**

```
virtual int ShowFrozenWithMtl()=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the state of the

**NODE\_SHOW\_FRZN\_WITH\_MTL** flag in the node and whether it is enabled or disabled.

**Prototype:**

```
virtual void XRayMtl(BOOL onOff)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

There is a new X-Ray Material display property which allows you to quickly make objects transparent. This method toggles it on or off for this node.

**Parameters:**

**BOOL onOff**

TRUE to use; FALSE to not use.

**Prototype:**

**virtual int HasXRayMtl()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns nonzero if the X-Ray Material display property is on for the node; otherwise zero.

**Prototype:**

**virtual int GetBoxMode()=0;**

**Remarks:**

Determines if the node is displayed in box mode in the scene.

**Return Value:**

Nonzero if the node is displayed in box mode; otherwise 0.

**Prototype:**

**virtual void BoxMode(BOOL onOff)=0;**

**Remarks:**

Controls if the node is displayed with a bounding box representation in the scene.

**Parameters:**

**BOOL onOff**

TRUE to display the node as its bounding box; FALSE for normal display.

**Prototype:**

**virtual void VertTicks(int onOff)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to enable or disable the display of vertex ticks on the

node.

**Parameters:**

**int onOff**

TRUE to enable; FALSE to disable.

**Prototype:**

**virtual int GetVertTicks()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the state of the vertex ticks display. TRUE if enabled; FALSE if disabled.

**Prototype:**

**virtual int GetAllEdges()=0;**

**Remarks:**

Determines if all the edges of the node are displayed.

**Return Value:**

Nonzero if all the edges (including "hidden" ones) are displayed; otherwise 0.

**Prototype:**

**virtual void AllEdges(BOOL onOff)=0;**

**Remarks:**

Controls the display of all the edges of the node (including "hidden" ones).

**Parameters:**

**BOOL onOff**

TRUE to display all the node's edges; FALSE to not display "hidden" edges.

**Prototype:**

**virtual int GetBackCull()=0;**

**Remarks:**

Determines if back-face culling is being used to draw the node.

**Return Value:**

Nonzero if back-face culling is used; otherwise 0.

**Prototype:**

**virtual void BackCull(BOOL onOff)=0;**

**Remarks:**

Controls if the node is displayed using back-face culling (faces whose surface normals are pointing away from the observer are not drawn).

**Parameters:**

**BOOL onOff**

TRUE if the node should be drawn using back-face culling; FALSE if all faces should be drawn.

**Prototype:**

**virtual int GetTrajectoryON();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns nonzero if the trajectory display is on; zero if the trajectory display is off.

**Prototype:**

**virtual void SetTrajectoryON(BOOL onOff);**

**Remarks:**

This method is available in release 2.0 and later only.

This method may be used to turn on or off the trajectory display for the node.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void ShowBone(int boneVis)=0;**

**Remarks:**

Controls the display of Bones in the scene. A bone is just the link (or line) connecting the node to its parent. These are the same options as available in the 3ds max user interface in the Display branch, under Link Display, i.e. Display Links and Link Replaces Object.

**Parameters:**

**int boneVis**

Specifies the display state:

- 0: Bones are not drawn.
- 1: Bones are drawn.
- 2: Only bones are shown.

**Prototype:**

**virtual BOOL IsBoneOnly();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the bone is showing but the object is hidden; FALSE if both the bone and the node is hidden.

**Prototype:**

**virtual void BoneAsLine(int onOff)=0;**

**Remarks:**

Controls the display of Bones as simple lines in the scene.

**Parameters:**

**int onOff**

Nonzero if bones should be shown as lines only; 0 for normal display.

**Prototype:**

**virtual BOOL IsBoneShowing()=0;**

**Remarks:**

Returns TRUE if the node's bone is turned on; otherwise FALSE.

**Prototype:**

**virtual DWORD GetWireColor()=0;**

**Remarks:**

Retrieves the node's wire-frame color. See [COLORREF-DWORD format](#).

**Prototype:**

**virtual void SetWireColor(DWORD newcol)=0;**

**Remarks:**

Sets the node's wire-frame color. This can be any of the 16 million possible colors in 24 bit. See [COLORREF-DWORD format](#).

**Parameters:**

**DWORD newcol**

Specifies the new wire-frame color for the node. It may be specified using the RGB macro, for example: **RGB(0,0,255);**

**Prototype:**

**virtual int Selected()=0;**

**Remarks:**

Determines if the node is selected.

**Return Value:**

Nonzero if the node is selected; otherwise 0.

**Prototype:**

**virtual int Dependent()=0;**

**Remarks:**

Returns nonzero if the node has its dependent flag set; otherwise 0. This is dependent in the sense of 3ds max's Views/Show Dependencies mode. When in the Modify branch, Show Dependencies will show all the nodes that are dependent on the current modifier or object being editing by highlighting them in green. It also set a flag in the node. This method allows a developer to check this flag.

## **Rendering Attributes**

**Prototype:**

**virtual int CastShadows()=0;**

**Remarks:**

Retrieves the shadow casting attribute of the node.

**Return Value:**

Nonzero indicates the node casts shadows; zero if the node does not cast shadows.

**Prototype:**

**virtual void SetCastShadows(BOOL onOff)=0;**

**Remarks:**

Sets the shadow casting attribute of the node to on or off.

**Parameters:**

**BOOL onOff**

TRUE to turn shadow casting on; FALSE to turn it off.

**Prototype:**

**virtual int RcvShadows()=0;**

**Remarks:**

Retrieves the shadow receiving attribute of the node.

**Return Value:**

Nonzero indicates the node receives shadows; zero if the node does not receive shadows.

**Prototype:**

**virtual void SetRcvShadows(BOOL onOff)=0;**

**Remarks:**

Sets the shadow receiving attribute of the node to on or off.

**Parameters:**

**BOOL onOff**

TRUE to turn shadow receiving on; FALSE to turn it off.

**Prototype:****virtual int MotBlur()=0;****Remarks:**

Retrieves the type of motion blur used by the node.

**Return Value:**

One of the following values:

**0:** None.

**1:** Object Motion Blur.

**2:** Image Motion Blur.

**Prototype:****virtual void SetMotBlur(int kind)=0;****Remarks:**

This method is available in release 2.0 and later only.

Sets the type of motion blur used by the node.

**Parameters:****int kind**

The kind of motion blur. One of the following values:

**0:** None.

**1:** Object Motion Blur.

**2:** Image Motion Blur.

**Prototype:****virtual float GetImageBlurMultiplier(TimeValue t);****Remarks:**

This method is available in release 2.0 and later only.

Returns the image motion blur multiplier value at the specified time.

**Parameters:****TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual void SetImageBlurMultiplier(TimeValue t, float m);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the image blur multiplier value for the node. This is used to increase or decrease the length of the blur 'streak'.

**Parameters:**

**TimeValue t**

The time to set the value.

**float m**

The value to set.

**Prototype:**

**virtual Control \*GetImageBlurMultController();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the controller for the image blur multiplier value.

**Prototype:**

**virtual void SetImageBlurMultController(Control \*cont);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the controller used for the image blur multiplier value.

**Parameters:**

**Control \*cont**

Points for the controller to use.

**Prototype:**

**virtual BOOL GetMotBlurOnOff(TimeValue t);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the object motion blur controller is 'on' at the specified time; otherwise FALSE.

**Parameters:**

**TimeValue t**

The time to check.

**Default Implementation:**

```
{ return 1; }
```

**Prototype:**

```
virtual void SetMotBlurOnOff(TimeValue t, BOOL m);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the state of the object motion blur controller to on or off at the specified time.

**Parameters:**

**TimeValue t**

The time to set the value.

**BOOL m**

TRUE for on; FALSE for off.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual Control *GetMotBlurOnOffController();
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the controller handling the object motion blur on / off setting.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

**virtual void SetMotBlurOnOffController(Control \*cont);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the controller used for handling the object motion blur on / off setting.

**Parameters:**

**Control \*cont**

Points to the controller to set.

**Default Implementation:**

**{}**

**Prototype:**

**virtual int Renderable()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns nonzero if the renderable flag is on; zero if off.

**Prototype:**

**virtual void SetRenderable(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the state of the node's renderable flag. If this flag is on the node will appear in rendered images; if off it won't.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetRenderOccluded(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the state of the node's 'Render Occluded Object' flag.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetRenderOccluded()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the node's 'Render Occluded Object' flag is set; otherwise FALSE.

**Prototype:**

**virtual void SetApplyAtmospherics(BOOL onOff)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the "apply atmospherics" flag for the node.

**Parameters:**

**BOOL onOff**

TRUE to enable the flag, FALSE to disable.

**Prototype:**

**virtual int ApplyAtmospherics()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the on/off state of the "apply atmospherics" flag.

**Prototype:**

**virtual void SetPrimaryVisibility(BOOL onOff) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the primary visibility flag and define whether or

not the node is visible to the camera.

**Parameters:**

**BOOL onOff**

TRUE to enable the flag, FALSE to disable.

**Prototype:**

**virtual int GetPrimaryVisibility() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the on/off state of the primary visibility to determine whether or not the node is visible to the camera.

**Prototype:**

**virtual void SetSecondaryVisibility(BOOL onOff) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the secondary visibility flag and define whether or not the node is visible to reflections and refractions..

**Parameters:**

**BOOL onOff**

TRUE to enable the flag, FALSE to disable.

**Prototype:**

**virtual int GetSecondaryVisibility() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the on/off state of the secondary visibility to determine whether or not the node is visible to reflections and refractions.

## **Vertex Color Attributes**

**Prototype:**

**virtual int GetCVertMode();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns nonzero if the vertex color flag is on; otherwise zero.

**Prototype:**

**virtual void SetCVertMode(int onOff);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the vertex color flag to on or off. This controls the display of assigned vertex colors. Vertex colors are assigned in the editable mesh in vertex or face sub-object level. Vertex colors only appear in viewports using Smooth or Smooth + Highlight display modes, regardless of the state of this flag.

**Parameters:**

**int onOff**

Nonzero for on; zero for off.

**Prototype:**

**virtual int GetShadeCVerts();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns nonzero if the vertex color shaded flag is on; zero if off.

**Prototype:**

**virtual void SetShadeCVerts(int onOff);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the vertex color shaded flag. This determines whether the vertex colors appear shaded in the viewport. When this is off, the colors are unshaded, and appear in their pure RGB values, looking a little like self-illuminated materials. When on, the colors appear like any other assigned color in the viewports

**Parameters:****int onOff**

Nonzero of on; zero of off.

**Prototype:****void CopyProperties(INode \*from);****Remarks:**

This method is available in release 4.0 and later only.

Calling this method copies the various display, rendering, bone and other general parameters from the "from" object. This is used in Edit(able) Spline and Edit(able) Patch to copy node settings into new nodes created by the "Detach" operations.

**Parameters:****INode \*frame**

A pointer to the INode to copy the properties from.

**Object Reference****Prototype:****virtual Object\* GetObjectRef()=0;****Remarks:**

Returns the object that this node references unless the node has been bound to a Space Warp. In that case this method will not return the WSM derived object even though the node's object reference points at it. Instead it will return the item that the WSM derived object references. Thus, this method will always return essentially the object space portion of the pipeline. In contrast, see **GetObjOrWSMRef()** below.

See the [Geometry Pipeline](#) section for additional details.

**Prototype:****virtual void SetObjectRef(Object \*o)=0;****Remarks:**

Sets the object that this node references. See the [Geometry Pipeline](#) section for

additional details.

**Parameters:**

**Object \*o**

The object this node will reference.

**Prototype:**

**virtual Object \*GetObjOrWSMRef()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the actual object reference of the node directly. So if this node is bound to a Space Warp this method will return a WSM derived object.

If you want the object space portion of the pipeline see **GetObjectRef()** above.

## **Controller Access**

**Prototype:**

**virtual Control\* GetTMController()=0;**

**Remarks:**

Retrieves the node's transform controller. The standard 3ds max transform controllers each have sub -controllers for position, rotation and scale.

To access the data of the node's transform controller you may use [Class IKeyControl](#). The following code fragment shows an example of how this may be done for a PRS controller.

**Control \*c;**

**c = node->GetTMController()->GetPositionController();**

**IKeyControl \*ikeys = GetKeyControlInterface(c);**

With this controller interface you can use its methods to get information about the keys, for example:

**int num = ikeys->GetNumKeys();**

**Prototype:**

**virtual void SetTMController(Control \*m3cont)=0;**

**Remarks:**

Sets the node's transform controller.

**Parameters:**

**Control \*m3cont**

The Matrix3 transform controller to use.

**Prototype:**

**virtual Control \*GetVisController()=0;**

**Remarks:**

Returns the visibility controller for this node.

**Prototype:**

**virtual void SetVisController(Control \*cont)=0;**

**Remarks:**

Sets the visibility controller for this node.

**Parameters:**

**Control \*cont**

The controller to use for visibility control.

## Visibility Related Methods

**Prototype:**

**virtual float GetVisibility(TimeValue t,Interval \*valid=NULL)=0;**

**Remarks:**

Retrieves the visibility of the node at the time passed and updates the validity interval passed. Values < 0 indicate off while values > 0 indicate on. The node is fully visible (opaque) when 1.0 and fully invisible (transparent) when 0.0.

**Parameters:**

**TimeValue t**

The time to get the visibility value.

**Interval \*valid=NULL**

The validity interval to update based on the validity of the visibility.

**Prototype:**

**virtual void SetVisibility(TimeValue t,float vis)=0;**

**Remarks:**

Sets the visibility of the node to the value passed at the time passed.

**Parameters:****TimeValue t**

The time to set the visibility value.

**float vis**

The visibility of the node to set. This is treated like a boolean value where < 0 means off and > 0 means on.

**Prototype:**

**virtual float GetLocalVisibility(TimeValue t,Interval \*valid=NULL)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the local visibility of the node. The value returned from this method is treated like a boolean value where < 0 means off and > 0 means on.

If a node returns TRUE from **GetInheritVisibility()** then its visibility is determined by this method. If **GetInheritVisibility()** method returns FALSE then the visibility of the node is determined by **GetVisibility()**.

**Parameters:****TimeValue t**

The time to get the local visibility value.

**Interval \*valid=NULL**

The validity interval to update based on the validity of the local visibility controller.

**Prototype:**

**virtual BOOL GetInheritVisibility()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the node's visibility is determined by the visibility of the parent of the node; otherwise returns FALSE.

**Prototype:**

**virtual void SetInheritVisibility(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to set the state of the node's inherit visibility flag.

**Parameters:**

**BOOL onOff**

Pass TRUE to have the node inherit its visibility from its parent; otherwise pass FALSE and the node's visibility will be determined by the node itself (not its parent).

## Renderer Materials

**Prototype:**

**virtual Mtl\* GetMtl()=0;**

**Remarks:**

Returns a pointer to the renderer material for the node. If the value returned is NULL the user has not assigned a material yet. See [Class Mtl](#), [Working with Materials and Textures](#).

**Prototype:**

**virtual void SetMtl(Mtl\* matl)=0;**

**Remarks:**

Sets the renderer material used by the node. If the value set is NULL it indicates a material has not been assigned. In this case, the renderer uses the wireframe color of the node for rendering. See [Class Mtl](#), [Working with Materials and Textures](#).

**Parameters:**

**Mtl\* matl**

The materials used to render the node.

## GraphicsWindow Materials

### Prototype:

**virtual Material\* Mtls()=0;**

### Remarks:

Returns a pointer to the **GraphicsWindow** materials. See **NumMtls()** below for the number of entries in this array.

### Prototype:

**virtual int NumMtls()=0;**

### Remarks:

Returns the number of entries in the array of Materials returned by **Mtls()** above.

## Object-Offset methods

### Prototype:

**virtual void SetObjOffsetPos(Point3 p)=0;**

### Remarks:

Sets the position portion of the object offset from the node. See the Advanced Topics section on [Node and Object Offset Transformations](#) for an overview of the object offset transformation.

### Parameters:

**Point3 p**

Specifies the position portion of the object-offset.

### Prototype:

**virtual Point3 GetObjOffsetPos()=0;**

### Remarks:

Returns the position portion of the object-offset from the node as a Point3. See the Advanced Topics section on [Node and Object Offset Transformations](#) for an overview of the object offset transformation.

**Prototype:**

```
virtual void SetObjOffsetRot(Quat q)=0;
```

**Remarks:**

Sets the rotation portion of the object-offset from the node. See the Advanced Topics section on [Node and Object Offset Transformations](#) for an overview of the object offset transformation.

**Parameters:**

**Quat q**

The rotation offset.

**Prototype:**

```
virtual Quat GetObjOffsetRot()=0;
```

**Remarks:**

Returns the rotation portion of the object-offset from the node. See the Advanced Topics section on [Node and Object Offset Transformations](#) for an overview of the object offset transformation.

**Prototype:**

```
virtual void SetObjOffsetScale(ScaleValue sv)=0;
```

**Remarks:**

Sets the scale portion of the object-offset matrix. See the Advanced Topics section on [Node and Object Offset Transformations](#) for an overview of the object offset transformation.

**Parameters:**

**ScaleValue sv**

The scale portion of the offset. See [Class ScaleValue](#).

**Prototype:**

```
virtual ScaleValue GetObjOffsetScale()=0;
```

**Remarks:**

Returns the scale portion of the object-offset from the node. See [Class ScaleValue](#). See the Advanced Topics section on [Node and Object Offset Transformations](#) for an overview of the object offset transformation.

## Put Nodes into the Foreground Plane

### Prototype:

```
virtual void FlagForeground(TimeValue t,BOOL  
notify=TRUE)=0;
```

### Remarks:

Flags the node to put it in the foreground. For additional information see [Foreground / Background Planes](#).

### Parameters:

#### **TimeValue t**

The time to put the node in the foreground.

#### **BOOL notify=TRUE**

If TRUE, the reference message **REFMSG\_FLAGDEPENDENTS** with **PART\_PUT\_IN\_FG** is sent.

## Active Grid Object

### Prototype:

```
virtual int IsActiveGrid()=0;
```

### Remarks:

Determines if this node is the active grid object.

### Return Value:

Nonzero indicates the node is the active grid object; zero indicates it is not.

## Temporary Storage of Data with Nodes

### Prototype:

```
virtual void SetNodeLong(LONG l)=0;
```

### Remarks:

This method provides temporary storage of data with the node. Data stored with the node is only valid before you return control.

### Parameters:

## **LONG l**

The data to store with the node.

### **Prototype:**

**virtual LONG GetNodeLong()=0;**

### **Remarks:**

Retrieve the data stored with the node (using **SetNodeLong()**).

### **Return Value:**

The data stored with the node.

## **Access to Render Data**

### **Prototype:**

**virtual RenderData \*GetRenderData()=0;**

### **Remarks:**

Returns the render data for the node. See [Class RenderData](#).

### **Prototype:**

**virtual void SetRenderData(RenderData \*rd)=0;**

### **Remarks:**

Sets the render data for the node.

### **Parameters:**

**RenderData \*rd**

The render data to set.

## **Access user defined property text.**

For additional overview information on these methods see the Advanced Topics section [Custom Node Properties](#).

### **Prototype:**

**virtual void GetUserPropBuffer(TSTR &buf)=0;**

### **Remarks:**

This method allows access to the entire user defined property text buffer.

**Parameters:**

**TSTR &buf**

The buffer to hold the user defined property text.

**Prototype:**

**virtual void SetUserPropBuffer(const TSTR &buf)=0;**

**Remarks:**

This method allows a developer to set to the entire user defined property text buffer.

**Parameters:**

**const TSTR &buf**

The buffer containing the user defined property text.

**Prototype:**

**virtual BOOL GetUserPropString(const TSTR &key,TSTR &string)=0;**

**Remarks:**

This method retrieves a string based on the key passed.

**Parameters:**

**const TSTR &key**

The key (or name) of the user defined property text.

**TSTR &string**

Storage for the string to retrieve.

**Return Value:**

TRUE if the key was found; otherwise FALSE.

**Prototype:**

**virtual BOOL GetUserPropInt(const TSTR &key,int &val)=0;**

**Remarks:**

Retrieves an integer value from the node based on the key passed.

**Parameters:**

**const TSTR &key**

The key (or name) of the data to retrieve.

**int &val**

Storage for the integer value.

**Return Value:**

TRUE if the key was found; otherwise FALSE.

**Prototype:**

**virtual BOOL GetUserPropFloat(const TSTR &key,float &val)=0;**

**Remarks:**

Retrieves a floating point value from the node based on the key passed.

**Parameters:**

**const TSTR &key**

The key (or name) of the data to retrieve.

**float &val**

Storage for the float value.

**Return Value:**

TRUE if the key was found; otherwise FALSE.

**Prototype:**

**virtual BOOL GetUserPropBool(const TSTR &key,BOOL &b)=0;**

**Remarks:**

Retrieves a boolean value from the node based on the key passed.

**Parameters:**

**const TSTR &key**

The key (or name) of the data to retrieve.

**BOOL &b**

Storage for the boolean value.

**Return Value:**

TRUE if the key was found; otherwise FALSE.

**Prototype:**

**virtual void SetUserPropString(const TSTR &key,const TSTR &string)=0;**

**Remarks:**

Stores a string in the node using the key passed. If the key name already exists it is overwritten; otherwise it is created.

**Parameters:**

**const TSTR &key**

The key (or name) of the data to store.

**const TSTR &string**

The string to store.

**Prototype:**

**virtual void SetUserPropInt(const TSTR &key,int val)=0;**

**Remarks:**

Stores an integer value in the node using the key passed. If the key name already exists it is overwritten; otherwise it is created.

**Parameters:**

**const TSTR &key**

The key (or name) of the data to store.

**int val**

The value to store.

**Prototype:**

**virtual void SetUserPropFloat(const TSTR &key,float val)=0;**

**Remarks:**

Stores a floating point value in the node using the key passed. If the key name already exists it is overwritten; otherwise it is created.

**Parameters:**

**const TSTR &key**

The key (or name) of the data to store.

**float val**

The value to store.

**Prototype:**

**virtual void SetUserPropBool(const TSTR &key,BOOL b)=0;**

**Remarks:**

Stores a boolean value in the node using the key passed. If the key name already exists it is overwritten; otherwise it is created.

**Parameters:**

**const TSTR &key**

The key (or name) of the data to store.

**BOOL b**

The value to store.

**Prototype:**

**virtual BOOL UserPropExists(const TSTR &key)=0;**

**Remarks:**

This method simply checks to see if a key exists.

**Parameters:**

**const TSTR &key**

The key string to search for.

**Return Value:**

TRUE if the key was found; otherwise FALSE.

## **Geometry/Graphics (G) Buffer ID Access**

**Prototype:**

**virtual ULONG GetGBufID()=0;**

**Remarks:**

Returns the G-Buffer ID of this node. This is the ID available in the **BMM\_CHAN\_NODE\_ID** channel. See [Working with Bitmaps \(G-Buffer\)](#) for additional details.

**Prototype:**

**virtual void SetGBufID(ULONG id)=0;**

**Remarks:**

Sets the G-Buffer ID of this node. This is the ID available in the **BMM\_CHAN\_NODE\_ID** channel. See [Working with Bitmaps \(G-Buffer\)](#) for additional details.

**Parameters:**

**ULONG id**

The G-Buffer ID.

**Prototype:**

**virtual UWORD GetRenderID();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the G-Buffer render ID of the node. This is set by the renderer during a video post render when the **BMM\_CHAN\_NODE\_RENDER\_ID** is requested.

**Prototype:**

**virtual void SetRenderID(UWORD id);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the G-Buffer render ID of the node. This is set by the renderer during a video post render when the **BMM\_CHAN\_NODE\_RENDER\_ID** is requested.

**IK Related Methods**

The following methods deal with IK parameters associated with a node.

**Prototype:**

**virtual float GetPosTaskWeight();**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the position weight for the node.

**Prototype:**

**virtual float GetRotTaskWeight();**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the rotation weight for the node.

**Prototype:**

**virtual void SetPosTaskWeight(float w);**

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the position weight for the node.

**Parameters:**

**float w**

The position weight for the node. This value is  $\geq 0.0$ .

**Prototype:**

**virtual void SetRotTaskWeight(float w);**

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the rotation weight for the node.

**Parameters:**

**float w**

The rotation weight for the node. This value is  $\geq 0.0$ .

**Prototype:**

**virtual BOOL GetTaskAxisState(int which,int axis);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE of FALSE to indicate if the specified axis is set for position or rotation.

**Parameters:**

**int which**

Indicates if the method returns the position state or the rotation state: **0** specifies position; **1** specifies rotation.

**int axis**

The axis to check. **0** specifies X, **1** specifies Y, **2** specifies Z.

**Prototype:**

```
virtual void SetTaskAxisState(int which,int axis,BOOL onOff);
```

**Remarks:**

This method is available in release 2.0 and later only.

Sets the specified axis state for position or rotation.

**Parameters:****int which**

Indicates if the method returns the position state or the rotation state: **0** specifies position; **1** specifies rotation.

**int axis**

The axis to check. **0** specifies X, **1** specifies Y, **2** specifies Z.

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

```
virtual DWORD GetTaskAxisStateBits();
```

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the same information as GetTaskAxisState() above.

**Return Value:**

The first three bits indicate position X, Y and Z. Then the next three bits indicate rotation X, Y, Z.

**WSMDerivedObject Access****Prototype:**

**virtual void CreateWSMDerivedObject();**

**Remarks:**

This method is available in release 2.0 and later only.

Calling this method will create a WSM derived object for this node if one doesn't already exist.

**Default Implementation:**

```
{}
```

**Prototype:**

**virtual IDerivedObject \*GetWSMDerivedObject();**

**Remarks:**

This method is available in release 2.0 and later only.

This method pointer to the WSM Derived object for this node. Note that there is at most one WSM derived object per node.

**Default Implementation:**

```
{return NULL;}
```

## Deleting The Node

**Prototype:**

**virtual void Delete(TimeValue t, int keepChildPosition);**

**Remarks:**

This method is available in release 2.0 and later only.

This method will delete the node, handle removing it from the hierarchy, and handle undo.

**Parameters:**

**TimeValue t**

The time for the deletion.

**int keepChildPosition**

If TRUE the position of any children of this node are kept the same; otherwise linked children may move due to the deletion.

## **INodeTransformed methods**

### **Prototype:**

**virtual void DisposeTemporary();**

### **Remarks:**

If this was a temporary INode (like an **INodeTransformed**) then this method will delete it. Also see the method **Interface::GetModContexts()**.

### **Prototype:**

**virtual INode \*GetActualINode();**

### **Remarks:**

In the case of **INodeTransformed**, this method retrieves a pointer to the real node.

### **Return Value:**

A pointer to the node.

## **XRef Access**

Scene XRef related methods. These methods are only implemented by root nodes. Note that Scene XRefs are stored as complete scenes with root nodes where the XRef scene root node is a child of the current scene's root node. See Also: [Class IXRefObject](#).

### **Prototype:**

**virtual int GetXRefFileCount();**

### **Remarks:**

This method is available in release 3.0 and later only.  
Returns the number of scene xrefs.

### **Default Implementation:**

**{return 0;}**

### **Prototype:**

**virtual TSTR GetXRefFileName(int i);**

### **Remarks:**

This method is available in release 3.0 and later only.  
Returns the name of the scene xref whose index is passed.

**Parameters:**

**int i**

The zero based index of the scene xref (**0** to **GetXRefFileCount()-1**).

**Default Implementation:**

```
{return TSTR();}
```

**Prototype:**

```
virtual void SetXRefFileName(int i,TCHAR *fname,BOOL  
reload);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the name of the scene xref whose index is passed.

**Parameters:**

**int i**

The zero based index of the scene xref (**0** to **GetXRefFileCount()-1**).

**TCHAR \*fname**

The name to set.

**BOOL reload**

TRUE to reload; FALSE to not reload.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL AddNewXRefFile(TSTR &name, BOOL  
loadNow=TRUE);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Adds the specified file to the scene and optionally updates the scene now.

**Parameters:**

**TSTR &name**

The file name to load.

**BOOL loadNow=TRUE**

If TRUE the file is loaded immediately and the scene updated; if FALSE the scene is now updated until the user requests it.

**Return Value:**

TRUE if the XRef was loaded; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL DeleteXRefFile(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method removes the 'i-th' Scene XRef.

**Parameters:**

**int i**

The zero based index of the Scene XRef to load.

**Return Value:**

TRUE if the file was deleted from the scene; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL BindXRefFile(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

This methods binds the specified XRef. This deletes the XRef after merging it into the scene.

**Parameters:**

**int i**

The zero based index of the XRef to bind.

**Return Value:**

TRUE if the file was deleted; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void DeleteAllXRefs();
```

**Remarks:**

This method is available in release 3.0 and later only.

This method deletes all the XRefs from the scene. This is called when loading a new file, resetting or clearing the scene.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL ReloadXRef(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method will reload (updates from disk) the specified XRef.

**Parameters:**

**int i**

The zero based index of the XRef to reload.

**Return Value:**

TRUE if the XRef was reloaded; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void FlagXrefChanged(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method indicates that the specified XRef has been changed and should be updated.

**Parameters:**

**int i**

The zero based index of the XRef to flag.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL UpdateChangedXRefs(BOOL redraw=TRUE);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method updates all XRefs which have their changed flag set.

**Parameters:**

**BOOL redraw=TRUE**

TRUE to redraw the scene; otherwise FALSE.

**Return Value:**

TRUE if the XRefs were loaded okay; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual INode *GetXRefTree(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the root node of the tree for the specified XRef. This method, when called on a root node, will access the various XRef scenes. Note that these can be nested so calling this on the root node of the scene will return the root node of one of the scene XRefs. Calling it on the root node of the scene XRef will get the root node of a nested XRef and so on. Note that this is not the parent of the XRef (see **GetXRefParent()** below).

**Parameters:**

**int i**

The zero based index of the XRef.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual INode *GetXRefParent(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the parent node of the specified XRef. This is the node in the scene (if any) which the scene XRef is linked to through the special bind function in the scene XRef dialog.

**Parameters:**

**int i**

The zero based index of the XRef.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual void SetXRefParent(int i, INode *par);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the parent of the specified XRef to the node passed.

**Parameters:**

**int i**

The zero based index of the XRef.

**INode \*par**

The parent node to set.

**Default Implementation:**

```
{}
```

**Prototype:**

**virtual BOOL FindUnresolvedXRefs(Tab<TSTR\*> &fnames);**

**Remarks:**

This method is available in release 3.0 and later only.

Generates a table of names for the unresolved XRefs in the scene.

**Parameters:**

**Tab<TSTR\*> &fnames**

The table of names. See [Template Class Tab](#).

**Return Value:**

Returns TRUE if there are still unresolved XRefs; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

**virtual void AttemptToResolveUnresolvedXRefs();**

**Remarks:**

This method is available in release 3.0 and later only.

This method tries to load any XRefs that are currently unresolved.

**Default Implementation:**

```
{}
```

**Prototype:**

**virtual DWORD GetXRefFlags(int i);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the state of flags for the specified XRef.

**Parameters:**

**int i**

The zero based index of the XRef whose flags are returned.

**Return Value:**

See [List of XRef Flag Bits](#).

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual void SetXRefFlags(int i,DWORD flag,BOOL onOff);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the state of the specified flags in the specified XRef to on or off.

**Parameters:****int i**

The zero based index of the XRef whose flags are set.

**DWORD flag**

See [List of XRef Flag Bits](#).

**BOOL onOff**

TRUE for on; FALSE for off.

**Default Implementation:**

```
{}
```

**Bone Methods****Prototype:**

```
virtual void SetBoneNodeOnOff(BOOL onOff, TimeValue t);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the bone on/off property of the node.

**Parameters:****BOOL onOff**

Pass TRUE for on; FALSE for off.

**TimeValue t**

The time at which to set the property.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void SetBoneAutoAlign(BOOL onOff);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the bone auto-align property of the node.

**Parameters:**

**BOOL onOff**

Pass TRUE for on; FALSE for off.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void SetBoneFreezeLen(BOOL onOff);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the bone freeze length property of the node.

**Parameters:**

**BOOL onOff**

Pass TRUE for on; FALSE for off.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void SetBoneScaleType(int which);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the bone scale (stretch) type to the specified value.

**Parameters:**

**int which**

One of the following values:

**BONE\_SCALETYPE\_SCALE**

**BONE\_SCALETYPE\_SQUASH**

**BONE\_SCALETYPE\_NONE**

**Default Implementation:**

**{}**

**Prototype:**

**virtual void SetBoneAxis(int which);**

**Remarks:**

This method is available in release 4.0 and later only.

Sets the bone axis of the node to the specified value.

**Parameters:**

**int which**

One of the following values:

**BONE\_AXIS\_X**

**BONE\_AXIS\_Y**

**BONE\_AXIS\_Z**

**Default Implementation:**

**{}**

**Prototype:**

**virtual void SetBoneAxisFlip(BOOL onOff);**

**Remarks:**

This method is available in release 4.0 and later only.

Sets the state of the bone axis flip toggle.

**Parameters:**

**BOOL onOff**

Pass TRUE for on; FALSE for off.

**Default Implementation:**

```
}
```

**Prototype:**

```
virtual BOOL GetBoneNodeOnOff();
```

**Remarks:**

This method is available in release 4.0 and later only.  
Returns TRUE if the bone property is on; FALSE if off.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL GetBoneAutoAlign();
```

**Remarks:**

This method is available in release 4.0 and later only.  
Returns TRUE if the bone auto-align property is on; FALSE if off.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL GetBoneFreezeLen();
```

**Remarks:**

This method is available in release 4.0 and later only.  
Returns TRUE if the freeze length property is on; FALSE if off.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual int GetBoneScaleType();
```

**Remarks:**

This method is available in release 4.0 and later only.  
Returns a value which indicates the bone scale type.

**Return Value:**

One of the following values:

**BONE\_SCALETYPE\_SCALE**  
**BONE\_SCALETYPE\_SQUASH**  
**BONE\_SCALETYPE\_NONE**

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual int GetBoneAxis();
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns a value which indicates the bone axis.

**Return Value:**

One of the following values:

**BONE\_AXIS\_X**  
**BONE\_AXIS\_Y**  
**BONE\_AXIS\_Z**

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual BOOL GetBoneAxisFlip();
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns TRUE if the axis flip toggle is on; FALSE if off.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void RealignBoneToChild(TimeValue t);
```

**Remarks:**

This method is available in release 4.0 and later only.

Calling this method is the equivalent of pressing the Realign button in the UI.

**Parameters:****TimeValue t**

The time at which to reset the initial child position.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void ResetBoneStretch(TimeValue t) {};
```

**Remarks:**

This method is available in release 4.0 and later only.

Calling this method is the equivalent of pressing the Reset Stretch button in the UI. This will cause the X-axis of the bone to realign to point at the child bone (or average pivot of multiple children).

**Parameters:****TimeValue t**

The time at which to reset the bone stretch.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual Matrix3 GetStretchTM(TimeValue t, Interval  
*valid=NULL);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the stretchTM without the object offset included.

Normally matrix concatenation occurs in the following manner:

**objectTM = objectOffsetTM \* stretchTM \* nodeTM**

A plug-in that uses the objectTM will transparently inherit the effects of the stretchTM. However if a plug-in (such as skin) wants the stretchTM included

but not the object offset, this method will return the stretchTM alone. If the node is not a bone or has no stretching, this method will return the identity.

**Parameters:**

**TimeValue t**

The time at which to obtain the stretchTM.

**Interval \*valid = NULL**

The interval.

**Default Implementation:**

```
{return Matrix3(1);}
```

# LightObject

See Also: [Class LightObject](#).

This is one of the classes from which plug-in lights are derived. Light objects implement methods that return the properties of the light. Some are these methods are:

## **EvalLightState()**

This method provides access to the lights state which contains properties of the light such as color, intensity, etc.

## **CreateLightDesc()**

When the renderer goes to render the scene it asks all of the lights to create an ObjectLighDesc object. This is the method that is called to return this object.

# LinearShape

See Also: [Class LinearShape](#).

Defines a linear shape object class. This is a shape made up of entirely linear segments.

## Class LinearShape

See Also: [Class ShapeObject](#), [Class PolyShape](#), [Working with Shapes and Splines](#).

```
class LinearShape : public ShapeObject
```

### Description:

This class represents a linear shape object. This class is similar to a **SplineShape** except this class uses a **PolyShape** as its data while a **SplineShape** uses a **BezierShape** as its data. Therefore this is a shape made up of entirely linear segments. All methods of this class are implemented by the system.

### Data Members:

public:

**PolyShape shape;**

The PolyShape that holds the linear shape.

### Methods:

#### Prototype:

**LinearShape();**

#### Remarks:

Constructor.

#### Prototype:

**~LinearShape();**

#### Remarks:

Destructor.

#### Prototype:

**PolyShape& GetShape();**

#### Remarks:

Returns the **shape** data member.

**Prototype:**

**void SetPointFlags();**

**Remarks:**

This method does the job of setting all points in the PolyShape to **POLYPT\_KNOT** types, and removing the **POLYPT\_INTERPOLATED** flag. This is because the LinearShape knows nothing about its origin.

**Prototype:**

**virtual MtlID GetMatID(TimeValue t, int curve, int piece);**

**Remarks:**

This method is available in release 3.0 and later only.

This method provides access to the material IDs of the shape. It returns the material ID of the specified segment of the specified curve of this shape at the time passed. There is a default implementation so there is no need to implement this method if the shape does not support material IDs.

Note: **typedef unsigned short MtlID;**

**Parameters:**

**TimeValue t**

The time to evaluate the sub-curve.

**int curve**

The zero based index of the curve to evaluate.

**int piece**

The sub-curve (segment) to evaluate.

# Modifier

See Also: [Class Modifier](#).

This is the class from which you may derive modifier object plug-ins. The main methods are:

## **ModifyObject()**

This is the method that actually modifies the input object. This is where the modifier does its work.

## **LocalValidity()**

This is the validity interval of the modifier. It is the intersection of the validity intervals of any controllers the modifier has that are animating the modifier's parameters.

Methods are available to indicate to the system what type of input it needs from the pipeline.

## **ChannelsUsed()**

These are channels that the modifier needs in order to perform its modification.

## **ChannelsChanged()**

These are the channels that the modifier actually modifies.

## **InputType()**

This is the type of object that the modifier knows how to modify. Simple modifiers that just modify points of an object can operate on generic 'Deformable' objects. Deformable objects are any type of object that has points. A modifier could also work on a particular type of object such as a tri object or patch object.

Methods are also available for saving the data a modifier uses and stores internally to manage its operation.

## **LoadLocalData()**

## **SaveLocalData()**

When a 3ds max file is being loaded or saved, these methods are called so that the modifier can load and save data it needs for its operation.

# Mtl

See Also: [Class Mtl](#).

Plug-In Materials are subclassed off Mtl. Some of the methods of Mtl are:

## **Shade()**

This is called by the renderer to compute the color and transparency of the material for a given location.

## **NumSubMtls()**

This method returns the number of sub-materials used by the material.

# MtlBase

See Also: [Class MtlBase](#).

This is the base class for the creation of materials and texture maps plug-ins. The main methods are:

## **Requirements()**

Specifies the requirements for the material or texture map (and its sub-materials or sub-textures if any). These are things such as whether the material is two sided, wire frame, or requires UV bump vectors.

## **CreateParamDlg()**

This gets called when the material or texture is to be displayed in the material editor parameters area.

## Class MtlBase

See Also: [Class ReferenceTarget](#), [Class ISubMap](#), [Class Mtl](#), [Class Texmap](#), [Class ShadeContext](#), [Class RenderMapsContext](#), [Class RenderGlobalContext](#), [Class UVGen](#), [Class XYZGen](#), [Class PStamp](#), [Class Quat](#).

```
class MtlBase : public ReferenceTarget, public ISubMap
```

### Description:

This is the base class from which materials and textures are subclassed. Methods are provided to access the name, flags, and sub-materials/maps. There is also a method that is called when the material or texture is to be displayed in the material editor parameters area.

Note the following about dialog proc processing of sub-map buttons:

When you post the message:

```
PostMessage(hwmedit, WM_TEXMAP_BUTTON, i,  
(LPARAM)theMtl);
```

You are telling the system that the user clicked on the button for the 'i-th' sub-map of **theMtl**. The message doesn't propagate up -- it goes directly to the materials editor. It then uses calls on **theMtl->SetSubTexmap()** and **theMtl->GetSubTexmap()** to assign the new map. So even if your material has some complicated internal hierarchical structure of references, to the system it is still a simple "logical" hierarchy of a material with some sub-textmaps.

### Data Members:

public:

```
Quat meditRotate;
```

This data member is available in release 2.0 and later only.

This describes the rotation of the sample geometry in the materials editor.

```
ULONG gbufID;
```

This is the G-Buffer ID of the material or texmap. This is a "effects number" assigned in the materials editor to a map or material, and it will be written into the effects channel of the G-Buffer when a pixel with that material on it is rendered. To implement this, each map or material, in the beginning of its **Shade()**, **EvalColor()**, or **EvalMono()** methods should have the code:

**if (gbufID) sc.SetGBufferID(gbufID);**

**Method Groups:**

The following hyperlinks take you to the start of groups of related methods within the class:

[Naming Methods](#)

[BuildMaps Method](#)

[Flag Access Methods](#)

[Requirement Methods](#)

[Sub-Mtl-Texture Access](#)

[Update / Reset / Validity](#)

[Loading / Saving Methods](#)

[User Interface Methods](#)

[Multiple Map Display in the Viewports](#)

[G Buffer Methods](#)

[Enumerate Auxilliary Files Implementation](#)

[Postage Stamp Image Methods](#)

[Internal Methods](#)

[Operators](#)

**Methods:**

**Prototype:**

**MtlBase();**

**Remarks:**

Constructor. The flags and G-buffer id are initialized.

**Naming Methods**

**Prototype:**

**TSTR& GetName()**

**Remarks:**

Implemented by the System.

Returns the name of the material or texture.

**Prototype:**

**void SetName(TSTR s);**

**Remarks:**

Implemented by the System.  
Stores the name of the material or texture.

**Prototype:**

**virtual TSTR GetFullName();**

**Remarks:**

Implemented by the System.  
This method returns the name that appears in the track view. It returns the "Instance Name(class Name)". For example "Green Glass (Standard)". The default implementation should be used.

## **Flag Access Methods**

**Prototype:**

**void SetMtlFlag(int mask, BOOL val=TRUE)**

**Remarks:**

Implemented by the System.  
Alters the flags, either setting or clearing them, using the mask and method passed.

**Parameters:**

**int mask**

The flags to alter. See [List of Material Flags](#).

**BOOL val=TRUE**

If TRUE the mask is ORed into the flags (**mtlFlags |= mask**); otherwise (**mtlFlags &= ~mask**).

**Prototype:**

**void ClearMtlFlag(int mask)**

**Remarks:**

Implemented by the System.

Clears the specified flags.

**Parameters:**

**int mask**

The flags to clear. See [List of Material Flags](#).

**Prototype:**

**int TestMtlFlag(int mask)**

**Remarks:**

Implemented by the System.

Tests the specified flags. Returns nonzero if the flags are set; otherwise zero.

See [List of Material Flags](#).

**Parameters:**

**int mask**

The flags to test.

**Prototype:**

**BOOL AnyMulti();**

**Remarks:**

Implemented by the System.

This method may be called to recursively determine if there are any multi-materials or texmaps in the tree.

**Return Value:**

TRUE if the material or texture map has any multi-materials; otherwise FALSE.

## Requirement Methods

**Prototype:**

**virtual ULONG Requirements(int subMtlNum);**

**Remarks:**

Implemented by the Plug-In.

This method returns the cumulative requirements of the material and its sub-

materials and maps. The default implementation just ORs together the local requirements of the material with the requirements of all its children. Most materials will only need to implement **LocalRequirements()**.

**Parameters:**

**int subMtlNum**

Specifies the number of the sub-material whose requirements should be returned. **-1** may be used to return a value generated by looping over all the sub-materials and ORing together the requirements.

**Return Value:**

See [List of Material Requirements](#).

**Default Implementation:**

The default implementation automatically traverses the sub-material/map tree. On any MtlBase that returns TRUE for **IsMultiMtl()** it will only recursively call the sub material (or map) for the **subMtlNum** called. The exception to this is when **subMtlNum<0**: in this case all sub-mtls and submaps are enumerated. Therefore the **LocalRequirements()** method below only needs to consider the material or map itself, not the sub-mtls and sub-maps.

**Prototype:**

**virtual ULONG LocalRequirements(int subMtlNum)**

**Remarks:**

Implemented by the Plug-In.

Specifies various requirements for **this** material. The value returned from this method should **not** include requirements of its sub-materials and sub-maps.

**Parameters:**

**int subMtlNum**

Specifies the number of the sub-material whose requirements should be returned.

**Return Value:**

See [List of Material Requirements](#).

**Default Implementation:**

**{ return 0; }**

**Prototype:**

**virtual void MappingsRequired(int subMtlNum, BitArray &mapreq, BitArray &bumpreq);**

**Remarks:**

This method is available in release 3.0 and later only.

This method gives the UVW channel requirements of the material and its tree. The default implementation just OR's together the local mapping requirements of the material with the requirements of all its children. For most materials, all they need to implement is the **LocalMappingsRequired()** method.

**Parameters:****int subMtlNum**

Specifies the number of the sub-material whose mapping information is retrieved.

**BitArray &mapreq**

This array of bits is initialized to an empty set with MAX\_MESHMAPS elements. Each bit corresponds to a mapping channel. Set a bit to one to indicate the material requires the corresponding UVW channel.

**BitArray &bumpreq**

This array of bits is initialized to an empty set with MAX\_MESHMAPS elements. Each bit corresponds to a mapping channel. Set a bit to one to indicate the material requires the corresponding bump mapping channel.

**Prototype:**

**virtual void LocalMappingsRequired(int subMtlNum, BitArray &mapreq, BitArray &bumpreq);**

**Remarks:**

This method is available in release 3.0 and later only.

This method specifies UVW channel requirements for the material: This method should **not** include UVW channel requirements of any sub-materials and sub-maps.

**Parameters:****int subMtlNum**

Specifies the number of the sub-material whose mapping information is

retrieved.

### **BitArray &mapreq**

This array of bits is initialized to an empty set with MAX\_MESHMAPS elements. Each bit corresponds to a mapping channel. Set a bit to one to indicate the material requires the corresponding UVW channel.

### **BitArray &bumpreq**

This array of bits is initialized to an empty set with MAX\_MESHMAPS elements. Each bit corresponds to a mapping channel. Set a bit to one to indicate the material requires the corresponding bump mapping channel.

### **Default Implementation:**

```
{}
```

### **Sample Code:**

All 2D textures that use UVGen or otherwise select mapping channels need to implement this method Here's an example:

```
void LocalMappingsRequired(int subMtlNum, BitArray &  
mapreq, BitArray &bumpreq) {  
    uvGen->MappingsRequired(subMtlNum,mapreq,bumpreq);  
}
```

All 3D textures that use the XYZGen to put up a coordinates rollup must implement this method. Here's an example:

```
void LocalMappingsRequired(int subMtlNum, BitArray &  
mapreq, BitArray &bumpreq) {  
    xyzGen->MappingsRequired(subMtlNum,mapreq,bumpreq);  
}
```

## **BuildMaps Method**

### **Prototype:**

```
virtual int BuildMaps(TimeValue t, RenderMapsContext &rmc)
```

### **Remarks:**

Implemented by the Plug-In.

This method is called for a plug-in to do any work it needs to do prior to rendering. For example this is used by the 3ds max Mirror and Auto Reflect

materials to render their bitmaps.

**Parameters:**

**TimeValue t**

The current time.

**RenderMapsContext &rmc**

Provides information about the view being rendered and can provide access to the global rendering environment information via **RenderGlobalContext**

**\*gc = rmc.GetGlobalContext()**. See [Class RenderMapsContext](#) and [Class RenderGlobalContext](#).

**Return Value:**

Nonzero on success; zero on failure. In the case of failure the renderer is halted (rendering is cancelled).

## Methods to access sub texture maps of materials or texmaps

**Prototype:**

**virtual BOOL IsMultiMtl()**

**Remarks:**

Implemented by the Plug-In.

This method returns TRUE for materials or textures that select sub-materials based on submaterial number (for example a mesh **faceMtlIndex**).

The only materials for which this method should return TRUE are those that choose to use one of their sub-maps or sub-mtls based on the submaterial number. For the majority of materials and maps, they should return FALSE, and in that case all of the submaterials and maps are enumerated by

**MtlBase::Requirements()**.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**void DeactivateMapsInTree();**

**Remarks:**

Implemented by the System.

This method must be called on a sub-material or sub-map when it is removed, in case it or any of its sub-maps are active in the viewport.

## **Update / Reset / Validity**

### **Prototype:**

**virtual void Update(TimeValue t, Interval& valid)=0;**

### **Remarks:**

Implemented by the Plug-In.

A material has a **Shade()** method, and a texture map has a **EvalColor()** method. These are called by the renderer for every pixel. This method is called before rendering begins to allow the plug-in to evaluate anything prior to the render so it can store this information. In this way this information does not need to be calculated over and over for every pixel when **Shade()** or **EvalColor()** is called. This allows texture and material evaluation to be more efficient. It is generally called once at the start of each frame, or during interactive playback when the time changes. It is **not** guaranteed to get called on every frame, because if you returned a validity interval that covers several frames, your parent material won't bother to call you if the current frame is still in that interval.

### **Parameters:**

#### **TimeValue t**

The current time.

#### **Interval& valid**

The validity interval to update to reflect the validity interval of the material or texture at the time passed.

### **Prototype:**

**virtual void Reset()=0;**

### **Remarks:**

Implemented by the Plug-In.

This method is called to reset the material or texmap back to its default values.

### **Prototype:**

**virtual Interval Validity(TimeValue t)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to determine the validity interval of the material or texture.

**Parameters:**

**TimeValue t**

Specifies the time about which to compute the validity interval.

**Return Value:**

The validity interval of the item at the specified time.

## User Interface Methods

**Prototype:**

**virtual ParamDlg\* CreateParamDlg(HWND hwMtlEdit,  
IMtlParams \*imp)=0;**

**Remarks:**

Implemented by the Plug-In.

This method gets called when the material or texture is to be displayed in the material editor parameters area. The plug-in should allocate a new instance of a class derived from **ParamDlg** to manage the user interface.

**Note:** The following is a discussion of this **CreateParamDlg()** method, and the **SetDlgThing()** method, and the way they relate to the ParamMap2 system. A good example for this discussion is a texture map which typically has several rollouts, say its texmap parameters, a UVW coordinates rollout, and an Output rollout.

The normal way for a texmap to implement these other (common) rollouts is for the it to create a UVGen instance and a TextureOutput instance as 'sub-objects' in the map and then ask them to put up their rollouts when it is asked to do so by the Materials Editor (in CreateParamDlg()). The Materials Editor requires a ParamDlg pointer back from the CreateParamDlg() on which it calls methods to control the UI, such as time change updates or loading up a new texmap of the same class into the UI when the user switches them, so that the whole UI doesn't need to be rebuilt.

Prior to the ParamMap2 system, a texmap would implement its own ParamDlg subclass and would keep track of the UVGen and TextureOutput ParamDialogs and pass on any time change or SetThing() calls to them. ParamMap2 introduced its own ParamDlg subclass ([Class IAutoMParamDlg](#)) that you can ask it to build for you and have manage all interaction with the Materials Editor automatically. As before, this still needs to know about the other (sub-object) ParamDlgs, and so it has the ability to keep a list of 'secondary' ParamDlgs to which it passes on the SetTime()s and SetThing()s. When the Materials Editor asks the texmap to CreateParamDlg(), the texmap asks its ClassDesc2 to build one of these (ClassDesc2::CreateParamDlgs()). If the texmap itself has multiple ParamBlock2s, CreateParamDlgs() builds one ParamDlg per pblock/rollout, makes the first of them a 'master' and adds the rest as secondary ParamDlgs. The texmap then asks the UVGen and TextureOutput objects to CreateParamDlg() for their rollouts and adds them to the master IAutoMParamDlg also.

Now consider the need for the **SetDlgThing()** method below. It is related to the SetThing() method that the Materials Editor calls on the 'master' ParamDlg to switch into the UI a texmap of the same class as that currently in the UI. Normally, the master IAutoParamDlg would propagate the SetThing() to its registered secondary ParamDlgs. In the case of multiple paramblock2s in the texmap, this would be correct, since the 'thing' in this case is the incoming texmap. But this doesn't work for the secondary UVGen and TextureOutput ParamDlgs; their 'things' are the UVGen and TextureOutput subobjects of the incoming map. So, IAutoParamDlg first calls SetDlgThing() on the incoming texmap so that it gets a chance to call the correct SetThing()s on the sub-object ParamDlgs with the appropriate incoming sub-objects. A clear example of this is in **Gradient::SetDlgThing()** in

**\MAXSDK\SAMPLES\MATERIALS\GRADIENT.CPP**. It is called once for each secondary ParamDlg. For those ParamDlgs that have special SetThing() requirements, it does the appropriate sub-object SetThing() and returns TRUE. If it does no special handling for a particular ParamDlg, it returns FALSE, signalling to the IAutoMParamDlg that it should do the standard SetThing() propagation for that dialog.

The Render Effects dialog has a similar arrangement to the Materials Editor for controlling Effect UI and so there is an **IAutoEParamDlg** that works exactly the same way as the **IAuotMParamDlg**.

**Parameters:****HWND hwMtlEdit**

The window handle of the materials editor.

**IMtlParams \*imp**

The interface pointer for calling methods in 3ds max.

**Return Value:**

A pointer to the created instance of a class derived from **ParamDlg**.

**Multiple Map Display in the Viewports****Prototype:****virtual BOOL SupportsMultiMapsInViewport();****Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

Returns TRUE if this material supports the display of multi-maps in the viewports (interactive renderer); FALSE if it doesn't.

**Default Implementation:****{ return FALSE; }****Prototype:****virtual void SetupGfxMultiMaps(TimeValue t, Material \*mtl, MtlMakerCallback &cb);****Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

This method is called to initialize the interactive renderer Material passed with the properties of this MtlBase.

If a **MtlBase** (material or texmap) wants to display multiple textures in the viewports, it implements

**SupportsMultiMapsInViewport()** to return TRUE, and implements **SetupGfxMultiMaps** to store the necessary information in the Material passed in, including the **TextureInfo**'s for each texture.

The **MtlMakerCallback** passed in to **SetupGfxMultiMaps** provides functions to help in setting up the "Material" data structure. The function **NumberTexturesSupported()** lets you know the capabilities of the current hardware, so you can adapt accordingly. The function **GetGfxTexInfoFromTexmap** fills in the fields of a **TextureInfo** except the texHandle and texture ops.

The implementation of **SetupGfxMultiMaps** must create the "texHandle" for each of the textures described in its **TextureInfo** array. It typically does this by calling the submap's **GetVPDisplayDIB()** method, and then creates the texHandle by calling the callback function **MakeHandle(bmi)**. To avoid doing this calculation when not necessary it is best to save the texHandles along with their validity intervals. Then when **SetupGfxMultiMaps** is called, if valid texHandles are already in hand they can just be used without recomputing.

#### **Parameters:**

##### **TimeValue t**

The time at which to evaluate the material.

##### **Material \*mtl**

Points to the interactive renderer material to update.

##### **MtlMakerCallback &cb**

This callback object is provided as a helper to fill in the Material properties above. See [Class MtlMakerCallback](#).

#### **Default Implementation:**

```
{}
```

## **Loading and Saving Methods**

#### **Prototype:**

```
IOResult Load(ILoad *iload);
```

#### **Remarks:**

Implemented by the System.

This method is called to load the material or texture from disk. The common MtlBase data must be loaded as well. See the code below.

#### **Parameters:**

## **ILoad \*iload**

An interface pointer for calling methods to load data. [See Class ILoad](#).

### **Return Value:**

See [List of IOResults](#).

### **Sample Code:**

```
IOResult Gradient::Load(ILoad *iload) {  
    IOResult res;  
    int id;  
    while (IO_OK==(res=iload->OpenChunk())) {  
        switch(id = iload->CurChunkID()) {  
            case MTL_HDR_CHUNK:  
                res = MtlBase::Load(iload);  
                break;  
            case MAPOFF_CHUNK+0:  
            case MAPOFF_CHUNK+1:  
            case MAPOFF_CHUNK+2:  
                mapOn[id-MAPOFF_CHUNK] = 0;  
                break;  
        }  
        iload->CloseChunk();  
        if (res!=IO_OK)  
            return res;  
        }  
        iload->RegisterPostLoadCallback(  
            new  
            ParamBlockPLCB(versions,NUM_OLDVERSIONS,&curVersion,this,1));  
        return IO_OK;  
    }  
}
```

### **Prototype:**

**IOResult Save(ISave \*isave);**

### **Remarks:**

Implemented by the System.

This method saves the plug-in's data to disk.. The common MtlBase data must be saved as well. The base class method must be called in a chunk at the beginning of every Mtl and Texmap.

**Parameters:****ISave \*isave**

An interface pointer available for saving data. See [Class ISave](#).

**Return Value:**

See [List of IOResults](#).

**Sample Code:**

Note in the code below the base class method is called in a chunk before the rest of the plug-ins data is saved.

```
IOResult Gradient::Save(ISave *isave) {  
    IOResult res;  
    // Save common stuff  
    isave->BeginChunk(MTL_HDR_CHUNK);  
    res = MtlBase::Save(isave);  
    if (res!=IO_OK) return res;  
    isave->EndChunk();  
  
    for (int i=0; i<NSUBTEX; i++) {  
        if (mapOn[i]==0) {  
            isave->BeginChunk(MAPOFF_CHUNK+i);  
            isave->EndChunk();  
        }  
    }  
    return IO_OK;  
}
```

**Enumerate Auxilliary Files Implementation****Prototype:**

```
void EnumAuxFiles(NameEnumCallback& nameEnum, DWORD  
flags);
```

**Remarks:**

This method is available in release 2.0 and later only.

This is an implementation of the **Animatable** method. This default implementation simply recurses, skipping inactive subTexmaps if appropriate.

## **GBuffer (system) methods.**

### **Prototype:**

**ULONG GetGBufID();**

### **Remarks:**

Implemented by the System.

Returns the G-buffer ID of this material.

### **Prototype:**

**void SetGBufID(ULONG id);**

### **Remarks:**

Implemented by the System.

Sets the G-buffer ID of this material.

## **Operators:**

### **Prototype:**

**MtlBase& operator=(const MtlBase& m);**

### **Remarks:**

Implemented by the System.

Materials and Texmaps must use this operator to copy the common portion of themselves when cloning.

## **Postage Stamp Image Methods**

The Material / Map Browser supports the display of small and large icon images for material and texture maps. The methods below deal with the creation, access and deletion of these images. The small size image is 32 pixels. The large size is 88 pixels.

### **Prototype:**

**PStamp\* GetPStamp(int sz);**

### **Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Returns a pointer to the postage stamp image for the file.

**Parameters:**

**int sz**

One of the following values:

**PS\_SMALL** for small (32x32) images.

**PS\_LARGE** for large (88x88) images.

**PS\_TINY** for tiny (24x24) images.

**Prototype:**

**PStamp\* CreatePStamp(int sz, BOOL Render = FALSE);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Creates a postage stamp image and returns a pointer to it. If the postage stamp image already exists then it is simply returned.

Here's an example using this method to display a small material sample.

```
void DisplayMB(MtlBase *mb, HDC hdc, int x, int y) {  
mb->CreatePStamp(0,TRUE); // create and render a small pstamp  
PStamp *ps = mb->GetPStamp(0);  
if (ps) {  
int d = PSDIM(0);  
int scanw = ByteWidth(d);  
int nb = scanw*d;  
UBYTE *workImg = new UBYTE[nb];  
if (workImg==NULL)  
return;  
ps->GetImage(workImg);  
Rect rect;  
rect.left = x;  
rect.top = y;
```

```
rect.right = x+d;
rect.bottom = y+d;
GetGPort()->DisplayMap(hdc, rect,0,0, workImg, scanw);
delete [] workImg;
    }
}
```

**Parameters:**

**int sz**

One of the following values:

**PS\_SMALL** for small (32x32) images.

**PS\_LARGE** for large (88x88) images.

**PS\_TINY** for tiny (24x24) images.

**BOOL Render = FALSE**

This parameter is available in release 4.0 and later only.

If set to true, the postage stamp bitmap will have the **MtlBase** rendered into it automatically. The bitmap can then be retrieved using **PStamp::GetImage**, for drawing in the UI.

**Prototype:**

```
void DiscardPStamp(int sz);
```

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the System.

Discards the postage stamp image.

**Parameters:**

**int sz**

One of the following values:

**PS\_SMALL** for small (32x32) images.

**PS\_LARGE** for large (88x88) images.

**PS\_TINY** for tiny (24x24) images.

## **Internal Methods**

**Prototype:**

**int GetMeditObjType()**

**Remarks:**

This method is used internally.

**Prototype:**

**void SetMeditObjType(int t)**

**Remarks:**

This method is used internally.

**Prototype:**

**int GetMeditTiling()**

**Remarks:**

This method is used internally.

**Prototype:**

**void SetMeditTiling(int t)**

**Remarks:**

This method is used internally.

**Prototype:**

**BOOL TextureDisplayEnabled()**

**Remarks:**

This method is used internally.

**The following methods are for doing interactive texture display.**

**Prototype:**

**virtual BOOL SupportTexDisplay()**

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if this texture supports being used in the interactive renderer; otherwise FALSE. If the texture does return TRUE it is expected to implement the methods **ActivateTexDisplay()** and **GetActiveTexHandle()**.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void ActivateTexDisplay(BOOL onoff)
```

**Remarks:**

Implemented by the Plug-In.

This method is called when the usage of the texture the interactive renderer changes. This method must only be implemented if **SupportTexDisplay()** returns TRUE. This method does not cause the texture map to be drawn in the viewport but should be called with TRUE as the argument before this can occur. For viewport drawing of textures refer to **Interface::ActivateTexture()** and **Interface::DeActivateTexture()** instead.

**Parameters:**

**BOOL onoff**

TRUE if the texture is being used; FALSE if it is no longer being used.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual DWORD GetActiveTexHandle(TimeValue t,  
TexHandleMaker& thmaker)
```

**Remarks:**

Implemented by the Plug-In.

This method is called to retrieve a texture handle to this texture map.

**Parameters:**

**TimeValue t**

The time to return the texture handle.

**TexHandleMaker& thmaker**

This class provides methods for creating a texture handle from a 3ds max bitmap and a Windows DIB. It also has a method to retrieve the required size of the texture map. See [Class TexHandleMaker](#).

**Return Value:**

The texture handle.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
void IncrActive();
```

**Remarks:**

This method is used internally.

**Prototype:**

```
void DecrActive();
```

**Remarks:**

This method is used internally.

**Prototype:**

```
int Active();
```

**Remarks:**

This method is used internally.

# ReferenceMaker

See Also: [Class ReferenceMaker](#).

Any entity that makes references must be derived from this class. A reference creates an official record of the dependency between a ReferenceMaker and a ReferenceTarget. 3ds max uses a messaging system to notify ReferenceMakers about changes in their ReferenceTargets.

ReferenceMaker provides two methods associated with this system.

## **NotifyDependents()**

A plug-in calls this method any time it needs to broadcast a message to all its dependents. Typically this is a **REFMSG\_CHANGE** message indicating the plug-in has changed.

## **NotifyRefChanged()**

This method is called by the system when one of the entities a plug-in references calls **NotifyDependents()** to broadcast a message. The plug-in implements **NotifyRefChanged()** to receive the messages.

A reference maker must provide access to its references via the following methods:

## **NumRefs(), GetReference(i), SetReference(i)**

The system calls these methods when the plug-in creates and deletes references.

ReferenceMaker has additional methods for creating, replacing and deleting references.

## **MakeRefByID()**

Makes a new reference given the reference target and the index of the reference within the reference maker.

## **ReplaceReference()**

Deletes a reference if it is non-null and makes a new reference to a new target.

## **DeleteReference()**

Deletes a particular reference.

## **DeleteAllRefsFromMe()**

This method deletes all references that the reference maker has.

## **DeleteAllRefs()**

This deletes all references to AND from the reference maker.

## **EnumDependents()**

This allows a reference target to enumerate all reference to it. All reference targets have a list of backpointers to entities that reference it. This method enumerates those back pointers calling the given callback object once per dependent.

ReferenceMaker also has methods for the loading and saving of plug-ins.

## **Load()**

This method is called when a 3ds max file itself is being loaded. The plug-in would respond by loading its data through the given interface pointer. If the plug-in has references to other sub components, the system will automatically call **Load()** on the sub-components.

## **Save()**

This method is called when a 3ds max file itself is being saved. The plug-in would respond by saving its data through the given interface pointer.

# Class ReferenceMaker

See Also: [Class Animatable](#), [Class ReferenceTarget](#), [List of Reference Messages](#), [References](#), [Class ILoad](#), [Class ISave](#).

```
class ReferenceMaker : public Animatable
```

## Description:

Any entity that makes references must be derived from this class. A reference creates a record of the dependency between a **ReferenceMaker** and a **ReferenceTarget**. 3ds max uses a messaging system to notify dependent entities about changes. This class has a method that receives the notifications its dependents send when they change. It has methods that return the number of references it has, and methods to get and set these references. Also, there are methods for creating, replacing and deleting the references. File input/output is handled via methods of this class (**Load()** and **Save()**).

See the Advanced Topics section on [References](#) for an overview of the 3ds max reference architecture.

## Method Groups:

The hyperlinks below jump to the start of groups of related methods within the class:

[Dependent Notification](#)

[Reference Access \(Num/Get/Set\)](#)

[Making / Replacing / Deleting References](#)

[Finding/Checking Reference Targets](#)

[Loading / Saving Methods](#)

## Methods:

### Dependent Notification

#### Prototype:

```
virtual RefResult NotifyRefChanged(Interval changeInt,  
RefTargetHandle hTarget, PartID& partID,RefMessage  
message)=0;
```

#### Remarks:

Implemented by the Plug-In.

A plug-in which makes references must implement a method to receive and respond to messages broadcast by its dependents. This is done by implementing **NotifyRefChanged()**.

The plug-in developer usually implements this method as a switch statement where each case is one of the messages the plug-in needs to respond to.

Note: For developer that need to update a dialog box with data about an object you reference note the following related to this method: This method may be called many times. For instance, say you have a dialog box that displays data about an object you reference. This method will get called many time during the drag operations on that object. If you updated the display every time you'd wind up with a lot of 'flicker' in the dialog box. Rather than updating the dialog box each time, you should just invalidate the window in response to the **NotifyRefChanged()** call. Then, as the user drags the mouse your window will still receive paint messages. If the scene is complex the user may have to pause (but not let up on the mouse) to allow the paint message to go through since they have a low priority. This is the way many windows in 3ds max work.

#### **Parameters:**

##### **Interval changeInt**

This is the interval of time over which the message is active. Currently, all plug-ins will receive **FOREVER** for this interval.

##### **RefTargetHandle hTarget**

This is the handle of the reference target the message was sent by. The reference maker uses this handle to know specifically which reference target sent the message.

##### **PartID& partID**

This contains information specific to the message passed in. Some messages don't use the **partID** at all. See the section [List of Reference Messages](#) for more information about the meaning of the **partID** for some common messages.

##### **RefMessage message**

The **message** parameters passed into this method is the specific message which needs to be handled. See [List of Reference Messages](#).

**Return Value:**

The return value from this method is of type **RefResult**. This is usually **REF\_SUCCEED** indicating the message was processed. Sometimes, the return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

**Reference Access**

3ds max manages the access to an items references by using a virtual array. If the plug-in makes references, it must implement these three methods handle access to its references: **NumRefs()**, **GetReference(i)**, **SetReference(i)**.

**Prototype:**

```
virtual int NumRefs();
```

**Remarks:**

Implemented by the Plug-In.

The plug-in implements this method to return the number of references it makes. **NumRefs()** indicates valid values for 'i' when doing a **GetReference(i)** or **SetReference(i)**. An item may return different values at different times if the number of references is changing.

**Return Value:**

The number of references made by the plug-in.

**Default Implementation:**

```
{ return 0;}
```

**Prototype:**

```
virtual RefTargetHandle GetReference(int i);
```

**Remarks:**

Implemented by the Plug-In.

The plug-in implements this method to return its 'i th' reference. The plug-in simply keeps track of its references using an integer index for each one. When the system calls this method, the plug-in returns its 'i th' reference.

**Parameters:**

**int i**

The virtual array index of the reference to get.

**Return Value:**

The reference handle of the 'i-th' reference.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void SetReference(int i, RefTargetHandle rtarg);
```

**Remarks:**

Implemented by the Plug-In.

The plug-in implements this method to store the reference handle passed into its 'i-th' reference. The plug-in simply keeps track of its references using an integer index for each one. When the system calls this method, the plug-in stores its 'i-th' reference.

**Parameters:**

**int i**

The virtual array index of the reference to store.

**RefTargetHandle rtarg**

The reference handle to store.

**Default Implementation:**

```
{ }
```

## **Making / Replacing / Deleting References**

**Prototype:**

```
RefResult MakeRefByID(Interval refInterval,int which,  
RefTargetHandle htarget)
```

**Remarks:**

Implemented by the System.

This method creates a reference between the object which calls the method,

and the ReferenceTarget specified by the **htarget** parameter.

Note that this method calls **SetReference()** after the reference is made to initialize it.

**Parameters:**

**refInterval**

Currently, all plug-ins must use **FOREVER** for this interval.

Indicates the interval of time over which this reference is active. Outside this interval, the reference is not considered to be a dependency. This allows the plug-in to have dependent relationship over only portions of an entire animation time range. If a plug-in has a dependency over the entire animation it may use the pre-defined interval **FOREVER** for this parameter.

**which**

Indicates which virtual array index this newly created reference is assigned to. The system uses a virtual array mechanism to access the references an item has. The developer simply assigns an integer index to each reference.

**hTarget**

This parameter is the handle of the item we are making a reference to.

**Return Value:**

The return value from this method is of type **RefResult**. This is usually **REF\_SUCCEED** indicating the reference was created and is registered by the reference target; otherwise **REF\_FAIL**.

**Prototype:**

**RefResult ReplaceReference(int which, RefTargetHandle newtarg, BOOL delOld=TRUE);**

**Remarks:**

Implemented by the System.

This method is used to replace a reference, for example when cloning reference makers, to delete an old reference and make a new one.

**Parameters:**

**int which**

The virtual array index of the reference to replace.

**RefTargetHandle newtarg**

The new reference target

**BOOL delOld=TRUE**

If TRUE, the old reference is deleted.

**Return Value:**

This is usually **REF\_SUCCEED** indicating the reference was replaced; otherwise **REF\_FAIL**.

**Prototype:**

**RefResult DeleteAllRefsFromMe();**

**Remarks:**

Implemented by the System.

Deletes all references from this ReferenceMaker.

**Return Value:**

This is always **REF\_SUCCEED** indicating the references were deleted.

**Prototype:**

**virtual RefResult DeleteAllRefsToMe()**

**Remarks:**

Implemented by the System.

This method deletes all the references to this reference maker/reference target.

This also sends the **REFMSG\_TARGET\_DELETED** message to all dependents.

**Return Value:**

This is **REF\_SUCCEED** if the references were deleted; otherwise it is **REF\_FAIL**.

**Prototype:**

**RefResult DeleteAllRefs();**

**Remarks:**

Implemented by the System.

Deletes all references both to and from this item.

**Return Value:**

This is **REF\_SUCCEED** if the references were deleted; otherwise it is **REF\_FAIL**.

**Prototype:**

**void DeleteMe();**

**Remarks:**

Implemented by the System.

This deletes all reference to and from the item, sends

**REFMSG\_TARGET\_DELETED** messages, handles UNDO, and deletes the object.

**Prototype:**

**RefResult DeleteReference(int i);**

**Remarks:**

Implemented by the System.

This method deletes the reference whose virtual array index is passed. The other reference indices are not affected, i.e. the number of references is not reduced nor are they re-ordered in any way. Note the system calls

SetReference(i, NULL) to set that reference to NULL. Also, if this is the last reference to the item, the item itself is deleted by calling its **DeleteThis()** method.

**Parameters:**

**int i**

The virtual array index of the reference to delete.

**Return Value:**

This is **REF\_SUCCEED** if the reference was deleted; otherwise it is **REF\_FAIL**.

**Prototype:**

**virtual BOOL CanTransferReference(int i)**

**Remarks:**

This method is used internally. It is used by certain system objects that have references. A reference maker can choose not to let TransferReferences() affect it. Note that plug-ins probably should not use this.

**Default Implementation:**

```
{return TRUE;}
```

## **Finding/Checking Reference Targets**

**Prototype:**

```
int FindRef(RefTargetHandle rtarg)
```

**Remarks:**

Implemented by the System.

This method returns the virtual array index of the reference target passed.

**Parameters:**

**RefTargetHandle rtarg**

The reference target to find the index of.

**Return Value:**

The virtual array index of the reference target to find. If the reference target is not found, -1 is returned.

**Prototype:**

```
virtual BOOL IsRefTarget();
```

**Remarks:**

This function differentiates things subclassed from **ReferenceMaker** from subclasses of **ReferenceTarget**. The implementation of this method (in **ReferenceMaker**) returns FALSE and its implementation in **ReferenceTarget** returns TRUE. This can be useful when tracing back up the reference hierarchy, to know when you run into something that was subclassed directly off of **ReferenceMaker**, and hence to stop the traversal at that point.

**Default Implementation:**

```
{ return FALSE; };
```

**Prototype:**

**virtual BOOL IsRealDependency(ReferenceTarget \*rtarg)**

**Remarks:**

When a reference target's last "real" reference is deleted the target is deleted.

Any leftover "non-real" reference makers will receive a

**REFMSG\_TARGET\_DELETED** notification. This method returns TRUE if the reference dependency is "real". Otherwise it returns FALSE.

Certain references are not considered "real" dependencies. For instance, internally there are certain reference makers such as the object that handles editing key info in the motion branch. This object implements this method to return FALSE because it is not a "real" reference dependency. It's just needed while the editing is taking place. Plug-in developers don't need to concern themselves with this method because it is used internally.

**Parameters:**

**ReferenceTarget \*rtarg**

A pointer to the reference target.

**Default Implementation:**

```
{return TRUE;}
```

## Loading / Saving Methods

**Prototype:**

**virtual IOResult Save(ISave \*isave);**

**Remarks:**

Implemented by the Plug-In.

Called by the system to allow the plug-in to save its data.

**Parameters:**

**ISave \*isave**

This pointer may be used to call methods to write data to disk. See the Advanced Topics section on [Loading and Saving](#) more an overview of the load/save process.

**Return Value:**

**IO\_OK**

The result was acceptable - no errors.

### **IO\_ERROR**

This should be returned if an error occurred.

#### **Default Implementation:**

```
{ return IO_OK; }
```

#### **Prototype:**

```
virtual IOResult Load(ILoad *iload);
```

#### **Remarks:**

Implemented by the Plug-In.

Called by the system to allow the plug-in to load its data. See the Advanced Topics section on [Loading and Saving](#) more an overview of the load/save process.

#### **Parameters:**

##### **ILoad \*iload**

This interface pointer may be used to call methods to read data from disk.

#### **Return Value:**

##### **IO\_OK**

The result was acceptable - no errors.

##### **IO\_ERROR**

This should be returned if an error occurred.

#### **Default Implementation:**

```
{ return IO_OK; }
```

#### **Prototype:**

```
virtual int RemapRefOnLoad(int iref)
```

#### **Remarks:**

This method is used when you have modified a **ReferenceMaker** to add or delete references, and are loading old files. It gets called during the reference mapping process, after the **Load()** method is called. You determine what version is loading in the **Load()**, and store the version in a variable which you can look at in **RemapRefOnLoad()** to determine how to remap references.

The default implementation of this method just returns the same value it is passed, so you don't need to implement it unless you have added or deleted references from your class. This method makes it a lot easier to load old files when the reference topology has changed.

**Parameters:**

**int iref**

The input index of the reference.

**Return Value:**

The output index of the reference.

**Default Implementation:**

```
{ return iref; }
```

**Prototype:**

```
virtual void RescaleWorldUnits(float f);
```

**Remarks:**

This method is available in release 2.0 and later only.

This may be implemented to rescale the size of all world units in reference hierarchy. Developers must call

```
if (TestAFlag(A_WORK1))
```

```
    return;
```

```
    SetAFlag(A_WORK1);
```

before doing this on a reference hierarchy.

See the sub-section 'Scaling Parameter Values' in [Updating MAX 1.0 Plug-Ins to work with MAX 2.0](#) for additional details.

**Parameters:**

**float f**

The scale factor.

**Prototype:**

```
virtual void SaveEnum(SaveEnumProc& sep, BOOL  
isNodeCall=0);
```

**Remarks:**

This method is used internally.

**Prototype:**

**void BlockEval()**

**Remarks:**

Implemented by the System.

This method is used internally.

**Prototype:**

**void UnblockEval()**

**Remarks:**

Implemented by the System.

This method is used internally.

**Prototype:**

**int Evaluating()**

**Remarks:**

Implemented by the System.

This method is used internally.

**Prototype:**

**RefResult StdNotifyRefChanged(Interval changeInt,  
RefTargetHandle hTarget, PartID partID, RefMessage message,  
BOOL propagate=TRUE);**

**Remarks:**

This method is used internally.

**The following methods are for release 4.2 or above**

**Prototype:**

**virtual void\* GetInterface(ULONG id)**

**Remarks:**

Returns a pointer to the interface.

**Parameters:****ULONG id**

The id of the interface.

**Prototype:****virtual BaseInterface\* GetInterface(Interface\_ID id)****Remarks:**

Returns a pointer to the Base Interface for the interface ID passed.

**Parameters:****Interface\_ID id**

The unique ID of the interface to get

## Reference Target

See Also: [Class ReferenceTarget](#).

Almost all plug-ins are derived from ReferenceTarget. And because ReferenceTarget is derived from ReferenceMaker, these plug-ins may also make references.

The primary methods allow deleting any references that have been made to the target, and copying of the reference target.

### **DeleteAllRefsToMe()**

This method deletes all references to the target. This is often used when an item is being deleted and it wants to remove all references from itself.

### **Clone()**

This method copies both the data structure and all the data residing in the data structure of the reference target.

## Class ReferenceTarget

See Also: [Class ReferenceMaker](#), [Class RefList](#), [Class DependentEnumProc](#), [Class RemapDir](#), [References](#).

```
class ReferenceTarget : public ReferenceMaker
```

### Description:

This class is used by anything that may need to have references made to it. Most plug-ins are derived from this class. This class includes methods to send notification of changes by the object, seeing which other objects in the system depend on it (reference it), and a method used to create a copy of the object ('clone' it). See the Advanced Topics section on [References](#) for an overview of the 3ds max reference architecture.

### Method Groups:

The hyperlinks below jump to the start of groups of related methods within the class:

[Dependent Notification](#)

[Cyclic Reference Testing / Dependency Testing](#)

[Adding / Deleting / Transferring References](#)

[Dependent Enumeration](#)

[Cloning](#)

### Methods:

#### Dependent Notification

##### Prototype:

```
virtual RefResult NotifyDependents(Interval changeInt, PartID  
partID, RefMessage message, SClass_ID  
sclass=NOTIFY_ALL,BOOL propagate=TRUE, RefTargetHandle  
hTarg=NULL );
```

##### Remarks:

Implemented by the System.

This method broadcasts the message specified by the **message** parameter to all the items which reference this item.

Note the following on how reference messages propagate (that is, travel to the dependents):

When a plug-in sends a message via **NotifyDependents()**, the message propagates to ALL the items that reference it. And also to all the items which reference those items. And so on. The only exceptions to this are as follows:

- 1) The **propagate** parameter passed is **FALSE**. In that case the message only goes to the immediate dependents.
- 2) If the **SClass\_ID sclass=NOTIFY\_ALL** parameter limits the dependents to a certain specified Super Class.
- 3) If one of the items that references the plug-in processes the message inside its **NotifyRefChanged()** and returns **REF\_STOP** instead of **REF\_SUCCEED**. In this case, the message is not further propagated.

Also, whenever a message propagates, the **hTarget** parameter received in **NotifyRefChanged()** is reset to the **this** pointer of the immediate dependent (not the originator) who propagates the message.

#### Parameters:

##### **Interval changeInt**

Currently all plug-ins must pass **FOREVER** for this interval. This indicates the interval of time over which the change reported by the message is in effect.

##### **PartID partID**

This parameter is used to pass message specific information to the items which will receive the message. See the

**ReferenceMaker::NotifyRefChanged()** method for more details.

##### **RefMessage message**

The message to broadcast to all dependents. See the

**ReferenceMaker::NotifyRefChanged()** method for more details. See

[List of Reference Messages](#).

##### **SClass\_ID sclass=NOTIFY\_ALL**

This parameter defaults to **NOTIFY\_ALL**. If this value is passed to **NotifyDependents()** all dependents will be notified. Other super class values may be passed to only send the message to certain items whose SuperClassID matches the one passed.

**BOOL propagate=TRUE**

This parameter defaults to TRUE. This indicates that the message should be sent to all 'nested' dependencies. If passed as FALSE, this parameter indicates the message should only be sent to first level dependents. Normally this should be left to default to TRUE.

**RefTargetHandle hTarg=NULL**

This parameter must always default to NULL.

**Return Value:**

This method always returns **REF\_SUCCEED**.

**Prototype:**

**virtual void NotifyForeground(TimeValue t)**

**Remarks:**

Implemented by the System.

This method is called to flag dependents into the foreground. The default implementation just sends out the notification

**REFMSG\_FLAGDEPENDENTS** with **PART\_PUT\_IN\_FG** as the **partID**. In particular, a slave controller could override this method and call its master's version of this method.

**Parameters:**

**TimeValue t**

The time to send the notification.

**Prototype:**

**virtual void NotifyTarget(int message, ReferenceMaker\* hMaker);**

**Remarks:**

This method is available in release 4.0 and later only.

Used by a ReferenceMaker to send 'reverse' notification messages to its ReferenceTargets.

**Parameters:**

**int message**

The message sent to the reference target. See List of Reference Target Messages. **\*\*aztodo\*\*** link this

## **ReferenceMaker\* hMaker**

The ReferenceMaker sending the message.

## **Cyclic Reference Testing / Dependency Testing**

### **Prototype:**

**RefResult TestForLoop(Interval refInterval, RefMakerHandle hmaker);**

### **Remarks:**

Implemented by the System.

This method may be called to test for cyclical references. Normally developers don't need to call this method since when a developer calls **MakeRefByID()** to make a reference it performs its own test internally and will not succeed if there is a cyclical reference being created.

### **Parameters:**

#### **Interval refInterval**

This interval is reserved for future use. Currently any plug-in should specify FOREVER for this interval.

#### **RefMakerHandle hmaker**

The reference maker performing the loop test.

### **Return Value:**

**REF\_SUCCEED** if a cyclic reference would be created; otherwise **REF\_FAIL**.

### **Prototype:**

**HasDependents()**

### **Remarks:**

Implemented by the System.

Returns nonzero if the reference target has items that reference it; otherwise 0.

### **Prototype:**

**RefList& GetRefList()**

### **Remarks:**

Implemented by the System.

Returns a list of references to this reference target.

**Return Value:**

List of references to this reference target.

**Prototype:**

**void BeginDependencyTest()**

**Remarks:**

Implemented by the System.

To see if this reference target depends on something: first call

**BeginDependencyTest()** then call **NotifyDependents()** on the thing with the message **REFMSG\_TEST\_DEPENDENCY**. If

**EndDependencyTest()** returns TRUE this target is dependent on the thing.

**Prototype:**

**BOOL EndDependencyTest()**

**Remarks:**

Implemented by the System.

To see if this reference target depends on something: first call

**BeginDependencyTest()** then call **NotifyDependents()** on the thing with the message **REFMSG\_TEST\_DEPENDENCY**. If

**EndDependencyTest()** returns TRUE this target is dependent on the thing.

**Return Value:**

Returns TRUE if the target is dependent.

## **Adding / Deleting / Transferring References**

**Prototype:**

**virtual void RefAdded(RefMakerHandle rm)**

**Remarks:**

Implemented by the Plug-In.

This method is called after making a reference. If the target needs to know it can override it.

**Parameters:**

**RefMakerHandle rm**

The ReferenceMaker creating the reference.

**Prototype:**

**RefResult DeleteAllRefsToMe();**

**Remarks:**

Implemented by the System.

This method deletes all references to this reference target.

**Return Value:**

Always returns **REF\_SUCCEED**.

**Prototype:**

**virtual RefResult AutoDelete()**

**Remarks:**

Implemented by the Plug-In.

This method is called when a target's last reference is deleted. Most subclasses will not need to override this. If you don't want to be deleted when the last reference is deleted, plug in a no-op.

More likely, a developer would override **Animatable::DeleteThis()** if they didn't want to be deleted from memory.

**Return Value:**

Always returns **REF\_SUCCEED**.

**Prototype:**

**virtual void RefDeleted()**

**Remarks:**

Implemented by the Plug-In.

This is called after deleting a reference to a reference target, in the case that the target was not deleted. If a target needs to know, it should override this method.

**Prototype:**

**virtual void RefDeletedUndoRedo();**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called when a reference is deleted because of an undo or a redo.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void RefAddedUndoRedo(RefMakerHandle rm);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called when a reference is added because of an undo or a redo.

**Parameters:**

**RefMakerHandle rm**

The reference maker that is now referencing this target.

**Default Implementation:**

**{}**

**Prototype:**

**RefResult TransferReferences(RefTargetHandle oldTarget, BOOL delOld=FALSE);**

**Remarks:**

Implemented by the System.

This method is used to transfer all the references from **oldTarget** to this reference target.

**Parameters:**

**RefTargetHandle oldTarget,**

The previous reference target.

**BOOL delOld=FALSE);**

If this is TRUE the previous reference target is deleted.

**Return Value:**

Always returns **REF\_SUCCEED**.

**Dependent Enumeration****Prototype:**

```
virtual int EnumDependents(DependentEnumProc* dep);
```

**Remarks:**

Implemented by the System.

This allows a reference target to enumerate all reference to it. All reference targets have a list of back pointers to entities that reference it. This method enumerates those back pointers calling the given callback object once per dependent.

**Parameters:**

**DependentEnumProc\* dep**

The callback object called for each dependent.

**Return Value:**

Return 1 to stop the enumeration and 0 to continue.

**Cloning****Prototype:**

```
virtual RefTargetHandle Clone(RemapDir &remap =  
NoRemap());
```

**Remarks:**

Implemented by the Plug-In.

This method is called to have the plug-in clone itself. This method should copy both the data structure and all the data residing in the data structure of this reference target. The plug-in should clone all its references as well.

**Important Note:** See the remarks in method **BaseClone()** below.

**Parameters:**

**RemapDir &remap = NoRemap()**

This class is used for remapping references during a Clone. See [Class](#)

[RemapDir](#).

**Return Value:**

A pointer to the cloned item.

**Prototype:**

**virtual void BaseClone(ReferenceTarget \*from, ReferenceTarget \*to, RemapDir &remap);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

**Important Note:** All plug-ins that implement a **Clone()** method have to call this **BaseClone()** method with the old and the new object as parameters. The ordering in regards to when is method is called is unimportant however it must, of course, be called after the cloned object is created.

This method allows base classes to copy their data into a new object created by the clone operation. All overwrites of **BaseClone()** **must** call the base class implementation. The base class implementation copyies the CustAttrib objects into the newly created object.

**Parameters:**

**ReferenceTarget \*from**

Points to the old object to clone.

**ReferenceTarget \*to**

Points to the new object created.

**RemapDir &remap**

This class is used for remapping references during a Clone. See [Class RemapDir](#).

**Prototype:**

**RefResult MaybeAutoDelete();**

**Remarks:**

This method is used internally.

**The following methods are for release 4.2 and above**

**Prototype:**

**virtual void\* GetInterface(ULONG id)**

**Remarks:**

Returns a pointer to the interface.

**Parameters:**

**ULONG id**

The id of the interface.

**Prototype:**

**virtual BaseInterface\* GetInterface(Interface\_ID id)**

**Remarks:**

Returns a pointer to the Base Interface for the interface ID passed.

**Parameters:**

**Interface\_ID id**

The unique ID of the interface to get

# Object

See Also: [Class Object](#).

The object class is the base class for all objects. An object is one of two things: A procedural object or a derived object. Derived objects are part of the system and may not be created by plug-ins. They are containers for modifiers.

Procedural objects can be many different things such as cameras, lights, helper objects, geometric objects, etc.

These are some of the main methods of Object.

## **InitNodeName()**

This method provides a default name for the node in the scene when the object is first created.

## **DoOwnSelectHilite()**

If a plug-in overrides this method to return TRUE, then the plug-in becomes responsible for drawing itself in a selected state (if it is selected). Otherwise, the system sets the current line/material color appropriately.

## **ObjectValidity()**

This method returns the validity interval for the object. This is the intersection of the validity intervals for all of the object's channels.

## **IsDeformable()**

If an object is deformable as a point object, then it should return TRUE from this method. A deformable object is a generic point object that supports the following methods:

## **NumPoints()**

Returns the number of deformable points

## **GetPoint(i)**

Returns the 'i-th' point.

## **SetPoint(i)**

This method sets the 'i-th' point.

## **PointsWereChanged()**

If a modifier calls **SetPoint()**, when it is finished it should call this method so the object can invalidate and/or update its bounding box and any other data it might cache.

## **Deform()**

This is the preferred method of deforming a deformable object rather than calling **GetPoint(i)** and **SetPoint(i)**. This method applies the given deformable object to its points (or just its selected points if the selected parameter is specified).

### **GetDeformBBox()**

Gets the bounding box of the object in a particular coordinate system specified by the given matrix. This will only be called on an object that is itself deformable.

All objects must be able to convert to **TriObjects**. And optionally they may convert to other types. A modifier specifies the type of object it wishes to operate on (using **InputType()**). The modifier may only be applied if the object is of the type that the modifier requests or is capable of converting to that type.

### **CanConvertToType()**

This method returns a boolean to indicate if the object can be converted to the desired type passed in.

### **ConvertToType()**

If the object can be converted to the type passed in, this method creates a new instance of that object type and returns it.

### **IntersectRay()**

This is used for ray tracing, or defining planes of tangency at a point (for example, defining a construction plane at a specific point on an object). It is also used by commands such as Place Highlight that require the surface normal at a point on an object. The mesh class implements this method so if your object is represented by a mesh this may be calculated automatically.

# ParticleObject

See Also: [Class ParticleObject](#).

This is the base class from which plug-in particle systems may be derived. The main methods are:

## **ApplyForceField()**

A force field can be applied to a particle system by a space warp. The force field provides a function of position in space, velocity and time that gives a force. The force is then used to compute an acceleration on a particle which modifies its velocity. This method adds a force to the list of forces affecting a particle system.

## **ApplyCollisionObject()**

A collision object can be applied to a particle system by a space warp. The collision object checks a particle's position and velocity and determines if the particle will collide with it. This method adds a collision object to the list of collision objects affecting a particle system.

# PatchObject

See Also: Class [PatchObject](#).

A bezier patch object. This class implements many of the methods from BaseObject, Object and GeomObject to work with bezier patches.

# ShapeObject

See Also: [Class ShapeObject](#).

ShapeObjects are open or closed hierarchical shape objects. This is the base class that **SimpleSpline**, **SimpleShape**, **SplineShape**, and **LinearShape** are derived from. A ShapeObject can be made up of one or more splines thus a shape may have multiple 'pieces'.

# SimpleMod

See Also: [Class SimpleMod](#).

The SimpleMod class supplies most of the methods needed to implement a deformation modifier, reducing the amount of work needed to create one. This class is appropriate for modifiers which alter the geometry (vertices or points) of an object. The plug-in must implement a method to return the Deformer (the process which actually modifies the objects). If the plug-in displays any animated parameters to the user it must implement a method to update these parameters if the user has changed the current time slider.

These are the main methods:

## **GetDeformer()**

This gets the deformer callback object from the derived class. This is the class with methods that actually deforms the object.

## **UpdateUI()**

If the modifier is currently being edited, any parameters that appear in the user interface should be updated so that if the current time has changed, the UI will properly reflect the parameter values at that time.

## **GetValidity()**

This method computes the validity interval for the modifier.

# SimpleObject

See Also: [Class SimpleObject](#).

This is a base class to simplify the development of procedural objects. The procedural object must represent itself as a mesh in order to subclass off SimpleObject.

# SimpleParticle

See Also: [Class SimpleParticle](#).

Plug-In particle systems may be derived from SimpleParticle. The main methods are:

## **UpdateParticles()**

Update the state of the particles at the specified time.

## **BuildEmitter()**

Constructs the particle emitter at the time passed.

## **InvalidateUI()**

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

# SimpleShape

See Also: [Class SimpleShape](#).

Plug-In procedural shapes are subclassed from SimpleShape. The main methods are:

## **BuildShape()**

Called to create the spline shape at the specified time.

## **ValidForDisplay()**

Called to determine if the spline may be displayed at the time passed.

## **void InvalidateUI()**

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

# SimpleSpline

See Also: [Class SimpleSpline](#).

This is a class used in the creation of shape plug-ins. Most of the 3ds max shapes and splines are derived from this class. For example, Line, Arc, Circle, Ellipse and Star are all **SimpleSplines**.

# SimpleWSMMod

See Also: [Class SimpleWSMMod](#).

The SimpleWSMMod class supplies most of the methods needed to implement a Space Warp plug-in (world space modifier).

To be a Simple WSM modifier, the following assumptions are made:

The modifier only modifies the geometry channel.

The modifier uses an instance of a class derived from Deformer to do the modifying.

The main methods are:

## **GetDeformer()**

This method is used to retrieve the callback object that will handle the deformation.

## **UpdateUI()**

This method is called when the user has moved the time slider to a new time and the UI parameters need to be updated to reflect the correct value at the new time.

## **GetValidity()**

The SimpleWSMMod class calls this method to retrieve the validity interval of the modifier.

## Class SimpleWSMMod

See Also: [Class Modifier](#), [Class IParamBlock](#), [Class INode](#).

```
class SimpleWSMMod : public Modifier
```

### Description:

The SimpleWSMMod class supplies most of the methods needed to implement a world space modifier.

To be a 'Simple' WSM modifier, the following assumptions are made:

The modifier only modifies the geometry channel.

The modifier uses an instance of a class derived from Deformer to do the modifying.

This class maintains a pointer to a parameter block. If the client of **SimpleWSMMod** uses a single parameter block then **SimpleWSMMod** can manage all the methods associated with SubAnims and References for the client. If the client of **SimpleWSMMod** maintains several parameter blocks then the client must implement the methods **NumSubs()**, **SubAnim(i)**, **SubAnimName(i)**, **NumRefs()**, **GetReference(i)** and **SetReference(i)** and call the **SimpleWSMMod** methods when 'i' refers to the parameters maintained by **SimpleWSMMod**.

### Data Members:

Clients of **SimpleWSMMod** should use the following pointers when the references are created.

protected:

**IParamBlock \*pblock;**

Pointer to a parameter block.

**WSMObject \*obRef;**

Pointer to the world space modifier object referenced by the **WSMModifier**.

**INode \*nodeRef;**

Pointer to the node in the scene referenced by the **WSMModifier**.

**static IObjParam \*ip;**

Storage for the interface pointer.

**static SimpleWSMMod \*editMod;**

Storage for the modifier currently being edited in the command panel.

Clients of **SimpleWSMMod** should use the following values as the reference indexes of the object, node and parameter block.

```
#define SIMPWSMMOD_OBREF 0
```

```
#define SIMPWSMMOD_NODEREF 1
```

```
#define SIMPWSMMOD_PBLOCKREF 2
```

The example code below (from `\MAXSDK\SAMPLES\HOWTO\RIPPLE.CPP`) shows how these are used:

```
MakeRefByID(FOREVER,SIMPWSMMOD_NODEREF,node);  
MakeRefByID(FOREVER, SIMPWSMMOD_PBLOCKREF,  
CreateParameterBlock(descModVer0,  
MODPBLOCK_LENGTH,  
CURRENT_MODVERSION));
```

#### **Methods:**

#### **Prototype:**

```
virtual Deformer& GetDeformer(TimeValue t,ModContext  
&mc,Matrix3& mat,Matrix3& invmat)=0;
```

#### **Remarks:**

Implemented by the Plug-In.

This method is used to retrieve the callback object that will handle the deformation.

#### **Parameters:**

##### **TimeValue t**

Specifies the time the modification is being performed.

##### **ModContext &mc**

A reference to the ModContext.

##### **Matrix3& mat**

A reference to a matrix that describes the space the modification is supposed to happen in. This is computed from the ModContext matrix and the controllers controlling the gizmo and center of the modifier. The plug-in developers job is simply to transform each point to be deformed by this matrix before it performs its own deformation to the point. After the modifier applies its own deformation to the point, the developer transforms the point by the

inverse of this matrix (passed below).

### **Matrix3& invmat**

This is the inverse of the matrix above. See the comment above for how this is used.

### **Return Value:**

A C++ reference to the deformer callback object.

See Also: The Advanced Topics section on the [Geometry Pipeline System](#).

### **Prototype:**

**virtual void InvalidateUI()**

### **Remarks:**

Implemented by the Plug-In.

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

### **Example:**

If the plug-in uses a parameter map for handling its UI, it may call a method of the parameter map to handle this: **pmapParam->Invalidate();**

If the plug-in does not use parameter maps, it should call the **SetValue()** method on each of its controls that display a value, for example the spinner controls. This will cause to the control to update the value displayed. The code below shows how this may be done for a spinner control. Note that ip and pblock are assumed to be initialized interface and parameter block pointers

```
float newval;
```

```
Interval valid=FOREVER;
```

```
TimeValue t=ip->GetTime();
```

```
// Get the value from the parameter block at the current time.
```

```
pblock->GetValue( PB_ANGLE, t, newval, valid );
```

```
// Set the value. Note that the notify argument is passed as  
FALSE.
```

```
// This ensures no messages are sent when the value changes.
```

**angleSpin->SetValue( newval, FALSE );**

**Prototype:**

**virtual Interval GetValidity(TimeValue t)**

**Remarks:**

Implemented by the Plug-In.

The SimpleWSMMod class calls this method to retrieve the validity interval of the modifier. The modifier provides this interval by starting an interval at FOREVER and intersecting it with each of its parameters validity intervals.

**Parameters:**

**TimeValue t**

The time to compute the validity interval.

**Default Implementation:**

**{return FOREVER;}**

**Return Value:**

The validity interval of the modifier.

See Also: The Advanced Topics section on [Intervals](#).

**Prototype:**

**virtual ParamDimension \*GetParameterDim(int pbIndex)**

**Remarks:**

Implemented by the Plug-In.

Returns the dimension of the parameter whose index is passed.

**Parameters:**

**int pbIndex**

The index of the parameter.

**Default Implementation:**

**{return defaultDim;}**

**Return Value:**

The dimension of the parameter.

See Also: Class [ParamDimension](#).

**Prototype:**

```
virtual TSTR GetParameterName(int pbIndex)
```

**Remarks:**

Implemented by the Plug-In.

Returns the name of the parameter whose index is passed.

**Parameters:**

```
int pbIndex
```

Index of the parameter.

**Default Implementation:**

```
{return TSTR(_T("Parameter"));} 
```

**Return Value:**

The name of the parameter.

**Prototype:**

```
WSMObject *GetWSMObject(TimeValue t);
```

**Remarks:**

Implemented by the System.

Evaluates the node reference and returns the WSM object. If you look in `\MAXSDK\SAMPLES\HOWTO\MISC\SIMPMOD.CPP` you'll see that all this method does is call **EvalWorldState()** on the Node reference.

**Parameters:**

```
TimeValue t
```

The time to get the WSMObject.

When clients of SimpleWSMMod are cloning themselves, they should call this method on the clone to copy SimpleWSMMod's data.

```
void SimpleWSMModClone(SimpleMod *smodSource);
```

Clients of SimpleWSMMod probably want to override these. If they do they should call these from within their methods.

```
void BeginEditParams(IObjParam *ip, ULONG flags,Animatable *prev);
```

```
void EndEditParams(IObjParam *ip, ULONG flags,Animatable *next);
```

## SimpleWSMObject

See Also: [Class WSMObject](#).

This is a base class to simplify the development of space warp objects. The space warp must represent itself as a mesh in order to subclass off SimpleWSMObject. The main methods are:

### **BuildMesh()**

This method is called to build the mesh representation of the object using its parameter settings at the time passed.

### **InvalidateUI()**

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

### **OKtoDisplay()**

This method returns a BOOL to indicate if it is okay to draw the object at the time passed.

## Class SimpleWSMObject

See Also: [Class WSMObject](#), [Class IParamBlock](#), [Class Mesh](#), [Class Interval](#).

```
class SimpleWSMObject : public WSMObject
```

### Description:

This is the base class for creating space warp objects. This class implements many of the methods required to create a space warp object. The only limitation for a space warp object using SimpleWSMObject as a base class is that it must represent itself with a mesh.

### Data Members:

Note: Methods of the base class refer to these data members. For example the base class implementations of the bounding box methods use the **mesh** data member. Therefore the plug-in derived from SimpleWSMObject must use these same data members. These are listed below:

public:

#### **IParamBlock \*pblock;**

Pointer to a parameter block. Clients of SimpleWSMObject should use this pointer when the pblock reference is created.

#### **Mesh mesh;**

The mesh object that is built by **BuildMesh()**.

#### **Interval ivalid;**

The validity interval of the mesh.

### Methods:

Space warp object plug-ins which subclass off SimpleWSMObject must implement these methods. The default implementations are noted.

### Prototype:

```
virtual void BuildMesh(TimeValue t)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called to build the mesh representation of the object using its parameter settings at the time passed.

### Parameters:

## **TimeValue t**

The time at which to build the mesh.

### **Prototype:**

**virtual ParamDimension \*GetParameterDim(int pbIndex)**

### **Remarks:**

Implemented by the Plug-In.

This method returns the parameter dimension of the parameter whose index is passed.

### **Parameters:**

**int pbIndex**

The index of the parameter to return the dimension of.

### **Return Value:**

Pointer to a ParamDimension.

### **Example:**

**return stdNormalizedDim;**

### **Default Implementation:**

The default implementation returns **defaultDim**.

See Also: [ParamDimension](#)

### **Prototype:**

**virtual TSTR GetParameterName(int pbIndex)**

### **Remarks:**

Implemented by the Plug-In.

This method returns the name of the parameter whose index is passed.

### **Parameters:**

**int pbIndex**

The index of the parameter to return the name of.

### **Return Value:**

The name of the parameter.

### **Default Implementation:**

The default implementation returns **TSTR(\_T("Parameter"))**

**Prototype:**

**virtual void InvalidateUI()**

**Remarks:**

Implemented by the Plug-In.

This is called if the user interface parameters needs to be updated because the user moved to a new time. The UI controls must display values for the current time.

**Example:**

If the plug-in uses a parameter map for handling its UI, it may call a method of the parameter map to handle this: **pmapParam->Invalidate();**

If the plug-in does not use parameter maps, it should call the SetValue() method on each of its controls that display a value, for example the spinner controls. This will cause to the control to update the value displayed. The code below shows how this may be done for a spinner control. Note that ip and pblock are assumed to be initialized interface and parameter block pointers

```
(IObjParam *ip, IParamBlock *pblock).  
float newval;  
Interval valid=FOREVER;  
TimeValue t=ip->GetTime();  
// Get the value from the parameter block at the current time.  
pblock->GetValue( PB_ANGLE, t, newval, valid );  
// Set the value. Note that the notify argument is passed as  
FALSE.  
// This ensures no messages are sent when the value changes.  
angleSpin->SetValue( newval, FALSE );
```

**Prototype:**

**virtual BOOL OKtoDisplay(TimeValue t)**

**Remarks:**

Implemented by the Plug-In.

This method returns a BOOL to indicate if it is okay to draw the object at the

time passed. Normally it is always OK to draw the object, so the default implementation returns TRUE. However for certain objects it might be a degenerate case to draw the object at a certain time (perhaps the size went to zero for example), so these objects could return FALSE.

**Parameters:**

**TimeValue t**

The time at which the object would be displayed.

**Default Implementation:**

```
{ return TRUE; }
```

**Return Value:**

TRUE if the object may be displayed; otherwise FALSE.

## SoundObj

See Also: [Class SoundObj](#).

There is always one sound object in the scene. A sound object's primary purpose is to provide a sound track for the scene, but it also has another important function; it serves as a clock that controls timing when an animation is played. This ensure the animation is synched to the sound object. It has methods to start and stop the animation playing, play a specified range of the sound, and toggle the sound on and off.

# SplineShape

See Also: [Class SplineShape](#).

Spline shape is an implementation of a shape object. A procedural shape can convert itself to a spline shape if it needs to be deformed.

## Class SplineShape

See Also: [Class ShapeObject](#), [Class BezierShape](#), [Working with Shapes and Splines](#).

```
class SplineShape : public ShapeObject, ISplineOps, ISplineSelect,  
ISplineSelectData, ISubMtlAPI, AttachMatDlgUser
```

### Description:

Defines a spline object class. The **SplineShape** is the object that flows down the 3ds max geometry pipeline. This class is a container for the **BezierShape shape**. All methods of this class are implemented by the system.

Spline Shape plug-ins use a Super Class ID of **SHAPE\_CLASS\_ID**.

### Data Members:

public:

**BezierShape shape;**

The shapes of this SplineShape.

### Methods:

#### Prototype:

**SplineShape();**

#### Remarks:

Constructor. Initialize the member variables.

#### Prototype:

**~SplineShape();**

#### Remarks:

Destructor.

#### Prototype:

**BezierShape& GetShape();**

#### Remarks:

Returns the BezierShape data member maintained by this class.

**Prototype:**

**void SelectBySegment(BOOL interactive = TRUE);**

**Remarks:**

This method is available in release 4.0 and later only.

This method operates in vertex level only, and sets the vertex selection set based on the segments that are selected. Any vertex which is part of a selected segment will be selected.

**Parameters:****BOOL interactive**

If set to FALSE, an Undo object is not created and the method does not initiate a redraw.

**Prototype:**

**void SelectBySpline(BOOL interactive = TRUE);**

**Remarks:**

This method is available in release 4.0 and later only.

This method operates in vertex and segment level only, and sets the vertex or segment selection set based on the splines that are selected. If in vertex mode, any vertex which is part of a selected spline will be selected. If in segment mode, any segment which is part of a selected spline will be selected.

**Parameters:****BOOL interactive**

If set to FALSE, an Undo object is not created and the method does not initiate a redraw.

**Prototype:**

**virtual void GetUIParam(splineUIParam uiCode, int &ret);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit spline parameters from the command panel. Currently not in use.

**Parameters:**

**splineUIParam uiCode**

This enum is currently empty.

**int &ret**

The returned value.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void SetUIParam(splineUIParam uiCode, int val);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit spline parameters from the command panel. Currently not in use.

**Parameters:****splineUIParam uiCode**

This enum is currently empty.

**int val**

The value to set.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void GetUIParam(splineUIParam uiCode, float &ret);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit spline parameters from the command panel. Currently not in use.

**Parameters:****splineUIParam uiCode**

This enum is currently empty.

**float &ret**

The returned value.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void SetUIParam(splineUIParam uiCode, float val);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit spline parameters from the command panel. Currently not in use.

**Parameters:**

**splineUIParam uiCode**

This enum is currently empty.

**float val**

The value to set.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
bool Editing();
```

**Remarks:**

This method is available in release 4.0 and later only.

This method will return TRUE if the SplineShape object or Edit Spline modifier is active in the command panel.

**Default Implementation:**

```
{ return (ip && (editObj==this)) ? TRUE : FALSE; }
```

# StdControl

See Also: [Class StdControl](#).

The purpose of this class is to simplify some aspects of implementing controllers. The only restriction when using this class is that the controller can not evaluate itself as a function of its input. This class handles processing Out of Range Types, ease curves and multiplier curves.

There are a few other methods that StdControl requires:

## **Extrapolate()**

Extrapolate is used to calculate some of the ORTs. There are several types of extrapolations that need to be done and there are template functions implemented in `\MAXSDK\INCLUDE\CONTROL.H` to do them.

## **CreateTempValue()**

When processing the ORTs the system might need a temporary variable to hold an intermediate value. Since the system doesn't actual know the type of the data that the controller is controlling it can't allocate the right amount of temporary storage. It calls this method to do so.

## **DeleteTempValue()**

This method simply deletes the memory allocated by **CreateTempValue()**.

## **ApplyValue()**

Applies the given value to the given input value. For position, rotation, and scale controllers, the input value will be a matrix and the value being applied will be a Point3, quaternion, or ScaleValue, respectively. For other controllers the input value is the same type as the value being applied.

## **MultiplyValue()**

If the controller has multiplier curves then the system will calculate the factor from all the multiplier curves and then ask the controller to multiply the scalar value to the particular data type.

# Texmap

See Also: [Class Texmap](#).

Plug-In 2D and 3D texture maps are subclassed off Texmap. Some of the methods of Texmap are:

## **EvalColor()**

This returns the color of the texture map at a specific location.

## **EvalNormalPerturb()**

This computes a perturbation to apply to a normal for bump mapping.

# TriObject

See Also: [Class TriObject](#).

A triangle mesh object. All procedural objects must be able to convert themselves to TriObjects. TriObjects are "deformable". This means they have points which a modifier may operate upon to deform the geometry. This class provides default implementations of many of the methods from Object and GeomObject.

## Class TriObject

See Also: [Class GeomObject](#), [Class ClassDesc](#), [Class Mesh](#), [Class TessApprox](#).

```
class TriObject : public GeomObject
```

### Description:

This class represents a renderable, deformable, triangle mesh object. All procedural objects must be able to convert themselves to TriObjects. This class provides implementations of all the required methods of Animatable, ReferenceMaker, ReferenceTarget, Base Object, Object, and GeomObject. All methods of this class are implemented by the system.

### Data Members:

public:

#### **Mesh mesh;**

This is the mesh of the TriObject. See [Class Mesh](#) for methods to manipulate this mesh.

The following data members are used by the Displacement Mapping mechanism in 3ds max.

#### **TessApprox mDispApprox;**

The object which describes the properties of the tessellation approximation of the mesh.

#### **bool mSubDivideDisplacement;**

The subdivision displacement flag. When TRUE, displacement mapping mechanism subdivides mesh faces to accurately displace the map, using the method and settings you specify in the Subdivision Presets and Subdivision Method group boxes. When FALSE, the modifier applies the map by moving vertices in the mesh, the way the Displace modifier does.

#### **bool mDisableDisplacement;**

TRUE to disable displacement mapping; FALSE to enable it.

#### **bool mSplitMesh;**

The split mesh flag. This flag affects texture mapping as done by the displacement mapping mechanism. When on, the modifier splits the mesh into individual faces before displacing them: this helps preserve texture mapping. When off, the modifier uses an internal method to assign texture mapping. Default=On.

**Methods:**

**Prototype:**

**Mesh& GetMesh();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to the **mesh** data member of this TriObject.

**Prototype:**

**TessApprox& DisplacmentApprox();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to the **mDispApprox** data member.

**Prototype:**

**bool& DoSubdivisionDisplacment();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to the boolean **mSubDivideDisplacement** data member.

**Prototype:**

**bool& SplitMeshForDisplacement();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to the boolean **mSplitMesh** data member.

**Prototype:**

**void SetDisplacmentApproxToPreset(int preset);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used internally to set the **mDispApprox** data member to one

of the low/medium/high subdivision presets.

**Prototype:**

**void DisableDisplacementMapping(BOOL disable);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the **mDisableDisplacement** data member to the given state.

**Parameters:**

**BOOL disable**

TRUE to disable; FALSE to enable.

**Related Functions:**

There are several global functions (not part of class TriObject) for dealing with TriObjects and the control of standard TriObject versus Editable TriObjects. These functions are described below:

**Function:**

**TriObject \*CreateNewTriObject();**

**Remarks:**

This method is used to create a new TriObject. Use this instead of **new TriObject**. It will use the registered descriptor if one is registered, otherwise you'll get a default TriObject.

**Function:**

**ClassDesc\* GetTriObjDescriptor();**

**Remarks:**

Returns a pointer to the class descriptor for the regular, standard TriObject.

**Function:**

**ClassDesc\* GetEditTriObjDesc();**

**Remarks:**

Returns a pointer to the class descriptor for the editable TriObject. It returns the default if none has been registered. See below.

**Function:**

**void RegisterEditTriObjDesc(ClassDesc\* desc);**

**Remarks:**

A new descriptor can be registered to replace the default TriObject descriptor. This new descriptor will then be used to create TriObjects.

**Parameters:**

**ClassDesc\* desc**

The class descriptor to replace the default TriObject descriptor.

# WSMObject

See Also: [Class WSMObject](#).

WSM objects are the gizmos for space warps. WSM stands for World Space Modifier, another name for space warp. They are usually non-renderable by the production renderer but may show up in the viewports to allow the user to orient them in the scene. They implement a single method that they implement:

## **CreateWSMMod()**

When the user binds a node to a space warp, a new modifier must be created and added to the node's WSM derived object. This method creates the new modifier.

## Class WSMObject

See Also: [Class Object](#), [Class SimpleWSMObject](#), [Class ForceField](#), [Class CollisionObject](#).

```
class WSMObject : public Object
```

### Description:

This class is a base class used to derived the helper object for a space warp modifier (WSM Modifier).

World Space Object plug-ins use a Super Class ID of **WSM\_OBJECT\_CLASS\_ID**.

### Methods:

#### Prototype:

```
SClass_ID SuperClassID()
```

#### Remarks:

Implemented by the System.

Returns the super class ID of this plug-in type:

**WSM\_OBJECT\_CLASS\_ID**.

#### Prototype:

```
virtual Modifier *CreateWSMMod(INode *node)=0;
```

#### Remarks:

Implemented by the Plug-In.

When the user binds a node to a space warp, a new modifier must be created and added to the node's WSM derived object. This method creates the new modifier.

#### Parameters:

```
INode *node
```

The node of the WSMObject.

#### Return Value:

A pointer to the new modifier.

**Prototype:**

```
virtual int UsesWireColor();
```

**Remarks:**

This method is available in release 2.0 and later only.

This is a method of **Object**. Below is shown the default implementation provided by this class.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual BOOL NormalAlignVector(TimeValue t,Point3 &pt, Point3  
&norm);
```

**Remarks:**

This method is available in release 2.0 and later only.

This is a method of **Object**. Below is shown the default implementation provided by this class.

**Default Implementation:**

```
{pt=Point3(0,0,0);norm=Point3(0,0,-1);return TRUE;}
```

**Prototype:**

```
virtual BOOL SupportsDynamics();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if spacewarp or collision object supports Dynamics; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual ForceField *GetForceField(INode *node);
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to a **ForceField**. This pointer can be used during dynamics calculations, but should not be hung on to after that. For example, you shouldn't have the pointer long enough for it to be possible for the user to delete the space warp object. When you're done using the **ForceField** call its **DeleteThis()** method. This method may be called several times on the same space warp object with different **INode\*** if it is instanced.

**Parameters:**

**INode \*node**

This is the space warp object's node.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual CollisionObject *GetCollisionObject(INode *node);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the collision object for the WSM. This works just like **GetForceField()** documented above.

**Parameters:**

**INode \*node**

This is the space warp object's node.

**Default Implementation:**

```
{return NULL;}
```

## Class IObjParam

See Also: [Class Interface](#).

class IObjParam : public Interface

### **Description:**

This class is identical to [Class Interface](#). Refer to that section for a description of the methods.

## Class IObjCreate

See Also: [Class Interface](#), [Class IObjParam](#).

```
class IObjCreate : public IObjParam
```

### **Description:**

This class is identical to [Class Interface](#). Refer to that section for a description of the methods.

## Class ViewExp

See Also: [List of Snap Flags](#), [Class Ray](#), [Class Point3](#), [Class IPoint2](#), [Class Matrix3](#), [Class ModContext](#), [Class HitData](#), [Class GraphicsWindow](#), [Class INode](#).

```
class ViewExp : public InterfaceServer
```

### Description:

This class provides methods to access properties of the viewport, convert points between viewport and world coordinates, snap points and lengths, etc. Many methods associated with hit testing are also here. All the methods of this class are implemented by the system.

### Method Groups:

The hyperlinks below jump to the start of groups of related methods within the class:

[Screen Space - World Space Conversion](#)

[Osnap Related Methods](#)

[Perspective/Camera View Properties](#)

[Snapping](#)

[Access to Viewport Properties](#)

[Node Level Hit Testing](#)

[Sub-Object Level Hit Testing](#)

[Controller Gizmo Hit Testing](#)

[AutoGrid Related Methods](#)

### Methods:

## Screen Space - World Space Conversion

### Prototype:

```
virtual float NonScalingObjectSize()=0;
```

### Remarks:

The value returned from this method may be used as a scale factor that will counteract the viewport zoom. For example, lights, cameras, and tape helper objects use this factor so the size of the node in the scene remains constant when the viewport is zoomed in and out.

This value is affected by the 'Non-Scaling Object Size' spinner in the Viewport Preferences dialog, so the user has some control over this as well.

### Sample Code:

This sample is from

**\MAXSDK\SAMPLES\OBJECTS\TAPEHELP.CPP**. The computed matrix is used in several places like displaying, snapping, hit testing, etc.

```
void TapeHelpObject::GetMat(TimeValue t, INode* inode,
ViewExp* vpt, Matrix3& tm)
{
tm = inode->GetObjectTM(t);
tm.NoScale();
float scaleFactor = vpt->NonScalingObjectSize() *
vpt->GetVPWorldWidth(tm.GetTrans())/(float)360.0;
tm.Scale(Point3(scaleFactor,scaleFactor,scaleFactor));
}
```

### Prototype:

```
virtual Point3 GetPointOnCP(const IPoint2 &ps)=0;
```

### Remarks:

Returns a point in world space on the current construction plane based on the specified screen coordinate.

### Parameters:

```
const IPoint2 &ps
```

The 2D screen point to convert to a 3D world space coordinate.

### Return Value:

The world space coordinate on the current construction plane.

### Prototype:

```
virtual float GetCPDisp(const Point3 base, const Point3& dir,
const IPoint2& sp1, const IPoint2& sp2 )=0;
```

### Remarks:

This method returns a length in world space given a start screen point, an end screen point, a base point and a direction vector. For example, when creating a

cylinder, the user clicks the mouse down to define the center point of the cylinder (**base**), then drags out a radius. They then drag out a height for the cylinder. This method is used to return intermediate and final heights for the cylinder based on the initial base point, the direction vector (the Z axis), the start mouse point, and the current point the user is interactively adjusting.

**Parameters:**

**const Point3 base**

Base point in object space.

**const Point3& dir**

Direction vector in object space.

**const IPoint2& sp1**

Screen start point. This is the point where the user clicked down with the mouse.

**const IPoint2& sp2**

Screen end point. This is the point where the user let up the mouse.

**Return Value:**

The length in world space based on the screen points and their projection onto the direction vector.

**Sample Code:**

```
float h = vpt->SnapLength(vpt-  
>GetCPDisp(p[1],Point3(0,0,1),sp1,m));
```

From `\MAXSDK\SAMPLES\OBJECTS\CYL.CPP` in

```
CylinderObjCreateCallBack::proc
```

**Prototype:**

```
virtual Point3 MapScreenToView( IPoint2& sp, float depth )=0;
```

**Remarks:**

Given a point on the screen (in window coordinates), and a depth in view coordinates, this method maps the point into view coordinates. This is just a scaling operation for parallel projections, but involves a divide by Z for perspective projections.

**Parameters:**

**IPoint2& sp**

Point in window coordinates.

**float depth**

Depth in view coordinates.

**Return Value:**

Point in view coordinates.

**Sample Code:**

```
Point3 p0 = vpt-  
>MapScreenToWorldRay(mBase,GetPerspMouseSpeed());
```

**Prototype:**

```
virtual void MapScreenToWorldRay(float sx, float sy, Ray&  
ray)=0;
```

**Remarks:**

Creates a Ray in world space passing through the specified pixel directed toward the scene in the direction of view.

**Parameters:**

**float sx**

The x screen coordinate.

**float sy**

The y screen coordinate.

**Ray& ray**

The Ray in world space. See [Class Ray](#).

**Prototype:**

```
virtual void GetAffineTM(Matrix3& tm)=0;
```

**Remarks:**

This method retrieves the affineTM which transforms from World coordinates to View coordinates. See the sample code below for an example of its use.

**Parameters:**

**Matrix3& tm**

The matrix to hold the affine TM.

**Sample Code:**

```

// This routine returns the view direction from the active viewport.
Point3 Utility::GetViewDirection() {
  Matrix3 aTM, coordSysTM;
  ViewExp *ve = ip->GetActiveViewport();
  // The affine TM transforms from world coords to view coords
  // so we need the inverse of this matrix
  ve->GetAffineTM(aTM);
  coordSysTM = Inverse(aTM);
  // The Z axis of this matrix is the view direction.
  Point3 viewDir = coordSysTM.GetRow(2);
  ip->ReleaseViewport(ve);
  return viewDir;
}

```

Note: You can also get the view position from this matrix. For example, in the above code, the statement:

```
Point3 viewPt = coordSysTM.GetRow(3);
```

gets the point in space the view is taken from.

**Prototype:**

```
virtual BOOL SetAffineTM(const Matrix3& m)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the viewport affine transformation and returns TRUE if the view is a user view (isometric or perspective). If the view is not a user view then the transformation is not changed and the method returns FALSE. See **SetViewUser()** below.

**Parameters:**

```
const Matrix3& m
```

The transformation matrix to set.

**Prototype:**

```
virtual float GetScreenScaleFactor(const Point3 worldPoint)=0;
```

**Remarks:**

Returns the screen scale factor for a point given in world coordinates. This

factor gives the width in world-space units at the point's distance of the viewport.

**Parameters:**

**const Point3 worldPoint**

The point in world coordinates.

**Return Value:**

The screen scale factor in world space units.

**Prototype:**

**virtual float GetVPWorldWidth(const Point3 wPoint)=0;**

**Remarks:**

Returns the viewport screen width factor in world space at a point in world space.

**Parameters:**

**const Point3 wPoint**

The point in world space.

**Return Value:**

The viewport screen width factor in world space.

**Prototype:**

**virtual Point3 MapCPToWorld(const Point3 cpPoint)=0;**

**Remarks:**

Given a point on the construction plane this method returns the corresponding world space point. For example, if you use **GetPointOnCP()** to convert a screen coordinate to a point on the construction plane, you could then call this method to convert that point on the construction plane to a world space point.

**Parameters:**

**const Point3 cpPoint**

The point on the construction plane.

**Return Value:**

The world space point.

## AutoGrid Related Methods

### Prototype:

```
virtual void TrackImplicitGrid(IPoint2 m, Matrix3* mat = NULL,  
    ULONG hitTestFlags = 0)=0;
```

### Remarks:

This method is available in release 3.0 and later only.

If AutoGrid is enabled, this method determines a grid coordinate system by casting a ray into the scene through the screen coordinate **m**, obtaining a surface normal from the closest node, and using the "arbitrary axis algorithm" to orient the xy axes. You can get this coordinate system back by passing in a pointer to a matrix. A tripod is displayed in the viewports showing the orientation.

### Parameters:

#### **IPoint2 m**

The 2D screen point that the user clicked on.

#### **Matrix3\* mat = NULL**

The implicit grid coordinate system matrix can be retrieved by passing a pointer to a matrix here.

#### **ULONG hitTestFlags = 0**

This parameter is available in release 4.0 and later only.

See [Hit Test Flags](#).

### Prototype:

```
virtual void CommitImplicitGrid(IPoint2 m, int mouseflags,  
    Matrix3* mat = NULL)=0;
```

### Remarks:

This method is available in release 3.0 and later only.

If AutoGrid is enabled, this method creates a grid and activates it. The **mouseflags** parameter is used to determine if the ALT key is down. If it is, this grid will not be deactivated in **ReleaseImplicitGrid()** (below).

### Parameters:

#### **IPoint2 m**

The 2D screen point that the user clicked on.

### **int mouseflags**

These flags describe the state of the mouse buttons. See [List of Mouse Callback Flags](#).

### **Matrix3\* mat = NULL**

Developers can get the implicit grid coordinate system back by passing in a pointer to a matrix here.

### **Prototype:**

```
virtual void ReleaseImplicitGrid()=0;
```

### **Remarks:**

This method is available in release 3.0 and later only.

This method deactivates an implicit grid and restores the previously active grid. If the implicit grid was committed with ALT-key held down, then this call does nothing.

## **Osnap Related Methods**

### **Prototype:**

```
virtual void SnapPreview(const IPoint2 &in, IPoint2 &out,  
Matrix3 *plane2d=NULL, DWORD flags=0)=0;
```

### **Remarks:**

This method is available in release 2.0 and later only.

This is a method used as part of the osnap system in 3ds max. It is the method that displays the snap marker in the viewports prior to the first point event. It's really just a call to **SnapPoint()** which returns nothing. This method should be called in response to a **MOUSE\_FREEMOVE** event from any creation or transformation proc which calls **SnapPoint()**. Here's an example creation proc:

### **Sample Code:**

```
int PointHelpObjCreateCallBack::proc(ViewExp *vpt,int msg, int  
point, int flags, IPoint2 m, Matrix3& mat ) {  
if (msg == MOUSE_FREEMOVE)
```

```

// Show a preview snap in the viewport prior to the
// first point event.
vpt->SnapPreview(m,m,NULL, SNAP_IN_3D);

if (msg==MOUSE_POINT||msg==MOUSE_MOVE) {
switch(point) {
case 0:
ob->suspendSnap = TRUE;
mat.SetTrans(vpt->SnapPoint(m,m,NULL,SNAP_IN_3D));
break;
case 1:
mat.SetTrans(vpt->SnapPoint(m,m,NULL,SNAP_IN_3D));
if (msg==MOUSE_POINT) {
ob->suspendSnap = FALSE;
return 0;
}
break;
}
} else
if (msg == MOUSE_ABORT) {
return CREATE_ABORT;
}
return 1;
}

```

**Parameters:**

**const IPoint2 &in**

The 2D screen coordinate to snap.

**IPoint2 &out**

The snapped 2D screen coordinate. This is used if you need to move the mouse position to the snapped location.

**Matrix3 \*plane2d = NULL**

This optional argument allows you to use any plane (not just the current construction plane).

**int flags**

See [List of Snap Flags](#).

**Prototype:**

**virtual void GetGridDims(float \*MinX, float \*MaxX, float \*MinY, float \*MaxY) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used internally. It fills up it's arguments with the world space extents of the home grid (i.e. the extents of the grid as **displayed**). It doesn't work for grid helper which always display to their size limits. This was exposed so 3ds max could do the grid snapping and is not needed by plug-in developers.

## **Perspective/Camera View Properties**

**Prototype:**

**virtual BOOL IsPerspView()=0;**

**Remarks:**

Returns TRUE if the viewport is a perspective view; otherwise returns FALSE.

**Prototype:**

**virtual int GetViewType()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the type of view. One of the following values:

**enum ViewType {**

**VIEW\_LEFT,VIEW\_RIGHT,VIEW\_TOP,VIEW\_BOTTOM,VI  
VIEW\_ISO\_USER, VIEW\_PERSP\_USER, VIEW\_CAMERA,**

**VIEW\_GRID, VIEW\_NONE,  
VIEW\_TRACK, VIEW\_SPOT, VIEW\_SHAPE,  
VIEW\_SCHEMATIC, VIEW\_OTHER  
};**

**Prototype:**

**virtual float GetFOV()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the field of view of a perspective viewport in radians.

**Prototype:**

**virtual float GetFocalDist()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the focal distance of a perspective view.

**Prototype:**

**virtual INode \*GetViewCamera()=0;**

**Remarks:**

Returns the INode pointer of the camera associated with this viewport. If this is not a camera view then NULL is returned.

**Prototype:**

**virtual void SetViewCamera(INode \*camNode)=0;**

**Remarks:**

Set this viewport to a camera view.

**Parameters:**

**INode \*camNode**

The camera node to set.

**Prototype:**

**virtual INode \*GetViewSpot();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the INode pointer of the spotlight associated with this viewport. If this is not a spotlight view then NULL is returned.

**Prototype:**

```
virtual void SetViewSpot(INode *spotNode)=0;
```

**Remarks:**

Set this viewport to a spotlight view.

**Parameters:**

**INode \*spotNode**

The spotlight node to set.

**Prototype:**

```
virtual void SetViewUser(BOOL persp)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the viewport to be a user view, with the **persp** argument indicating whether this should be a perspective or iso view. Note that the user viewport defaults are used for field-of-view, etc.

**Parameters:**

**BOOL persp**

TRUE for perspective; FALSE for isometric.

## Snapping

**Prototype:**

```
virtual Point3 SnapPoint(const IPoint2 &in, IPoint2 &out,  
Matrix3 *plane2d = NULL, int flags = 0)=0;
```

**Remarks:**

Given a 2D screen coordinate, this method returns a 3D point on the current construction plane based on the current snap settings and flags passed.

**Parameters:**

**const IPoint2 &in**

The 2D screen coordinate to snap.

**IPoint2 &out**

The snapped 2D screen coordinate. This is used if you need to move the mouse position to the snapped location.

**Matrix3 \*plane2d = NULL**

This optional argument allows you to use any plane (not just the current construction plane).

**int flags**

See [List of Snap Flags](#).

**Return Value:**

The snapped 3D point in world space.

**Prototype:**

**virtual float SnapLength(float in)=0;**

**Remarks:**

Given the distance passed, this method snaps the length to the nearest snap increment and returns the snapped distance.

**Parameters:**

**float in**

The input distance to be snapped.

**Return Value:**

The snapped distance.

## Access to Viewport Properties

**Prototype:**

**virtual GraphicsWindow \*getGW()=0;**

**Remarks:**

Returns a pointer to the instance of **GraphicsWindow** associated with this viewport.

Note: A **GraphicsWindow** always has a transform associated with it, for faster object-to-screen space conversions. The **GraphicsWindow \*** returned

by this method may have a non-identity transform already in place. A developer can call **gw->setTransform()** with a node's transform for fast work in Display routines. But this value must be explicitly set to the identity for world-to-screen displays.

**Prototype:**

**virtual HWND GetHwnd()=0;**

**Remarks:**

This returns the window handle of the viewport - this is the transparent window that catches mouse input. Note that this window handle is different than the handle that can be retrieved from the viewport's GraphicsWindow. **getGW()->getHwnd()** is the window that things are drawn on.

**Return Value:**

The window handle of the viewport.

**Prototype:**

**virtual BOOL setBkgImageDsp(BOOL onOff)=0;**

**Remarks:**

This method is used to turn on and off the background image display in this viewport. Note that it is necessary to redraw the viewports in order to see the effect of this method. Use the method **Interface::RedrawViews()** to do this.

**Parameters:**

**BOOL onOff**

TRUE to turn the background image on; FALSE to turn it off.

**Return Value:**

TRUE if the image was set; otherwise FALSE.

**Prototype:**

**virtual int getBkgImageDsp()=0;**

**Remarks:**

Returns nonzero if the background image is displayed in this viewport; otherwise 0.

**Prototype:**

**virtual void setSFDisplay(int onOff)=0;**

**Remarks:**

This method may be used to turn the safe frame display on and off in this viewport.

**Parameters:**

**int onOff**

Nonzero to turn on the safe frame; zero to turn it off.

**Prototype:**

**virtual int getSFDisplay()=0;**

**Remarks:**

Returns nonzero if the safe frame is displayed in this viewport; otherwise 0.

**Prototype:**

**virtual int IsWire()=0;**

**Remarks:**

Determines if this viewport is in wire-frame rendering mode (as opposed to a shaded mode).

**Return Value:**

Nonzero if the viewport is in wire-frame rendering mode; otherwise 0.

**Prototype:**

**virtual Rect GetDammageRect()=0;**

**Remarks:**

Returns the damaged rectangle of the viewport. This is the area that needs to be updated on the next screen refresh. This can be used for example, to pass into the **Mesh** method **render()** to only display the damaged area of the object. A developer could also use this in the implementation of their own **Display()** method.

**Sample Code:**

**int SimpleObject::Display(TimeValue t, INode\* inode,**

```

ViewExp *vpt, int flags) {
if (!OKtoDisplay(t)) return 0;
GraphicsWindow *gw = vpt->getGW();
Matrix3 mat = inode->GetObjectTM(t);
UpdateMesh(t); // UpdateMesh() just calls BuildMesh()
gw->setTransform(mat);
mesh.render(gw, inode->Mtls(),
(flags&USE_DAMAGE_RECT) ? &vpt->GetDammageRect() : NULL,
COMP_ALL, inode->NumMtls());
return(0);
}

```

**Prototype:**

**virtual void GetConstructionTM( Matrix3 &tm )=0;**

**Remarks:**

Retrieves the transformation matrix of the construction plane.

**Parameters:**

**Matrix3 &tm**

The transformation matrix is returned here.

**Prototype:**

**virtual void SetGridSize(float size)=0;**

**Remarks:**

Sets the size of the construction grid spacing.

**Parameters:**

**float size**

Specifies the grid spacing.

**Prototype:**

**virtual float GetGridSize()=0;**

**Remarks:**

Returns the construction grid spacing. This is the grid spacing on a per viewport basis. It is dependent on how far zoomed in or out the user is. This is

the exact same value that you can see in the right most status panel below the viewports.

**Prototype:**

**virtual BOOL IsGridVisible()=0;**

**Remarks:**

Returns TRUE if the grid is turned on for this viewport; otherwise FALSE.

**Prototype:**

**virtual int GetGridType()=0;**

**Remarks:**

Returns the grid type. One of the following values (from OBJECT.H):

**GRID\_PLANE\_NONE**

**GRID\_PLANE\_TOP**

**GRID\_PLANE\_LEFT**

**GRID\_PLANE\_FRONT**

**GRID\_PLANE\_BOTTOM**

**GRID\_PLANE\_RIGHT**

**GRID\_PLANE\_BACK**

**Prototype:**

**virtual BOOL IsActive()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the viewport is the active on; otherwise FALSE.

**Prototype:**

**virtual BOOL IsEnabled()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the viewport is enabled; FALSE if disabled.

## For node level hit-testing

### Prototype:

**virtual void ClearHitList()=0;**

### Remarks:

Implemented by the System.

Clears the list of node level hit records.

### Prototype:

**virtual INode \*GetClosestHit()=0;**

### Remarks:

Implemented by the System.

Returns the INode pointer of the node that was the closest of all those hit. If none were hit, NULL is returned.

### Prototype:

**virtual INode \*GetHit(int i)=0;**

### Remarks:

This method is available in release 2.0 and later only.

Returns the INode pointer of the 'i-th' node level hit.

### Parameters:

**int i**

The index of the hit to retrieve.

### Prototype:

**virtual int HitCount()=0;**

### Remarks:

Implemented by the System.

Returns the number of hits recorded by the last node level hit test.

## For sub-object level hit-testing

### Prototype:

**virtual void LogHit(Node \*nr, ModContext \*mc, DWORD dist, ulong info, HitData \*hitdat = NULL)=0;**

**Remarks:**

Implemented by the System.

This method records a sub-object level hit record with the system using the specified parameters. This hit can later be retrieved using the method **GetSubObjHitList()** and the methods of class HitLog.

**Parameters:**

**Node \*nr**

The node that was hit.

**ModContext \*mc**

The ModContext of the modifier.

**DWORD dist**

The 'distance' of the hit. What the distance actually represents depends on the rendering level of the viewport. For wireframe modes, it refers to the distance in the screen XY plane from the mouse to the sub-object component. In a shaded mode, it refers to the Z depth of the sub-object component. In both cases, smaller values indicate that the sub-object component is 'closer' to the mouse cursor.

**ulong info**

Identifies the sub-object component that was hit.

**HitData \*hitdat = NULL**

If the **info** data member is insufficient to indicate the sub-object component that was hit, pass an instance of the **HitData** class that contains the needed information.

**Prototype:**

**virtual HitLog& GetSubObjHitList()=0;**

**Remarks:**

Returns the sub-object hit list. See [Class HitLog](#).

**Prototype:**

**virtual void ClearSubObjHitList()=0;**

**Remarks:**

Clears the sub-object hit list. This deletes all previously saved HitRecords.

**Prototype:**

```
virtual int NumSubObjHits()=0;
```

**Remarks:**

Returns the number of sub-object hits recorded.

**For controller apparatus hit testing.****Prototype:**

```
virtual void CtrlLogHit(INode *nr, DWORD dist, ulong info,  
    DWORD infoExtra)=0;
```

**Remarks:**

This method records a controller sub-object level hit record with the system using the specified parameters. This hit can later be retrieved using the method **GetCtrlHitList()** and the methods of class CtrlHitLog.

**Parameters:****INode \*nr**

The node that was hit.

**DWORD dist**

The 'distance' of the hit. What the distance actually represents depends on the rendering level of the viewport. For wireframe modes, it refers to the distance in the screen XY plane from the mouse to the sub-object component. In a shaded mode, it refers to the Z depth of the sub-object component. In both cases, smaller values indicate that the sub-object component is 'closer' to the mouse cursor.

**ulong info**

A general unsigned long value. Most controllers will just need this to identify the sub-object element. The meaning of this value (how it is used to identify the element) is up to the plug-in.

**DWORD infoExtra;**

If the above **hitInfo** data member is not sufficient to describe the sub-object element this data member may be used as well.

**Prototype:**

**virtual CtrlHitLog& GetCtrlHitList()=0;**

**Remarks:**

Returns the list of controller gizmo hits recorded. See [Class CtrlHitLog](#).

**Prototype:**

**virtual void ClearCtrlHitList()=0;**

**Remarks:**

Clears the controller hit list. This deletes all the HitRecords previously recorded.

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);**

**Remarks:**

This method is available in release 4.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new cmd numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:****int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the cmd option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the cmd option is discussed for more details on the meaning of this value.

## Class ILoad

See Also: [Loading and Saving](#), [Character Strings](#), [Class ISave](#), [Class PostLoadCallback](#), [Class INode](#), [Class ClassDesc](#), [Class ClassDirectory](#), [Class Interface](#).

```
class ILoad : public InterfaceServer
```

### Description:

This class provides methods to load data from disk and to register post load callbacks.

Note: It is not valid, to write two **CStrs** in the same chunk of a 3ds max file, since **ILoad::ReadCStringChunk()** sets the size for the string to the **ChunkSize**. However it is possible to write other data, such as two ints, into the same chunk.

All methods of this class are implemented by the system.

### Methods:

### Prototype:

```
virtual IOResult OpenChunk()=0;
```

### Remarks:

This method is used to open a chunk. If **OpenChunk()** returns **IO\_OK**, use the following 3 functions to get the info about the chunk. If it returns **IO\_END** this indicates there are no more chunks at this level.

### Return Value:

**IO\_OK** - The result was acceptable - no errors.

**IO\_END** - This is returned from **ILoad::OpenChunk()** when the end of the chunks at a certain level have been reached. It is used as a signal to terminate the processing of chunks at that level.

**IO\_ERROR** - This is returned if an error occurred. Note that the plug-in should not put up a message box if a read error occurred. It should simply return the error status. This prevents a overabundance of messages from appearing.

### Prototype:

**virtual USHORT CurChunkID()=0;**

**Remarks:**

This method returns the ID of the most recently opened chunk.

**Prototype:**

**virtual ChunkType CurChunkType()=0;**

**Remarks:**

This method returns the type of the most recently opened chunk. This may be one of the following values:

**NEW\_CHUNK**

**CONTAINER\_CHUNK**

**DATA\_CHUNK**

**Prototype:**

**virtual ULONG CurChunkLength()=0;**

**Remarks:**

This method returns the chunk length NOT including the header.

**Prototype:**

**virtual int CurChunkDepth()=0;**

**Remarks:**

This method is used internally for checking for balanced OpenChunk/CloseChunk pairs.

**Prototype:**

**virtual USHORT PeekNextChunkID()=0;**

**Remarks:**

This method returns the ID of the next chunk without opening it. It returns 0 if there are no more chunks.

**Prototype:**

**virtual IOResult CloseChunk()=0;**

**Remarks:**

This method is used to close the currently opened chunk, and position at the next chunk.

**Return Value:**

A return value of **IO\_ERROR** indicates there is no open chunk to close; otherwise **IO\_OK**.

**Prototype:**

```
virtual IOResult Read(void *buf, ULONG nbytes, ULONG *nread)=0;
```

**Remarks:**

This method is used to read a block of bytes from the output stream.

**Parameters:**

**void \*buf**

A pointer to the buffer to read.

**ULONG nbytes**

The number of bytes to read.

**ULONG \*nread**

The number of bytes that were read.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

```
virtual IOResult ReadWStringChunk(char** buf)=0;
```

**Remarks:**

This method read a string that was stored as Wide characters. Note: This method reads a string from a string chunk. It is assumed the chunk is already open, it will NOT close the chunk.

**Parameters:**

**char\*\* buf**

A pointer to an array of characters.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

```
virtual IOResult ReadWStringChunk(wchar_t** buf)=0;
```

**Remarks:**

This method read a string that was stored as Wide chars. Note: This method reads a string from a string chunk. It is assumed the chunk is already open, it will NOT close the chunk.

**Parameters:**

**wchar\_t\*\* buf**

A pointer to an array of wide characters.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

```
virtual IOResult ReadCStringChunk(char** buf)=0;
```

**Remarks:**

This method reads a string that was stored as single byte characters.

**Parameters:**

**char\*\* buf**

A pointer to an array of single byte characters. This method will allocate an internal buffer, stored in the ILoadImp class that is big enough to hold the string chunk read in. You must then copy or parse out the data and store it in your own area: you can't hang on to the string pointer it hands back because it will not be valid.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

**virtual IOResult ReadCStringChunk(wchar\_t\*\* buf)=0;**

**Remarks:**

This method may be used to read a string that was stored as single byte characters.

**Parameters:**

**wchar\_t\*\* buf**

A pointer to an array of wide characters. This method will allocate an internal buffer, stored in the ILoadImp class that is big enough to hold the string chunk read in. You must then copy or parse out the data and store it in your own area: you can't hang on to the string pointer it hands back because it will not be valid.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

**virtual void SetObsolete()=0;**

**Remarks:**

You may call this if you encounter obsolete data to cause a message to be displayed after loading.

**Prototype:**

**virtual void RegisterPostLoadCallback(PostLoadCallback \*cb)=0;**

**Remarks:**

Registers a procedure to be called after loading. These will be called in the order that they are registered. It is assumed that if the callback needs to be deleted, the proc will do it.

**Parameters:**

**PostLoadCallback \*cb**

Points to the callback object.

**Prototype:**

**virtual TCHAR \*GetDir(int which)=0;**

**Remarks:**

Retrieves the specified standard 3ds max directory name (fonts, scenes, images, ...).

**Parameters:**

**int which**

Specifies the directory name to retrieve. See [List of Directory Names](#). The constants are defined in **MAXAPI.H**

**Return Value:**

The name of the specified directory.

**Prototype:**

**virtual void RecordBackpatch(int imaker, void\*\* patchThis, DWORD flags = 0)=0;**

**Remarks:**

This method may be used to load a pointer from disk. This is a pointer that was saved using **ISave::GetRefID()**. You pass the index returned from **GetRefID()** and a pointer to a pointer that will get set. This method will patch the address immediately if it is available, otherwise it will happen later when it is known. During the load process if you need to work with this information you'll have to use a post load callback since all the addresses are not updated immediately. See **RegisterPostLoadCallback()** above.

**Parameters:**

**int imaker**

This is the index returned from **ISave::GetRefID()**.

**void\*\* patchThis**

This is a pointer to the pointer you want patched.

**DWORD flags = 0**

This flag indicates that backpatches (and their subsequent references) should be merged as well.

**Prototype:**

**virtual void\* GetAddr(int imaker)=0;**

**Remarks:**

This method may be used to load a pointer from disk. It returns the memory address of the specified object Scene stream. This may be NULL if the address is not available. See **RecordBackpatch()** above for a work around.

**Parameters:****int imaker**

This is the index returned from **ISave::GetRefID()**. that was used to save the pointer.

**Prototype:****virtual void SetRootAddr(void \*addr)=0;****Remarks:**

This method is used internally.

**Prototype:****virtual void\* GetRootAddr()=0;****Remarks:**

This method is used internally.

**Prototype:****virtual TCHAR \*GetDir(int which)=0;****Remarks:**

Returns the name of various directories used by 3ds max. See [List of Directory Names](#).

**Prototype:****virtual FileIOType DoingWhat()=0;****Remarks:**

Determines if we are loading a standard 3ds max file (**.MAX**) or a material library (**.MAT**).

**Return Value:**

One of the following values:

**IOTYPE\_MAX**  
**IOTYPE\_MATLIB**

**Prototype:**

**virtual INode \*RootNode()=0;**

**Remarks:**

Returns the root node to attach to when loading a node with no parent.

**Prototype:**

**virtual ClassDesc\* GetClassDesc(USHORT refID);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the ClassDesc corresponding to the specified reference ID in the ClassDirectory stream.

**Parameters:**

**USHORT refID**

The reference ID in the ClassDirectory stream.

**Default Implementation:**

**{ return NULL; }**

**Prototype:**

**virtual DWORD GetFileSaveVersion();**

**Remarks:**

This method is available in release 4.0 and later only.

This function is available to plug-ins when loading files. It returns a value describing the version of 3ds max used to save the file. The

value is composed of: **LOWORD(value)** = Build number. For example 41 for build #41. **HIWORD(value)** = **MAX\_RELEASE** (defined as 3ds max release version \* 1000), thus 3ds max version 4.0 is 4000

**Return Value:**

This function returns 0 if the file does not contain this value.

**Default Implementation:**

```
{ return 0; }
```

## Class ISave

See Also: [Loading and Saving](#), [Character Strings](#), [Class ILoad](#), [Class ClassDesc](#), [Class ClassDirectory](#), [Class Interface](#).

```
class ISave : public InterfaceServer
```

### Description:

This class provides methods to save data to disk.

Note: It is not valid, to write two **CStrs** in the same chunk of a 3ds max file, since **ILoad::ReadCStringChunk()** sets the size for the string to the **ChunkSize**. However it is possible to write other data, such as two ints, into the same chunk.

Note about member alignment: Please make sure that when you save data from your plugin you save individual data members using a chunk ID instead of saving the image of a class. Saving (and loading) a class image puts you at risk of running into member alignment problems and as such could potentially corrupt saved files. File IO would be put further at risk when you keep Intel's IA-64 architecture in mind which depends on member alignment. What you should not do is outlined in the following example when loading a class image; **iload->Read(&myclass, sizeof(MyClass), &ab);**

Once you change the class in such a way that it affects the data size you run the risk of having to support different versions, file IO incompatibility, and member alignment issues.

The following global function is not part of this class but is available for use:

### Function:

**DWORD GetSavingVersion();**

### Remarks:

This function is available in release 3.0 and later only.

This function is reserved for future use as of 3ds max 4.

This global function is used to find out if we are saving an old version .3ds max file. If this returns 0, then either we are not in a save or we are saving the current version. If it returns non-zero, it is the max release number being saved, multiplied by 1000. Thus, when saving 3ds max R2 files, it will return 2000. This function can be used in NumRefs() and GetRef() to make an objects references appear as they did in the old 3ds max version.

**Methods:**

All methods of this class are implemented by the system.

**Prototype:**

**virtual void BeginChunk(USHORT id) =0;**

**Remarks:**

This method is used to begin a chunk. The ID passed need only be unique within the plug-ins data itself.

**Parameters:**

**USHORT id**

The id for the chunk.

**Prototype:**

**virtual void EndChunk()=0;**

**Remarks:**

This method is used to end a chunk, and back-patch the length.

**Prototype:**

**virtual int CurChunkDepth()=0;**

**Remarks:**

This method is used internally for checking balanced BeginChunk/EndChunk.

**Prototype:**

**virtual IOResult Write(const void \*buf, ULONG nbytes, ULONG \*nwrite)=0;**

**Remarks:**

This method writes a block of bytes to the output stream.

**Parameters:**

**const void \*buf**

The buffer to write.

**ULONG nbytes**

The number of bytes to write.

## **ULONG \*nwrit**

The number of bytes actually written.

### **Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred. Note that the plug-in should not put up a message box if a write error occurs. It should simply return the error status. This prevents a overabundance of messages from appearing.

### **Prototype:**

```
virtual IOResult WriteWString(const char *str)=0;
```

### **Remarks:**

This method is used to write wide character strings.

### **Parameters:**

**const char \*str**

The string to write.

### **Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

### **Prototype:**

```
virtual IOResult WriteWString(const wchar_t *str)=0;
```

### **Remarks:**

This method is used to write wide character strings.

### **Parameters:**

**const wchar\_t \*str**

The string to write.

### **Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

### **Prototype:**

**virtual IOResult WriteCString(const char \*str)=0;**

**Remarks:**

This method is used to write single byte character strings.

**Parameters:**

**const char \*str**

The string to write.

**Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**virtual IOResult WriteCString(const wchar\_t \*str)=0;**

**Remarks:**

This method is used to write single byte character strings.

**Parameters:**

**const wchar\_t \*str**

The string to write.

**Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**virtual int GetRefID(void \*ptarg)=0;**

**Remarks:**

This method is not normally used because the reference hierarchy is saved automatically. In certain cases however this method is quite useful. This method is used in saving a pointer to some object (or a table of pointers). This is a pointer to one of the objects that the scene saves with the reference hierarchy, but it is not a pointer that itself is a reference. For example the Ring Array plug-in uses this when it saves its table of nodes in the ring array. To save these pointers you can call this method and it will return an ID that you can save to disk. Then when you load this ID from disk you can call

**ILoad::RecordBackpatch()** and get back a pointer.

**Parameters:**

**void \*ptarg**

The pointer to save.

**Return Value:**

The id that may be saved to disk.

**Prototype:**

**virtual FileIOType DoingWhat()=0;**

**Remarks:**

Determines if we are saving a standard 3ds max file (**.MAX**) or a material library (**.MAT**).

**Return Value:**

One of the following values:

**IOTYPE\_MAX**

**IOTYPE\_MATLIB**

**Prototype:**

**virtual DWORD SavingVersion();**

**Remarks:**

This method is available in release 3.0 and later only.

This function is reserved for future use as of 3ds max 4.

This version returns a value to indicate the current version of the file being saved. It returns 0 0 for the current version, 2000 for version 2.0, 3000 for 3.0, etc. This basically duplicates the global function **GetSavingVersion()**. In general, Save routines need not to be concerned that they are saving chunks types that are unknown to the old version, because they will be skipped on load, but there may be cases where the Save routine needs to do things differently.

**Default Implementation:**

**{ return 0; }**

**Prototype:**

```
virtual USHORT GetClassDescID(ReferenceMaker* rm);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method returns a load reference ID for the given Reference Maker's ClassDesc object in the ClassDirectory stream.

**Parameters:**

**ReferenceMaker\* rm**

Points to the reference maker.

**Default Implementation:**

```
{ return 0xffff; }
```

## Class IScene

See Also: [Class ITreeEnumProc](#), [Class INode](#).

class IScene

### Description:

Methods of this class may be used to enumerate the scene and to flag certain nodes in the scene. Nodes chosen by the plug-in may be flagged using the **EnumTree()** method. Selected nodes may be flagged using **FlagFGSelected()**. Animated nodes may be flagged using **FlagFGAnimated()** and dependent nodes may be flagged using **FlagFGDependent()**.

### Methods:

#### Prototype:

```
virtual int EnumTree( ITreeEnumProc *proc )=0;
```

#### Remarks:

Implemented by the System.

This may be called to enumerate every INode in the scene. The callback may flag any of these nodes (using **INode::FlagForeground()**).

#### Parameters:

**ITreeEnumProc \*proc**

This callback object is called once for each INode in the scene.

#### Return Value:

Nonzero if the process was aborted by the callback (**TREE\_ABORT**); otherwise 0.

#### Prototype:

```
virtual void FlagFGSelected( TimeValue t )=0;
```

#### Remarks:

Implemented by the System.

Flags all selected nodes in the scene.

#### Parameters:

**TimeValue t**

The time to flag the nodes.

**Prototype:**

```
virtual void FlagFGAnimated( TimeValue t )=0;
```

**Remarks:**

Implemented by the System.

Flags all animated nodes in the scene.

**Parameters:****TimeValue t**

The time to flag the nodes.

**Prototype:**

```
virtual void FlagFGDependent( TimeValue t, BaseObject *obj )=0;
```

**Remarks:**

Implemented by the System.

Flags nodes that are dependent on the given object.

**Parameters:****TimeValue t**

The time to flag the nodes.

**BaseObject \*obj**

The object whose dependent nodes should be flagged.

## Class IMtlParams

See Also: [Class Mtl](#), [Working with Materials](#), [Class IRollupWindow](#).

class IMtlParams : public InterfaceServer

### Description:

This is the interface that is passed in to the material or texture map when it is in the material editor. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual void MtlChanged()=0;**

#### Remarks:

This method may be called to causes the viewports to be redrawn. It should be called when any parameter that affects the look of the material in the viewport has been altered. If the material is not on a visible node in a shaded view, nothing will happen. This method should NOT be called as a spinner is being dragged, but only upon release of the mouse button.

#### Prototype:

**virtual HWND AddRollupPage(HINSTANCE hInst, TCHAR \*dlgTemplate, DLGPROC dlgProc, TCHAR \*title, LPARAM param=0,DWORD flags=0, int category = ROLLUP\_CAT\_STANDARD)=0;**

#### Remarks:

This method may be called to add a rollup page to the material editor dialog. It returns the window handle of the dialog that makes up the rollup page.

#### Parameters:

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**TCHAR \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here. This may be later retrieved using the `GetWindowLong()` call from the Windows API.

**DWORD flags=0**

The following flag is defined:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The window handle of the rollup page.

**Prototype:**

```
virtual HWND AddRollupPage(HINSTANCE hInst,  
DLGTEMPLATE *dlgTemplate, DLGPROC dlgProc, TCHAR  
*title, LPARAM param=0, DWORD flags=0, int category =  
ROLLUP_CAT_STANDARD)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method may be called to add a rollup page to the material editor dialog. It returns the window handle of the dialog that makes up the rollup page. This method is currently not used.

**Parameters:**

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**DLGTEMPLATE \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here. This may be later retrieved using the GetWindowLong() call from the Windows API.

**DWORD flags=0**

The following flag is defined:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the

**RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The window handle of the rollup page.

**Prototype:**

```
virtual HWND ReplaceRollupPage(HWND hOldRollup,  
HINSTANCE hInst, TCHAR *dlgTemplate, DLGPROC dlgProc,  
TCHAR *title, LPARAM param=0,DWORD flags=0, int category  
= ROLLUP_CAT_STANDARD)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is used to replace the rollup page whose index is passed.

**Parameters:****HWND hOldRollup**

The handle to the rollup to replace.

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**TCHAR \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here.

**DWORD flags=0**

Append rollup page flags:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be

displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USERREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The handle of the replacement page is returned.

**Prototype:**

**virtual HWND ReplaceRollupPage(HWND hOldRollup, HINSTANCE hInst, DLGTEMPLATE \*dlgTemplate, DLGPROC dlgProc, TCHAR \*title, LPARAM param=0, DWORD flags=0, int category = ROLLUP\_CAT\_STANDARD)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used to replace the rollup page whose index is passed. This method is currently not being used.

**Parameters:**

**HWND hOldRollup**

The handle to the rollup to replace.

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**DLGTEMPLATE \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here.

**DWORD flags=0**

Append rollup page flags:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The handle of the replacement page is returned.

**Prototype:**

```
virtual void DeleteRollupPage( HWND hRollup )=0;
```

**Remarks:**

This method may be called to remove a rollup page and destroy it.

**Parameters:****HWND hRollup**

The handle of the rollup window. This is the handle returned from **AddRollupPage()**.

**Prototype:**

```
virtual void RollupMouseMessage(HWND hDlg, UINT message,  
    WPARAM wParam, LPARAM lParam )=0;
```

**Remarks:**

This method allows hand cursor scrolling when the user clicks the mouse in an unused area of the dialog. When the user mouses down in dead area of the material editor, the plug-in should pass mouse messages to this function.

Note: In 3ds max 2.0 and later only use of this method is no longer required -- the functionality happens automatically.

**Parameters:****HWND hDlg**

The window handle of the dialog.

**UINT message**

The message sent to the dialog proc.

**WPARAM wParam**

Passed in to the dialog proc. Pass along to this method.

**LPARAM lParam**

Passed in to the dialog proc. Pass along to this method.

**Example:**

```
case WM_LBUTTONDOWN: case WM_LBUTTONUP: case  
WM_MOUSEMOVE:  
    im->iMtlParams->  
    >RollupMouseMessage(hDlg,message,wParam,lParam);
```

**Prototype:**

```
virtual int IsRollupPanelOpen(HWND hwnd)=0;
```

**Remarks:**

Returns nonzero if the rollup page whose handle is passed is open; otherwise zero.

**Parameters:****HWND hwnd**

The window handle of the rollup page to check.

**Prototype:**

```
virtual int GetRollupScrollPos()=0;
```

**Remarks:**

Returns the rollup scroll position. This is used, for example, by the Standard material because it saves and restores the rollup page positions with the material. This is just a convenience for the user.

**Prototype:**

```
virtual void SetRollupScrollPos(int spos)=0;
```

**Remarks:**

This method may be called to set the rollup scroll position. If the position was previously saved, this method may be called to restore it. The Standard material uses this method because it saves and restores the rollup positions with the material as a convenience for the user.

**Parameters:**

**int spos**

Specifies the rollup scroll position.

**Prototype:**

```
virtual void RegisterTimeChangeCallback(TimeChangeCallback *tc)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Registers a callback object that will get called every time the user changes the 3ds max frame slider.

**Parameters:**

**TimeChangeCallback \*tc**

The callback called when the time changes. See [Class TimeChangeCallback](#).

**Prototype:**

```
virtual void
```

```
UnRegisterTimeChangeCallback(TimeChangeCallback *tc)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method un-registers the time change callback.

**Parameters:**

**TimeChangeCallback \*tc**

The callback called when the time changes. See [Class TimeChangeCallback](#).

**Prototype:**

**virtual void RegisterDlgWnd( HWND hDlg )=0;**

**Remarks:**

This is called automatically in **AddRollupPage()**, so a plug-in doesn't need to do it. It is now obsolete.

**Parameters:**

**HWND hDlg**

The handle of the dialog window.

**Prototype:**

**virtual int UnRegisterDlgWnd( HWND hDlg )=0;**

**Remarks:**

This method is called automatically and is now obsolete.

**Parameters:**

**HWND hDlg**

The handle of the dialog window.

**Prototype:**

**virtual TimeValue GetTime()=0;**

**Remarks:**

Returns the current time (the frame slider position).

**Return Value:**

The current time.

Pick an object from the scene

**Prototype:**

**virtual void SetPickMode(PickObjectProc \*proc)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Calling this methods puts the user into a pick mode where they can select items in the scene.

**Parameters:**

**PickObjectProc \*proc**

This is the callback object for the picking. Its methods allow for filtering the picks, changing cursors over valid hits, and allowing multiple picks. See [Class PickObjectProc](#).

**Prototype:**

**virtual void EndPickMode()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to terminate the pick mode.

**Prototype:**

**virtual IRollupWindow \*GetMtlEditorRollup()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns an interface to Materials Editor rollup.

## Class ITrackBar

See Also: [Class IKeyControl](#), [Class Interface](#).

class ITrackBar

### Description:

This class is available in release 3.0 and later only.

The track bar offers a quick way to manipulate keyframes for selected objects. Keys are displayed on the track bar just like they are in Track View. Developers are able to manipulate the track bar using this class. There is a method of class **Interface** which returns a pointer to an instance of this class:

**ITrackBar\* GetTrackBar().**

All methods of this class are Implemented by the System.

### Methods:

public:

### Prototype:

**virtual void SetVisible(BOOL bVisible) = 0;**

### Remarks:

Sets the visibility of the track bar to on or off.

### Parameters:

**BOOL bVisible**

TRUE to make visible; FALSE to hide.

### Prototype:

**virtual BOOL IsVisible() = 0;**

### Remarks:

Returns TRUE if the track bar is visible; otherwise FALSE.

### Prototype:

**virtual void SetFilter(UINT nFilter) = 0;**

### Remarks:

The track bar shows keys for all parametric animation as well as transforms.

This method sets the filter used which determines which keys are shown in the track bar.

**Parameters:**

**UINT nFilter**

One of the following values:

**TRACKBAR\_FILTER\_ALL**

Specifies to show all keys.

**TRACKBAR\_FILTER\_TMONLY**

Specifies to show transform keys only.

**TRACKBAR\_FILTER\_CURRENTTM**

Specifies to only show keys for the currently active transform (move, rotate or scale).

**TRACKBAR\_FILTER\_OBJECT**

Specifies to show keys for the controllers assigned anywhere in the pipeline to be included in the TrackBar key-display. This is essentially the modifiers and the base objects for the selected nodes (no transforms or materials).

**TRACKBAR\_FILTER\_MATERIAL**

Specifies to show keys for the controllers assigned anywhere for the selected nodes material - all controllers in the whole material tree will be included in the track bar display.

**Prototype:**

**virtual UINT GetFilter() = 0;**

**Remarks:**

Returns the filter value which determines which keys are shown in the track bar. See the list specified in the method above for details.

**Prototype:**

**virtual TimeValue GetNextKey(TimeValue tStart, BOOL bForward) = 0;**

**Remarks:**

This method returns the time of the next key given a start time and a flag

which indicates if the search should proceed forward or backwards. This is similar to the behavior of **Animatable::GetNextKeyTime()**.

**Parameters:**

**TimeValue tStart**

Specifies the time to start looking for the next key.

**BOOL bForward**

TRUE to return the time of the next key; FALSE for the previous key.

**Return Value:**

The time of the next (or previous) key.

**Prototype:**

**virtual void RedrawTrackBar(bool bForce = false) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Redraws the Track Bar if required an optionally forces a redraw (even if not known to be required).

**Parameters:**

**bool bForce = false**

Pass true to force a redraw; false to only redraw if 3ds max deems it required.

**Prototype:**

**virtual void SetShowFrames(bool b) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Controls the visibility of frame numbers in the Track Bar.

**Parameters:**

**bool b**

Pass true to show frame numbers; false to turn them off.

**Prototype:**

**virtual bool GetShowFrames() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns true if frame numbers are visible in the Track Bar; false if not.

**Prototype:**

**virtual void SetShowAudio(bool b) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Controls the visibility of the audio track.

**Parameters:**

**bool b**

Pass true to show the audio track; false to turn it off.

**Prototype:**

**virtual bool GetShowAudio() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns true if the audio track is displayed; false if it is not.

**Prototype:**

**virtual void SetShowSelectionRange(bool b) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Controls if the selection range bar is visible or not.

**Parameters:**

**bool b**

Pass true to display the selection range bar; false to turn it off.

**Prototype:**

**virtual bool GetShowSelectionRange() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns true if the selection range bar is visible; false if it is not.

**Prototype:**

**virtual void SetSnapToFrames(bool b) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Controls the snap to frames setting.

**Parameters:**

**bool b**

Pass true to turn it on; false to turn it off.

**Prototype:**

**virtual bool GetSnapToFrames() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns true if snap to frames is on; false if it is off.

**Prototype:**

**virtual void SetKeyTransparency(int xp) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the transparency of keyframes displayed in the trackbar.

**Parameters:**

**int xp**

The transparency value between 0 and 255.

**Prototype:**

**virtual int GetKeyTransparency() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the transparency of keyframes displayed in the trackbar.

**Return Value:**

The transparency value between 0 and 255.

**Prototype:**

```
virtual void SetSelKeyTransparency(int xp) = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the transparency of selected keyframes displayed in the trackbar.

**Parameters:**

**int xp**

The transparency value between 0 and 255.

**Prototype:**

```
virtual int GetSelKeyTransparency() = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the transparency of selected keyframes displayed in the trackbar.

**Return Value:**

The transparency value between 0 and 255.

**Prototype:**

```
virtual void SetCursorTransparency(int xp) = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the transparency of the cursor displayed in the trackbar.

**Parameters:**

**int xp**

The transparency value between 0 and 255.

**Prototype:**

```
virtual int GetCursorTransparency() = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the transparency of the cursor displayed in the trackbar.

**Return Value:**

The transparency value between 0 and 255.

## Class IXRefObject

See Also: [Class Object](#), [Class INode](#).

class IXRefObject: public Object

### Description:

This class is available in release 3.0 and later only.

This class is an interface to the parameters of an XRef object. Object XRefs are derived from this class.

If you have an object pointer you can check its super class ID and class ID to see if it is an XRef object. Then you can cast it to an instance of this class and call these methods. For example:

```
INode *node = ip->GetSelNode(0);  
Object *obj = node->GetObjectRef();  
if (obj->SuperClassID()==SYSTEM_CLASS_ID && obj-  
                >ClassID()==Class_ID(XREFOBJ_C  
                {  
    IXRefObject *ix = (IXRefObject *)obj;  
    ix->BrowseFile(FALSE);  
}
```

However you may have an object that depends on an XRef object (for example a Boolean with an XRef object as one of its operands) or you may have a pointer to the object that the XRef object references. In such cases the only way to know for sure is to look up and down the pipeline for XRef objects.

### Methods:

public:

### Prototype:

```
virtual void Init(TSTR &fname, TSTR &oname, Object *ob,  
    BOOL asProxy=FALSE)=0;
```

### Remarks:

This method initializes a newly created XRef object. The caller provides the name of the file (fname), the name of the object in the file (**oname**), and a pointer to the object being XRefed (**ob**).

**Parameters:**

**TSTR &fname**

The file name is set to this string.

**TSTR &oname**

The object name is set to this string.

**Object \*ob**

Points to the object being XRefed.

**BOOL asProxy=FALSE**

If TRUE then the above information is considered to specify the proxy portion of the XRef.

**Prototype:**

```
virtual void SetFileName(TCHAR *name, BOOL proxy=FALSE,  
BOOL update=TRUE)=0;
```

**Remarks:**

Sets the File name or the Proxy file name depending on the value passed.

**Parameters:**

**TCHAR \*name**

The name to set.

**BOOL proxy=FALSE**

TRUE to set the Proxy file name; FALSE for the XRef file name.

**BOOL update=TRUE**

TRUE to update the scene; FALSE to not update immediately.

**Prototype:**

```
virtual void SetObjName(TCHAR *name, BOOL  
proxy=FALSE)=0;
```

**Remarks:**

Sets the Object name or the Proxy name depending on the value passed.

**Parameters:**

**TCHAR \*name**

The name to set.

**BOOL proxy=FALSE**

TRUE to set the Proxy name; FALSE to set the Object name.

**Prototype:**

**virtual void SetUseProxy(BOOL onOff,BOOL redraw=TRUE)=0;**

**Remarks:**

Sets the state of the Use Proxy choice and optionally redraws the viewports.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**BOOL redraw=TRUE**

TRUE to redraw the viewports; otherwise FALSE.

**Prototype:**

**virtual void SetRenderProxy(BOOL onOff)=0;**

**Remarks:**

Sets the state of the Render Proxy option.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetUpdateMats(BOOL onOff)=0;**

**Remarks:**

Sets the state of the Update Materials option.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetIgnoreAnim(BOOL onOff,BOOL redraw=TRUE)=0;**

**Remarks:**

Sets the state of the Ignore Animation choice and optionally redraws the viewports.

**Parameters:****BOOL onOff**

TRUE for on; FALSE for off.

**BOOL redraw=TRUE**

TRUE to redraw the viewports; otherwise FALSE.

**Prototype:**

**virtual TSTR GetFileName(BOOL proxy=FALSE)=0;**

**Remarks:**

Returns the File name or the Proxy file name depending on the value passed.

**Parameters:****BOOL proxy=FALSE**

TRUE to return the Proxy file name; FALSE for the XRef file name.

**Prototype:**

**virtual TSTR GetObjName(BOOL proxy=FALSE)=0;**

**Remarks:**

Returns the Object name or the Proxy name depending on the value passed.

**Parameters:****BOOL proxy=FALSE**

TRUE to return the Proxy name; FALSE to return the Object name.

**Prototype:**

**virtual TSTR &GetCurFileName()=0;**

**Remarks:**

Returns the XRef File Name.

**Prototype:**

**virtual TSTR &GetCurObjName()=0;**

**Remarks:**

Returns the XRef Object Name.

**Prototype:**

**virtual BOOL GetUseProxy()=0;**

**Remarks:**

Returns TRUE if the Use Proxy option is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetRenderProxy()=0;**

**Remarks:**

Returns TRUE if the Render Proxy option is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetUpdateMats()=0;**

**Remarks:**

Returns TRUE if the Update Material option is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetIgnoreAnim()=0;**

**Remarks:**

Returns TRUE if the Ignore Animation option is on; otherwise FALSE.

**Prototype:**

**virtual void BrowseObject(BOOL proxy)=0;**

**Remarks:**

This method brings up the browse object (Merge) dialog.

**Parameters:**

**BOOL proxy**

TRUE to have Use Proxy mode active; FALSE to have it inactive.

**Prototype:**

**virtual void BrowseFile(BOOL proxy)=0;**

**Remarks:**

This method brings up the browse file (Open File) dialog.

**Parameters:**

**BOOL proxy**

TRUE to have Use Proxy mode active; FALSE to have it inactive.

**Prototype:**

**virtual void ReloadXRef()=0;**

**Remarks:**

Reloads this Object XRef.

## Class ILayer

See Also: [Class ReferenceTarget](#), [Class ILayerManager](#), [Class LayerProperty](#), [Class INode](#).

```
class ILayer : public ReferenceTarget
```

### Description:

This class is available in release 3.0 and later only.

This class is an interface to the Layers functionality provided by 3D Studio VIZ. Basically, Layers govern (override) some properties of the nodes which are on the layer. For example, a Layer could be used to freeze all the nodes on it without having to set this property of each node individually.

Some of the methods below are not functional in 3ds max. Such cases are noted in the remarks for the method.

### Data Members:

public:

```
static const SClass_ID kLayerSuperClassID;
```

The super class ID of the layer interface class.

### Methods:

public:

### Prototype:

```
virtual bool AddToLayer(INode *rtarg)=0;
```

### Remarks:

Adds the specified node to this layer.

### Parameters:

```
INode *rtarg
```

The node to add.

### Prototype:

```
virtual bool DeleteFromLayer(INode *rtarg)=0;
```

### Remarks:

Deletes the specified node from this layer. Note: This method does nothing in 3ds max.

**Parameters:**

**INode \*rtarg**

The node to delete from this layer.

**Prototype:**

**virtual void SetName(const TSTR &name)=0;**

**Remarks:**

Sets the name of this layer.

**Parameters:**

**const TSTR &name**

The name for this layer.

**Prototype:**

**virtual TSTR GetName() const=0;**

**Remarks:**

Returns the name of this layer. Note: The user of this method must delete the returned string.

**Prototype:**

**virtual void SetWireColor(DWORD newcol)=0;**

**Remarks:**

Sets the wire frame color.

**Parameters:**

**DWORD newcol**

See [COLORREF](#).

**Prototype:**

**virtual DWORD GetWireColor() const=0;**

**Remarks:**

Returns the wire frame color. See [COLORREF](#).

**Prototype:**

**virtual void Hide(bool onOff)=0;**

**Remarks:**

Sets the hidden state.

**Parameters:**

**bool onOff**

Use true for hidden; false for not hidden.

**Prototype:**

**virtual bool IsHidden() const=0;**

**Remarks:**

Returns true if hidden; false if not hidden.

**Prototype:**

**virtual void Freeze(bool onOff)=0;**

**Remarks:**

Sets the frozen state.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool IsFrozen() const=0;**

**Remarks:**

Returns true if frozen; false if not.

**Prototype:**

**virtual void SetRenderable(bool onOff)=0;**

**Remarks:**

Sets the renderable state.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool Renderable() const=0;**

**Remarks:**

Returns true if renderable; false if not.

**Prototype:**

**virtual void SetPrimaryVisibility(bool onOff) = 0;**

**Remarks:**

This method allows you to set or unset the primary visibility flag for the layer.

**Parameters:**

**bool onOff**

TRUE to set; FALSE to unset.

**Prototype:**

**virtual bool GetPrimaryVisibility() const = 0;**

**Remarks:**

This method returns TRUE if the primary visibility flag for the layer is set.

**Prototype:**

**virtual void SetSecondaryVisibility(bool onOff) = 0;**

**Remarks:**

This method allows you to set the secondary visibility flag for the layer.

**Parameters:**

**bool onOff**

TRUE to set; FALSE to unset.

**Prototype:**

**virtual bool GetSecondaryVisibility() const = 0;**

**Remarks:**

This method returns TRUE if the secondary visibility flag for the layer is set.

**Prototype:**

**virtual void XRayMtl(bool onOff)=0;**

**Remarks:**

Sets the X-Ray material property.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool HasXRayMtl() const=0;**

**Remarks:**

Returns true if X-Ray material is set; false if not.

**Prototype:**

**virtual void IgnoreExtents(bool onOff)=0;**

**Remarks:**

Sets the ignore extents property.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool GetIgnoreExtents() const=0;**

**Remarks:**

Returns true if ignore extents is on; false if off.

**Prototype:**

**virtual void BoxMode(bool onOff)=0;**

**Remarks:**

Sets the box mode state.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool GetBoxMode() const=0;**

**Remarks:**

Returns true if box mode is on; false if off.

**Prototype:**

**virtual void AllEdges(bool onOff)=0;**

**Remarks:**

Sets the all edges setting.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool GetAllEdges() const=0;**

**Remarks:**

Returns true if all edges is on; false if off.

**Prototype:**

**virtual void VertTicks(bool onOff)=0;**

**Remarks:**

Sets the vertex ticks state.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool GetVertTicks() const=0;**

**Remarks:**

Returns true if vertex ticks is on; false if off.

**Prototype:**

**virtual void BackCull(bool onOff)=0;**

**Remarks:**

Sets the backface culling state.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool GetBackCull() const=0;**

**Remarks:**

Returns true if backface culling is on; false if not.

**Prototype:**

**virtual void SetCVertMode(bool onOff)=0;**

**Remarks:**

Sets the color per vertex display mode.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool GetCVertMode() const=0;**

**Remarks:**

Returns true if the color vertex display mode is on; otherwise false.

**Prototype:**

**virtual void SetShadeCVerts(bool onOff)=0;**

**Remarks:**

Sets the shaded color vertex display mode.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool GetShadeCVerts() const=0;**

**Remarks:**

Returns true if the shaded color vertex mode is on; false if off.

**Prototype:**

**virtual void SetCastShadows(bool onOff)=0;**

**Remarks:**

Sets the cast shadow state.

**Parameters:**

**bool onOff**

Use true for on; false for off.

**Prototype:**

**virtual bool CastShadows() const=0;**

**Remarks:**

Returns true if cast shadow is on; false if off.

**Prototype:**

**virtual void SetRcvShadows(bool onOff)=0;**

**Remarks:**

Sets the receives shadow state.

**Parameters:**

## **bool onOff**

Use true for on; false for off.

### **Prototype:**

**virtual bool RcvShadows() const=0;**

### **Remarks:**

Returns true if receives shadow is on; false if off.

### **Prototype:**

**virtual void SetMotBlur(int kind)=0;**

### **Remarks:**

Sets the type of motion blur used by the layer.

### **Parameters:**

#### **int kind**

The kind of motion blur. One of the following values:

**0:** None.

**1:** Object Motion Blur.

**2:** Image Motion Blur.

### **Prototype:**

**virtual int MotBlur() const=0;**

### **Remarks:**

Returns the type of motion blur used by the layer. One of the following values:

**0:** None.

**1:** Object Motion Blur.

**2:** Image Motion Blur.

### **Prototype:**

**virtual int GetRenderFlags() const=0;**

### **Remarks:**

This method is for internal use.

**Prototype:**

**virtual void SetRenderFlags(int flags)=0;**

**Remarks:**

This method is for internal use.

**Prototype:**

**virtual int GetDisplayFlags() const=0;**

**Remarks:**

This method is for internal use.

**Prototype:**

**virtual int AddProperty(LayerProperty &lprop)=0;**

**Remarks:**

This method is currently unused and reserved for future use.

**Prototype:**

**virtual int SetProperty(LayerProperty &lprop)=0;**

**Remarks:**

This method is currently unused and reserved for future use.

**Prototype:**

**virtual int GetProperty(LayerProperty &lprop) const=0;**

**Remarks:**

This method is currently unused and reserved for future use.

**Prototype:**

**virtual bool Used() const=0;**

**Remarks:**

Returns true if the layer is used (nodes have been added); otherwise false.

**Prototype:**

**virtual bool GetFlag(int mask) const=0;**

**Remarks:**

This method is for internal use.

**Prototype:**

**virtual bool GetFlag2(int mask) const=0;**

**Remarks:**

This method is for internal use.

**Prototype:**

**virtual void UpdateSelectionSet()=0;**

**Remarks:**

This method is for internal use in VIZ. Note: This method does nothing in 3ds max.

**Prototype:**

**virtual int GetRenderFlags(int oldlimits) const = 0;**

**Remarks:**

This method returns the render flags associated with the layer.

**Parameters:**

**int oldlimits**

The old limits flag.

**Prototype:**

**virtual void SetInheritVisibility(bool onOff) = 0;**

**Remarks:**

This method allows you to set the inherit visibility flag for the layer.

**Parameters:**

**bool onOff**

TRUE to set; FALSE to unset.

**Prototype:**

**virtual bool GetInheritVisibility() const = 0;**

**Remarks:**

This method returns TRUE if the inherit visibility flag for the layer is set.

**Prototype:**

**virtual void Trajectory(bool onOff, bool temp = false) = 0;**

**Remarks:**

This method allows you to set the display trajectory flag for the layer.

**Parameters:**

**bool onOff**

TRUE to set; FALSE to unset.

**bool temp = false**

This is used internally.

**Prototype:**

**virtual bool GetTrajectory() const = 0;**

**Remarks:**

This method returns TRUE if the display trajectory flag for the layer is set.

**Prototype:**

**virtual void SetDisplayByLayer(BOOL onOff, INode \*) = 0;**

**Remarks:**

This method allows you to set the display by layer flag on a per-node basis.

**Parameters:**

**bool onOff**

TRUE to set; FALSE to unset.

**Node \***

The pointer to the node.

**Prototype:**

**virtual void SetRenderByLayer(BOOL onOff, INode \*) = 0;**

**Remarks:**

This method allows you to set the render by layer flag on a per-node basis.

**Parameters:**

**bool onOff**

TRUE to set; FALSE to unset.

**Node \***

The pointer to the node.

**Prototype:**

**virtual void SetMotionByLayer(BOOL onOff, INode \*) = 0;**

**Remarks:**

This method allows you to set the motion by layer flag on a per-node basis.

**Parameters:**

**bool onOff**

TRUE to set; FALSE to unset.

**Node \***

The pointer to the node.

**Prototype:**

**virtual BOOL GetDisplayByLayer(INode \*) = 0;**

**Remarks:**

This method returns the state of the display by layer flag for the specified node.

**Parameters:**

**Node \***

The pointer to the node.

**Return Value:**

TRUE if set; FALSE if not set.

**Prototype:**

**virtual BOOL GetRenderByLayer(INode \*) = 0;**

**Remarks:**

This method returns the state of the render by layer flag for the specified node.

**Parameters:****Node \***

The pointer to the node.

**Return Value:**

TRUE if set; FALSE if not set.

**Prototype:**

```
virtual BOOL GetMotionByLayer(INode *) = 0;
```

**Remarks:**

This method returns the state of the motion by layer flag for the specified node.

**Parameters:****Node \***

The pointer to the node.

**Return Value:**

TRUE if set; FALSE if not set.

**Prototype:**

```
virtual void SelectObjects(void) = 0;
```

**Remarks:**

This method will select the objects of the layer.

**Prototype:**

```
virtual float GetImageBlurMultiplier(TimeValue t) = 0;
```

**Remarks:**

This method allows you to set the image blur multiplier for the layer.

**Parameters:****TimeValue t**

The timevalue to get the image blur multiplier for.

**Prototype:**

**virtual void SetImageBlurMultiplier(TimeValue t, float m) = 0;**

**Remarks:**

This method allows you to set the image blur multiplier.

**Parameters:**

**TimeValue t**

The timevalue to set the image blur multiplier for.

**float m**

The multiplier to set.

**Prototype:**

**virtual bool GetMotBlurOnOff(TimeValue t) = 0;**

**Remarks:**

This method returns the state of the motion blur flag at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the flag.

**Prototype:**

**virtual void SetMotBlurOnOff(TimeValue t, bool m) = 0;**

**Remarks:**

This method allows you to set the state of the motion blur flag at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the flag.

**bool m**

TRUE to turn on; FALSE to turn off.

## Class IDerivedObject

See Also: [Class Object](#), [Class ModContext](#), [Class Modifier](#), [Class INode](#), [Geometry Pipeline System](#).

```
class IDerivedObject : public Object
```

### Description:

This class provides an interface into derived objects. Methods of this class are provided so developers can access the modifier stack, add and delete modifiers, etc. All methods of this class are implemented by the system.

To use this interface you must #include the following file:

```
#include "modstack.h"
```

To get an **IDerivedObject** pointer from the pipeline of a node in the scene first retrieve the object reference using **INode::GetObjectRef()**. Given this **Object** pointer check its SuperClassID to see if it is **GEN\_DERIVOB\_CLASS\_ID**. If it is, you can cast it to an **IDerivedObject**.

Note: The following functions are not part of class **IDerivedObject** but are available for use in conjunction with its methods.

### Prototype:

```
IDerivedObject *CreateDerivedObject(Object *pob=NULL);
```

### Remarks:

This method creates an object space derived object.

### Parameters:

```
Object *pob=NULL
```

If non-NULL then the derived object will be set up to reference this object.

### Return Value:

A pointer to the derived object.

### Prototype:

```
Object *MakeObjectDerivedObject(Object *obj);
```

### Remarks:

Creates a new empty derived object, sets it to point at the given object and returns a pointer to the derived object.

**Parameters:**

**Object \*obj**

The object reference of the derived object will point at this object.

**Return Value:**

A pointer to the derived object.

**Prototype:**

**IDerivedObject \*CreateWSDerivedObject(Object \*pob=NULL);**

**Remarks:**

This method creates a world space derived object.

**Parameters:**

**Object \*pob=NULL**

If non-NULL then the WS derived object will be set up to reference this object.

**Return Value:**

A pointer to the derived object.

**Methods:**

**Prototype:**

**virtual void AddModifier(Modifier \*mod, ModContext \*mc=NULL, int before=0)=0;**

**Remarks:**

Adds a modifier to this derived object.

**Parameters:**

**Modifier \*mod**

The modifier to add.

**ModContext \*mc=NULL**

The mod context for the modifier.

**int before=0**

If this value is set to 0 then the modifier will be placed at the end of the

pipeline (top of stack). If this value is set to **NumModifiers()** then the modifier will be placed at the start of the pipeline (bottom of stack).

**Prototype:**

**virtual Object \*GetObjRef()=0;**

**Remarks:**

Gets the object that this derived object references. This is the next object down in the stack and may be the base object.

**Return Value:**

The object that this derived object references.

**Prototype:**

**virtual RefResult ReferenceObject(Object \*pob)=0;**

**Remarks:**

Sets the object that this derived object references. This is the next object down in the stack and may be the base object.

**Parameters:**

**Object \*pob**

The object that this derived object should reference.

**Return Value:**

One of the following values:

**REF\_SUCCEED**

**REF\_FAIL**

**Prototype:**

**virtual int NumModifiers()=0;**

**Remarks:**

Returns the number of modifiers this derived object has.

**Prototype:**

**virtual void DeleteModifier(int index=0)=0;**

**Remarks:**

Deletes the specified modifier from the stack.

**Parameters:**

**int index=0**

The index of the modifier to delete.

**Prototype:**

**virtual Modifier \*GetModifier(int index)=0;**

**Remarks:**

Returns the modifier specified by the index.

**Parameters:**

**int index**

The index of the modifier to return.

**Prototype:**

**virtual void SetModifier(int index, Modifier \*mod)=0;**

**Remarks:**

This method replaces the modifier in the stack whose index is passed.

**Parameters:**

**int index**

The index of the modifier in the stack.

**Modifier \*mod**

The modifier that will replace it.

**Prototype:**

**virtual ModContext\* GetModContext(int index)=0;**

**Remarks:**

Returns the ModContext of the specified modifier.

**Parameters:**

**int index**

The index of the modifier in the stack.

**Prototype:**

```
virtual ObjectState Eval(TimeValue t, int modIndex = 0)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to evaluate the pipeline starting with a specific modifier index. Prior to version 4.0 you had to turn all the modApps off, evaluate and then turn them on again. Now this can be easily done by specifying the modifier index.

**Parameters:****TimeValue t**

Specifies the time to evaluate the object.

**int modIndex = 0**

The index of the modifier.

**Return Value:**

The result of evaluating the object as an ObjectState.

## Class ITextObject

See Also: [Class Animatable](#).

class ITextObject

### Description:

This is the text shape object interface. This class gives access to the standard 3ds max text object. It allows the text objects font, string, and style bits to be retrieved and set. All methods of this class are implemented by the system.

To get a pointer to an **ITextObject** interface given a pointer to a object, use the following macro (defined in **ANIMTBL.H** ). Using this macro, given any Animatable, it is easy to ask for the text object interface.

```
#define GetTextObjectInterface(anim)  
((ITextObject*)anim->GetInterface(I_TEXTOBJECT))
```

A plug-in developer may use this macro as follows:

```
ITextObject *ito = GetTextObjectInterface(anim);
```

This return value will either be NULL or a pointer to a valid text object interface. You may then use this pointer to call methods of this class to retrieve and modify the object data. For example:

```
ito->SetUnderline(TRUE);
```

Note: Some aspects of the text are controlled by its parameter block. Developers can access the parameter block by calling **ito->GetParamBlock()**. The following are the indices into the parameter block used to access the size, kerning and leading parameters:

```
TEXT_SIZE  
TEXT_KERNING  
TEXT_LEADING
```

### Methods:

### Prototype:

```
virtual BOOL ChangeText(TSTR string)=0;
```

### Remarks:

This method may be called to change the text string. Note that you can't change the string if the current font is not installed.

**Parameters:**

**TSTR string**

The new text string.

**Return Value:**

TRUE if the string is changed; otherwise FALSE.

**Prototype:**

**virtual BOOL ChangeFont(TSTR name, DWORD flags)=0;**

**Remarks:**

This method may be called to change the text font.

**Parameters:**

**TSTR name**

The name of the font.

**DWORD flags**

One or more of the following values:

**TEXTOBJ\_ITALIC**

**TEXTOBJ\_UNDERLINE**

**Return Value:**

TRUE if the font was successfully changed; otherwise FALSE.

**Prototype:**

**virtual TSTR GetFont()=0;**

**Remarks:**

Returns the name of the text font.

**Prototype:**

**virtual TSTR GetString()=0;**

**Remarks:**

Returns the text string.

**Prototype:**

**virtual BOOL GetItalic()=0;**

**Remarks:**

Returns TRUE if the text is italicized; otherwise FALSE.

**Prototype:**

**virtual BOOL GetUnderline()=0;**

**Remarks:**

Returns TRUE if the text is underlined; otherwise FALSE.

**Prototype:**

**virtual void SetItalic(BOOL sw)=0;**

**Remarks:**

Sets if the text is italicized or not.

**Parameters:**

**BOOL sw**

TRUE if the text should be italicized; FALSE if not.

**Prototype:**

**virtual void SetUnderline(BOOL sw)=0;**

**Remarks:**

Sets if the text is underlined or not.

**Parameters:**

**BOOL sw**

TRUE if the text should be underlined; FALSE if not.

## Class StdMat

See Also: [Class Mtl](#), [Class Color](#), [Working with Materials](#).

```
class StdMat : public Mtl
```

### Description:

This class provides access to the properties of the 3ds max Standard material. All methods of this class are implemented by the system.

Note that some properties such as the texture maps used are accessed using methods of the base class [MtlBase](#). See that class, or the Advanced Topics section [Working with Materials](#) for more details.

### Methods Groups:

The following hyperlinks take you to the beginning of groups of related method within the class.

[Setting material properties](#)

[Retrieving material properties](#)

### Methods:

#### Prototype:

```
virtual void SetShading(int s)=0;
```

#### Remarks:

Sets the shading limit for the material.

#### Parameters:

**int s**

One of the following values:

**SHADE\_CONST**

**SHADE\_PHONG**

**SHADE\_METAL**

**SHADE\_BLINN**

#### Prototype:

```
virtual void SetSoften(BOOL onoff)=0;
```

#### Remarks:

Sets the 'Soften' setting on or off.

**Parameters:**

**BOOL onoff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual void SetFaceMap(BOOL onoff)=0;**

**Remarks:**

Sets the 'Face Map' setting on or off.

**Parameters:**

**BOOL onoff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual void SetTwoSided(BOOL onoff)=0;**

**Remarks:**

Sets the '2 Sided' setting on or off.

**Parameters:**

**BOOL onoff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual void SetWire(BOOL onoff)=0;**

**Remarks:**

Sets the 'Wire' setting on or off.

**Parameters:**

**BOOL onoff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual void SetWireUnits(BOOL onOff)=0;**

**Remarks:**

Sets the wire size to pixels or units.

**Parameters:**

**BOOL onoff**

TRUE for units; FALSE for pixels.

**Prototype:**

**virtual void SetFalloffOut(BOOL onOff)=0;**

**Remarks:**

Sets the opacity falloff setting to out or in.

**Parameters:**

**BOOL onoff**

TRUE for Out; FALSE for In.

**Prototype:**

**virtual void SetTransparencyType(int type)=0;**

**Remarks:**

Sets the additive transparency setting.

**Parameters:**

**int type**

One of the following values:

**TRANSP\_SUBTRACTIVE**

**TRANSP\_ADDITIVE**

**TRANSP\_FILTER**

**Prototype:**

**virtual void SetAmbient(Color c, TimeValue t)=0;**

**Remarks:**

Sets the ambient color to the specified value at the time passed.

**Parameters:**

**Color c**

The color to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetDiffuse(Color c, TimeValue t)=0;**

**Remarks:**

Sets the diffuse color to the specified value at the time passed.

**Parameters:****Color c**

The color to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetSpecular(Color c, TimeValue t)=0;**

**Remarks:**

Sets the specular color to the specified value at the time passed.

**Parameters:****Color c**

The color to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetFilter(Color c, TimeValue t)=0;**

**Remarks:**

Sets the filter color to the specified value at the time passed.

**Parameters:****Color c**

The color to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetShininess(float v, TimeValue t)=0;**

**Remarks:**

Sets the shininess to the specified value at the time passed.

**Parameters:**

**float v**

The value to set in the range 0 - 1.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetShinStr(float v, TimeValue t)=0;**

**Remarks:**

Sets the shininess strength to the specified value at the time passed.

**Parameters:**

**float v**

The value to set in the range 0 - 1.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetSelfIllum(float v, TimeValue t)=0;**

**Remarks:**

Sets the self illumination to the specified value at the time passed.

**Parameters:**

**float v**

The value to set in the range 0 - 1.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetOpacity(float v, TimeValue t)=0;**

**Remarks:**

Sets the opacity to the specified value at the time passed.

**Parameters:**

**float v**

The value to set in the range 0 - 1.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetOpacFalloff(float v, TimeValue t)=0;**

**Remarks:**

Sets the opacity falloff to the specified value at the time passed.

**Parameters:**

**float v**

The value to set in the range 0 - 1.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetWireSize(float s, TimeValue t)=0;**

**Remarks:**

Sets the wire size to the specified value at the time passed.

**Parameters:**

**float s**

The value to set. This value should be > 0.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetIOR(float v, TimeValue t)=0;**

**Remarks:**

Sets the index of refraction to the specified value at the time passed.

**Parameters:**

**float v**

The value to set in the range 0 - 10.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void LockAmbDiffTex(BOOL onOff)=0;**

**Remarks:**

Locks or unlocks the ambient/diffuse textures together.

**Parameters:**

**BOOL onOff**

TRUE to lock; FALSE to unlock.

**Prototype:**

**virtual void EnableMap(int i, BOOL onoff)=0;**

**Remarks:**

Enables or disables the specified map type.

**Parameters:**

**int i**

See [List of Texture Map Indices](#).

**BOOL onoff**

TRUE to enable; FALSE to disable.

**Prototype:**

**virtual BOOL MapEnabled(int i)=0;**

**Remarks:**

Returns TRUE if the specified map is enabled; otherwise FALSE.

**Parameters:**

**int i**

See [List of Texture Map Indices](#).

**Prototype:**

**virtual void SetTexmapAmt(int imap, float amt, TimeValue t)=0;**

**Remarks:**

This method is used to change the 'Amount' setting of the specified map.

**Parameters:**

**int imap**

See [List of Texture Map Indices](#).

**float amt**

The amount to set in the range of 0-1.

**TimeValue t**

The time at which to set the amount.

**Prototype:**

**virtual float GetTexmapAmt(int imap, TimeValue t)=0;**

**Remarks:**

Returns the amount setting of the specified texture map at the time passed.  
The returned range is 0 to 1.

**Parameters:**

**int imap**

See [List of Texture Map Indices](#).

**TimeValue t**

The amount at this time is returned.

**Prototype:**

**virtual void SetSamplingOn(BOOL on)=0;**

**Remarks:**

This method is available in release 3.0 and later only.  
Set super sampling on or off (enabled or disabled).

**Parameters:**

**BOOL on**

TRUE for on; FALSE for off.

**Prototype:**

**virtual int GetShading()=0;**

**Remarks:**

Returns the shading limit. One of the following values:

**SHADE\_CONST**

**SHADE\_PHONG**

**SHADE\_METAL**

**SHADE\_BLINN**

**Prototype:**

**virtual BOOL GetSoften()=0;**

**Remarks:**

Returns TRUE if soften is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetFaceMap()=0;**

**Remarks:**

Returns TRUE if face mapping is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetTwoSided()=0;**

**Remarks:**

Returns TRUE if two sided is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetWire()=0;**

**Remarks:**

Returns TRUE if wire is on; otherwise FALSE.

**Prototype:**

**virtual BOOL GetWireUnits()=0;**

**Remarks:**

Returns TRUE if the wire size is in units; FALSE if the wire size is in pixels.

**Prototype:**

**virtual BOOL GetFalloffOut()=0;**

**Remarks:**

Returns the opacity falloff setting: 1 = Out, 0 = In.

**Prototype:**

**virtual int GetTransparencyType()=0;**

**Remarks:**

Returns the transparency type.

**Return Value:**

**int type**

One of the following values:

**TRANSP\_SUBTRACTIVE**

**TRANSP\_ADDITIVE**

**TRANSP\_FILTER**

**Prototype:**

**virtual Color GetAmbient(TimeValue t)=0;**

**Remarks:**

Returns the ambient color setting at the specified time.

**Parameters:**

**TimeValue t**

The time to return the color.

**Prototype:**

**virtual Color GetDiffuse(TimeValue t)=0;**

**Remarks:**

Returns the diffuse color setting at the specified time.

**Parameters:****TimeValue t**

The time to return the color.

**Prototype:**

**virtual Color GetSpecular(TimeValue t)=0;**

**Remarks:**

Returns the specular color setting at the specified time.

**Parameters:****TimeValue t**

The time to return the color.

**Prototype:**

**virtual Color GetFilter(TimeValue t)=0;**

**Remarks:**

Returns the filter color setting at the specified time.

**Parameters:****TimeValue t**

The time to return the color.

**Prototype:**

**virtual float GetShininess( TimeValue t)=0;**

**Remarks:**

Returns the shininess setting at the specified time.

**Parameters:****TimeValue t**

The value at this time is returned.

**Prototype:**

**virtual float GetShinStr(TimeValue t)=0;**

**Remarks:**

Returns the shininess strength setting at the specified time.

**Parameters:**

**TimeValue t**

The value at this time is returned.

**Prototype:**

**virtual float GetSelfIllum(TimeValue t)=0;**

**Remarks:**

Returns the self illumination setting at the specified time.

**Parameters:**

**TimeValue t**

The value at this time is returned.

**Prototype:**

**virtual float GetOpacity(TimeValue t)=0;**

**Remarks:**

Returns the opacity setting at the specified time.

**Parameters:**

**TimeValue t**

The value at this time is returned.

**Prototype:**

**virtual float GetOpacFalloff(TimeValue t)=0;**

**Remarks:**

Returns the opacity falloff setting at the specified time.

**Parameters:**

**TimeValue t**

The value at this time is returned.

**Prototype:**

**virtual float GetWireSize(TimeValue t)=0;**

**Remarks:**

Returns the wire size setting at the specified time.

**Parameters:**

**TimeValue t**

The value at this time is returned.

**Prototype:**

**virtual float GetIOR(TimeValue t)=0;**

**Remarks:**

Returns the index of refraction setting at the specified time.

**Parameters:**

**TimeValue t**

The value at this time is returned.

**Prototype:**

**virtual BOOL GetAmbDiffTexLock()=0;**

**Remarks:**

Returns TRUE if the ambient-diffuse texture lock is set; otherwise FALSE.

## Class BitmapTex

See Also: [Class Texmap](#), [Class StdUVGen](#), [Class TextureOutput](#).

class BitmapTex: public Texmap, public FPMixinInterface

### Description:

This class is an interface into the Bitmap texture. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual void SetFilterType(int ft)=0;**

#### Remarks:

Sets the bitmap filtering method used.

#### Parameters:

**int ft**

Image filtering types. One of the following values:

**FILTER\_PYR**

Pyramidal.

**FILTER\_SAT**

Summed Area.

**FILTER\_NADA**

None.

#### Prototype:

**virtual void SetAlphaSource(int as)=0;**

#### Remarks:

This method may be used to set the alpha source for the bitmap.

#### Parameters:

**int as**

Alpha source types. One of the following values:

**ALPHA\_FILE**

Image alpha (if present).

## **ALPHA\_RGB**

RGB Intensity.

## **ALPHA\_NONE**

None (opaque).

### **Prototype:**

```
virtual void SetAlphaAsMono(BOOL onoff)=0;
```

### **Remarks:**

The Mono Channel Intensity may be either RGB Intensity or Alpha.

### **Parameters:**

**BOOL onoff**

TRUE for Alpha; FALSE for RGB Intensity.

### **Prototype:**

```
virtual void SetAlphaAsRGB(BOOL onoff)=0;
```

### **Remarks:**

This method is available in release 2.0 and later only.

The Alpha Source may be either from the RGB channels or Image Alpha channel.

### **Parameters:**

**BOOL onoff**

TRUE for RGB; FALSE for Alpha channel.

### **Prototype:**

```
virtual void SetMapName(TCHAR *name)=0;
```

### **Remarks:**

Sets the filename of the bitmap used.

### **Parameters:**

**TCHAR \*name**

The filename of the bitmap.

### **Prototype:**

**virtual StdUVGen\* GetUVGen()=0;**

**Remarks:**

Retrieves a pointer to the **StdUVGen** interface for this bitmap. This allows access to the mapping parameters such as UV offsets, blur, angle, noise level, etc.

**The following methods allow access to the values from the 'Time' rollup.**

**Prototype:**

**virtual void SetEndCondition(int endcond)=0;**

**Remarks:**

Sets the end condition setting.

**Parameters:**

**int endcond**

The end condition. One of the following values:

**END\_LOOP**

**END\_PINGPONG**

**END\_HOLD**

**Prototype:**

**virtual void SetStartTime(TimeValue t)=0;**

**Remarks:**

Sets the start time setting.

**Parameters:**

**TimeValue t**

The new start time.

**Prototype:**

**virtual void SetPlaybackRate(float r)=0;**

**Remarks:**

Sets the playback rate setting.

**Parameters:**

**float r**

The new playback rate. This is frames of the bitmap per frame of rendering time. If the value is 1 then you are playing 1 frame for every render frame. If it is 0.5 then the bitmap frame is held for 2 rendering frames.

**Prototype:**

**virtual int GetFilterType()=0;**

**Remarks:**

Returns the filter type.

**Return Value:**

One of the following values:

**FILTER\_PYR**

Pyramidal.

**FILTER\_SAT**

Summed Area Table.

**FILTER\_NADA**

None.

**Prototype:**

**virtual int GetAlphaSource()=0;**

**Remarks:**

Returns the alpha source.

**Return Value:**

One of the following values:

**ALPHA\_FILE**

**ALPHA\_RGB**

**ALPHA\_NONE**

**Prototype:**

**virtual int GetEndCondition()=0;**

**Remarks:**

Returns the end condition setting.

**Return Value:**

One of the following values:

**END\_LOOP**

**END\_PINGPONG**

**END\_HOLD**

**Prototype:**

**virtual BOOL GetAlphaAsMono(BOOL onoff)=0;**

**Remarks:**

The Mono Channel Intensity may be either RGB Intensity or Alpha.

**Parameters:**

**BOOL onoff**

This parameter is ignored.

**Return Value:**

TRUE if Alpha; FALSE if RGB Intensity.

**Prototype:**

**virtual BOOL GetAlphaAsRGB(BOOL onoff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

The Alpha Channel may be either RGB or Image Alpha.

**Parameters:**

**BOOL onoff**

This parameter is ignored.

**Return Value:**

TRUE for RGB; FALSE for Image Alpha.

**Prototype:**

**virtual TCHAR \*GetMapName()=0;**

**Remarks:**

Returns the name of the bitmap file.

**Prototype:**

**virtual TimeValue GetStartTime()=0;**

**Remarks:**

Returns the start frame setting as a TimeValue.

**Prototype:**

**virtual float GetPlaybackRate()=0;**

**Remarks:**

Returns the playback rate setting.

**Prototype:**

**virtual StdUVGen\* GetUVGen()=0;**

**Remarks:**

Retrieves a pointer to the **StdUVGen** interface for this bitmap. This allows access to the mapping parameters such as UV offsets, blur, angle, noise level, etc.

**Prototype:**

**virtual TextureOutput\* GetTexout()=0;**

**Remarks:**

Returns a pointer to a class to access TextureOutput properties of this texture.

**Prototype:**

**virtual Bitmap \*GetBitmap(TimeValue t);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the Bitmap associated with this Bitmap Texture.

**Parameters:**

**TimeValue t**

The time at which to return the bitmap.

**Prototype:**

**virtual void SetBitmap(Bitmap \*bm);**

**Remarks:**

This method is available in release 4.0 and later only.

This will swap the bitmap pointer without updating BitmapInfo.

**Parameters:**

**Bitmap \*bm**

A pointer to the bitmap.

**Default Implementation:**

**{ }**

**Prototype:**

**virtual BitmapLoadDlg();**

**Remarks:**

This method is available in release 3.0 and later only.

This method brings up a bitmap loader dialog.

**Default Implementation:**

**{ return 0; }**

**Prototype:**

**virtual ReloadBitmapAndUpdate();**

**Remarks:**

This method is available in release 3.0 and later only.

This method forces the bitmap to reload and the view to be redrawn.

**Default Implementation:**

**{ return 0; }**

**Prototype:**

**void fnReload()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method reloads the bitmap texture and operates as if the user pressed the

reload button.

**Prototype:**

**void fnViewImage()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method will view the bitmap texture image and operates as if the user pressed the view image button.

# Class IWaveSound

SeeAlso: [Class Interface](#).

class IWaveSound

## Description:

This class provides an interface into 3ds max's default WAV sound object. Use the **Interface** method **GetSoundObject()** to get a pointer to the current sound object and then use the macro **GetWaveSoundInterface()** on the result to see if it supports this interface. See the sample code below:

```
// Retrieve the current sound object
SoundObj *sound = ip->GetSoundObject();
// Attempt to get a wave interface
IWaveSound *iWave = GetWaveSoundInterface(sound);
if (iWave) {
    iWave->SetSoundFileName(_T("LedZep.wav"))
}
```

All methods of this class are implemented by the system.

## Methods:

### Prototype:

```
virtual TSTR GetSoundFileName()=0;
```

### Remarks:

Returns the name of the current sound file.

### Prototype:

```
virtual BOOL SetSoundFileName(TSTR name)=0;
```

### Remarks:

Sets the sound file. This will cause the .WAV to be loaded into the track view.

### Parameters:

**TSTR name**

The name of the sound file to set.

### Return Value:

FALSE if the file can't be opened or no wave track exists; otherwise TRUE.

**Prototype:**

**virtual void SetStartTime(TimeValue t)=0;**

**Remarks:**

Sets the time offset for the wave file.

**Parameters:**

**TimeValue t**

The start time.

**Prototype:**

**virtual TimeValue GetStartTime()=0;**

**Remarks:**

Returns the start time offset for the wave.

**Prototype:**

**virtual TimeValue GetEndTime()=0;**

**Remarks:**

Returns the end time for the wave.

## Class ISplineSelect

See Also: [Class Animatable](#), [Working with Shapes and Splines](#).

class ISplineSelect

### Description:

This class is available in release 3.0 and later only.

This class provides an interface to the Spline Select Modifier. To obtain a pointer to this class use the method **Animatable::GetInterface()** passing **I\_SPLINESELECT**.

For example:

```
ISplineSelect *iss = (ISplineSelect*)anim->GetInterface(I_SPLINESELECT);  
DWORD sl = iss->GetSelLevel();
```

### Methods:

public:

### Prototype:

```
virtual DWORD GetSelLevel()=0;
```

### Remarks:

Returns a value indicating the current selection level of the modifier. One of the following values:

```
SS_VERTEX  
SS_SEGMENT  
SS_SPLINE  
SS_OBJECT
```

### Prototype:

```
virtual void SetSelLevel(DWORD level)=0;
```

### Remarks:

Sets the current level of selection of the modifier.

### Parameters:

```
DWORD level
```

The level to set. One of the following values:

**SS\_VERTEX**

**SS\_SEGMENT**

**SS\_SPLINE**

**SS\_OBJECT**

**Prototype:**

**virtual void LocalDataChanged()=0;**

**Remarks:**

This method must be called when the selection level of the modifier is changed. Developers can use the methods of **ISplineSelect** to get and set the actual selection data. When a developers does set any of these selection sets this method must be called when done.

## Class ISplineSelectData

See Also: [Class ISplineSelect](#), [Class Animatable](#), [Class BitArray](#), [Working with Shapes and Splines](#), [Class GenericNamedSelSetList](#).

class ISplineSelectData

### Description:

This class is available in release 3.0 and later only.

When a developer gets the **LocalModData** from the **ModContext** of the Spline Select Modifier, they may cast it to this class and use these methods. They may be used to get/set the vertex/segment/spline selection state of the modifier as well as the named selection sets.

To obtain a pointer to this class use the method **Animatable::GetInterface()** passing **I\_SPLINESELECTDATA**.

For example:

```
ISplineSelectData *iss = (ISplineSelectData*)anim->GetInterface(I_SPLINESELECTDATA);  
BitArray vs = iss->GetVertSel();
```

### Methods:

public:

### Prototype:

```
virtual BitArray GetVertSel()=0;
```

### Remarks:

Returns a **BitArray** that reflects the current vertex selection. There is one bit for each vertex. Bits that are 1 indicate the vertex is selected.

### Prototype:

```
virtual BitArray GetSegmentSel()=0;
```

### Remarks:

Returns a **BitArray** that reflects the current segment selection. There is one bit for each segment. Bits that are 1 indicate the segment is selected.

### Prototype:

**virtual BitArray GetSplineSel()=0;**

**Remarks:**

Returns a **BitArray** that reflects the current spline selection. There is one bit for each spline. Bits that are 1 indicate the spline is selected.

**Prototype:**

**virtual void SetVertSel(BitArray &set, ISplineSelect \*imod, TimeValue t)=0;**

**Remarks:**

Sets the vertex selection of the modifier.

**Parameters:**

**BitArray &set**

There is one bit for each vertex. Bits that are 1 indicate the vertex should be selected.

**ISplineSelect \*imod**

Points to the ISplineSelect instance (generally this is a modifier).

**TimeValue t**

The current time at which the call is made.

**Prototype:**

**virtual void SetSegmentSel(BitArray &set, ISplineSelect \*imod, TimeValue t)=0;**

**Remarks:**

Sets the segment selection of the modifier.

**Parameters:**

**BitArray &set**

There is one bit for each vertex. Bits that are 1 indicate the segment should be selected.

**ISplineSelect \*imod**

Points to the ISplineSelect instance (generally this is a modifier).

**TimeValue t**

The current time at which the call is made.

**Prototype:**

**virtual void SetSplineSel(BitArray &set, ISplineSelect \*imod, TimeValue t)=0;**

**Remarks:**

Sets the spline selection of the modifier.

**Parameters:**

**BitArray &set**

There is one bit for each spline. Bits that are 1 indicate the spline should be selected.

**ISplineSelect \*imod**

Points to the ISplineSelect instance (generally this is a modifier).

**TimeValue t**

The current time at which the call is made.

**Prototype:**

**virtual GenericNamedSelSetList &GetNamedVertSelList()=0;**

**Remarks:**

Returns a reference to a class used for manipulating the lists of vertex level named selection sets associated with this modifier.

**Prototype:**

**virtual GenericNamedSelSetList &GetNamedSegmentSelList()=0;**

**Remarks:**

Returns a reference to a class used for manipulating the lists of segment level named selection sets associated with this modifier.

**Prototype:**

**virtual GenericNamedSelSetList &GetNamedSplineSelList()=0;**

**Remarks:**

Returns a reference to a class used for manipulating the lists of spline level named selection sets associated with this modifier.

## Class ISplineOps

See Also: [Class ISplineSelect](#), [Class Animatable](#), [Working with Shapes and Splines](#).

class ISplineOps

### Description:

This class is available in release 3.0 and later only.

This class provides an interface to the command modes and button press operations of the Editable Spline object. To obtain a pointer to this class use the method **Animatable::GetInterface()** passing **I\_SPLINEOPS**.

For example:

```
ISplineOps *iso = (ISplineOps*)anim-  
>GetInterface(I_SPLINEOPS);  
iso->StartCommandMode(ScmAttach);
```

### Methods:

public:

### Prototype:

```
virtual void StartCommandMode(splineCommandMode mode)=0;
```

### Remarks:

Begins the specified interactive command mode.

### Parameters:

**splineCommandMode mode**

The mode to begin. One of the following values:

**ScmCreateLine**

**ScmAttach**

**ScmInsert**

**ScmConnect**

**ScmRefine**

**ScmFillet**

**ScmChamfer**

**ScmBind**

**ScmRefineConnect**  
**ScmOutline**  
**ScmTrim**  
**ScmExtend**  
**ScmBreak**  
**ScmUnion**  
**ScmCrossInsert**

**Prototype:**

**virtual void ButtonOp(splineButtonOp opcode)=0;**

**Remarks:**

Performs the same operation as a button press inside the Editable Spline UI.

**Parameters:**

**splineButtonOp opcode**

The button operation to execute. One of the following values:

**SopHide**  
**SopUnhideAll**  
**SopDelete**  
**SopDetach**  
**SopDivide**  
**SopBreak**  
**SopCycle**  
**SopUnbind**  
**SopWeld**  
**SopMakeFirst**  
**SopAttachMultiple**  
**SopExplode**  
**SopReverse**  
**SopClose**  
**SopUnion**

**SopSubtract**  
**SopIntersect**  
**SopMirrorHoriz**  
**SopMirrorVert**  
**SopMirrorBoth**  
**SopSelectByID**  
**SopFuse**

**Prototype:**

**virtual void GetUIParam(splineUIParam uiCode, int &ret);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit spline parameters from the command panel. Currently not in use.

**Parameters:**

**splineUIParam uiCode**

This enum is currently empty.

**int &ret**

The returned value.

**Default Implementation:**

**{ }**

**Prototype:**

**virtual void SetUIParam(splineUIParam uiCode, int val);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit spline parameters from the command panel. Currently not in use.

**Parameters:**

**splineUIParam uiCode**

This enum is currently empty.

**int val**

The value to set.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void GetUIParam(splineUIParam uiCode, float &ret);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit spline parameters from the command panel. Currently not in use.

**Parameters:**

**splineUIParam uiCode**

This enum is currently empty.

**float &ret**

The returned value.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void SetUIParam(splineUIParam uiCode, float val);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit spline parameters from the command panel. Currently not in use.

**Parameters:**

**splineUIParam uiCode**

This enum is currently empty.

**float val**

The value to set.

**Default Implementation:**

{ }

## Class IPatchSelect

See Also: [Class Animatable](#), [Working with Patches](#).

class IPatchSelect

### Description:

This class is available in release 3.0 and later only.

This class provides an interface to the Patch Select Modifier. To obtain a pointer to this class use the method **Animatable::GetInterface()** passing **I\_PATCHSELECT**.

For example:

```
IPatchSelect *ips = (IPatchSelect*)anim-  
>GetInterface(I_PATCHSELECT);  
DWORD sl = ips->GetSelLevel();
```

### Methods:

public:

### Prototype:

```
virtual DWORD GetSelLevel()=0;
```

### Remarks:

Returns a value indicating the current selection level of the modifier. One of the following values:

```
PO_VERTEX  
PO_EDGE  
PO_PATCH  
PO_OBJECT
```

### Prototype:

```
virtual void SetSelLevel(DWORD level)=0;
```

### Remarks:

Sets the current level of selection of the modifier.

### Parameters:

```
DWORD level
```

The level to set. One of the following values:

**PO\_VERTEX**

**PO\_EDGE**

**PO\_PATCH**

**PO\_OBJECT**

**Prototype:**

**virtual void LocalDataChanged()=0;**

**Remarks:**

This method must be called when the selection level of the modifier is changed. Developers can use the methods of this class to get and set the actual selection data. When a developers does set any of these selection sets this method must be called when done.

## Class IPatchSelectData

See Also: [Class IPatchSelect](#), [Class Animatable](#), [Class BitArray](#), [Working with Patches](#), [Class GenericNamedSelSetList](#).

class IPatchSelectData

### Description:

This class is available in release 3.0 and later only.

When a developer gets the **LocalModData** from the **ModContext** of the Patch Select Modifier, they may cast it to this class and use these methods. They may be used to get/set the vertex/edge/patch sub-object selection state of the modifier as well as the named selection sets.

To obtain a pointer to this class use the method **Animatable::GetInterface()** passing **I\_PATCHSELECTDATA**.

For example:

```
IPatchSelectData *ips = (IPatchSelectData*)anim-  
>GetInterface(I_PATCHSELECTDATA);  
BitArray vs = ips->GetVertSel();
```

### Methods:

public:

### Prototype:

```
virtual BitArray GetVertSel()=0;
```

### Remarks:

Returns a **BitArray** that reflects the current vertex selection. There is one bit for each vertex. Bits that are 1 indicate the vertex is selected.

### Prototype:

```
virtual BitArray GetEdgeSel()=0;
```

### Remarks:

Returns a **BitArray** that reflects the current edge selection. There is one bit for each edge. Bits that are 1 indicate the edge is selected.

### Prototype:

**virtual BitArray GetPatchSel()=0;**

**Remarks:**

Returns a **BitArray** that reflects the current patch selection. There is one bit for each patch. Bits that are 1 indicate the patch is selected.

**Prototype:**

**virtual void SetVertSel(BitArray &set, IPatchSelect \*imod, TimeValue t)=0;**

**Remarks:**

Sets the vertex selection of the modifier.

**Parameters:**

**BitArray &set**

There is one bit for each vertex. Bits that are 1 indicate the vertex should be selected.

**IPatchSelect \*imod**

Points to the IPatchSelect instance (generally this is a modifier).

**TimeValue t**

The current time at which the call is made.

**Prototype:**

**virtual void SetEdgeSel(BitArray &set, IPatchSelect \*imod, TimeValue t)=0;**

**Remarks:**

Sets the edge selection of the modifier.

**Parameters:**

**BitArray &set**

There is one bit for each edge. Bits that are 1 indicate the edge should be selected.

**IPatchSelect \*imod**

Points to the IPatchSelect instance (generally this is a modifier).

**TimeValue t**

The current time at which the call is made.

**Prototype:**

**virtual void SetPatchSel(BitArray &set, IPatchSelect \*imod, TimeValue t)=0;**

**Remarks:**

Sets the patch selection of the modifier.

**Parameters:**

**BitArray &set**

There is one bit for each patch. Bits that are 1 indicate the patch should be selected.

**IPatchSelect \*imod**

Points to the IPatchSelect instance (generally this is a modifier).

**TimeValue t**

The current time at which the call is made.

**Prototype:**

**virtual GenericNamedSelSetList &GetNamedVertSelList()=0;**

**Remarks:**

Returns a reference to a class used for manipulating the lists of vertex level named selection sets associated with this modifier.

**Prototype:**

**virtual GenericNamedSelSetList &GetNamedEdgeSelList()=0;**

**Remarks:**

Returns a reference to a class used for manipulating the lists of edge level named selection sets associated with this modifier.

**Prototype:**

**virtual GenericNamedSelSetList &GetNamedPatchSelList()=0;**

**Remarks:**

Returns a reference to a class used for manipulating the lists of patch level named selection sets associated with this modifier.

## Class IPatchOps

See Also: [Class Animatable](#), [Working with Patches](#).

class IPatchOps

### Description:

This class is available in release 3.0 and later only.

This class provides an interface to the command modes and button press operations of the Editable Patch object. To obtain a pointer to this class use the method **Animatable::GetInterface()** passing **I\_PATCHOPS**.

For example:

```
IPatchOps *ipo = (IPatchOps*)anim-  
>GetInterface(I_PATCHOPS);  
ipo->StartCommandMode(PcmAttach);
```

### Methods:

public:

### Prototype:

```
virtual void StartCommandMode(patchCommandMode mode)=0;
```

### Remarks:

Begins the specified interactive command mode.

### Parameters:

**patchCommandMode mode**

**PcmAttach**

**PcmExtrude**

**PcmBevel**

**PcmCreate**

**PcmWeldTarget**

**PcmFlipNormal**

**PcmBind**

### Prototype:

**virtual void ButtonOp(patchButtonOp opcode)=0;**

**Remarks:**

Performs the same operation as a button press inside the Editable Patch UI.

**Parameters:**

**patchButtonOp opcode**

The button operation to execute. One of the following values:

**PopBind**

**PopUnbind**

**PopHide**

**PopUnhideAll**

**PopWeld**

**PopDelete**

**PopSubdivide**

**PopAddTri**

**PopAddQuad**

**PopDetach**

**Prototype:**

**virtual void GetUIParam(patchUIParam uiCode, int &ret);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit patch parameters from the command panel. Currently not in use.

**Parameters:**

**patchUIParam uiCode**

This enum is currently empty.

**int &ret**

The returned value.

**Default Implementation:**

**{ }**

**Prototype:**

**virtual void SetUIParam(patchUIParam uiCode, int val);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit patch parameters from the command panel. Currently not in use.

**Parameters:**

**patchUIParam uiCode**

This enum is currently empty.

**int val**

The value to set.

**Default Implementation:**

{ }

**Prototype:**

**virtual void GetUIParam(patchUIParam uiCode, float &ret);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit patch parameters from the command panel. Currently not in use.

**Parameters:**

**patchUIParam uiCode**

This enum is currently empty.

**float &ret**

The returned value.

**Default Implementation:**

{ }

**Prototype:**

**virtual void SetUIParam(patchUIParam uiCode, float val);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit patch parameters from the command panel. Currently not in use.

**Parameters:**

**patchUIParam uiCode**

This enum is currently empty.

**float val**

The value to set.

**Default Implementation:**

```
{ }
```

## Template Class IFFDMod

See Also: [Class Control](#), [Class Point3](#), [Class IPoint3](#), [Class Modifier](#).

```
template <class T> class IFFDMod : public T
```

### Description:

This class is available in release 3.0 and later only.

This class is an interface into both FFD OS modifiers and also into helper objects for FFD spacewarps.

Given a pointer to ReferenceTarget **ref**, you can use it as follows:

```
Class_ID id = ref->ClassID();  
if (id==FFDNMOSSQUARE_CLASS_ID ||  
id==FFDNMOSCYL_CLASS_ID ||  
id==FFD44_CLASS_ID || id==FFD33_CLASS_ID ||  
id==FFD22_CLASS_ID)  
IFFDMod<Modifier>* ffd = (IFFDMod<Modifier>*)ref;  
// call various methods  
}  
else if(id==FFDNMWSSQUARE_CLASS_ID || id =  
FFDNMWSCYL_CLASS_ID)  
IFFDMod<WSMObject>* ffd = (IFFDMod<WSMObject>*)ref;  
// call various methods  
}
```

### Methods:

public:

### Prototype:

```
virtual int NumPts()=0;
```

### Remarks:

Returns the number of lattice control points.

### Prototype:

```
virtual int NumPtConts()=0;
```

### Remarks:

Returns the number of Control Points having controllers.

**Prototype:**

**virtual Control\* GetPtCont(int i)=0;**

**Remarks:**

Returns a pointer to the 'i-th' Control Point controller.

**Parameters:**

**int i**

The zero based index of the Control Point.

**Prototype:**

**virtual void SetPtCont(int i,Control \*c)=0;**

**Remarks:**

Sets the controller used by the specified Control Point.

**Parameters:**

**int i**

The zero based index of the Control Point.

**Control \*c**

Points to the controller to set.

**Prototype:**

**virtual Point3 GetPt(int i)=0;**

**Remarks:**

Returns the 'i-th' Control Point.

**Parameters:**

**int i**

The zero based index of the Control Point.

**Prototype:**

**virtual void SetPt(int i, Point3 p)=0;**

**Remarks:**

Sets the 'i-th' control point.

**Parameters:**

**int i**

The zero based index of the Control Point.

**Point3 p**

The point to set.

**Prototype:**

**virtual void SetGridDim(IPoint3 d);**

**Remarks:**

Sets the lattice dimension.

**Parameters:**

**IPoint3 d**

The dimensions to set.

**Default Implementation:**

{ }

**Prototype:**

**virtual IPoint3 GetGridDim();**

**Remarks:**

Returns the lattice dimensions.

**Default Implementation:**

{ return IPoint3(0,0,0); }

**Prototype:**

**virtual void AnimateAll();**

**Remarks:**

This method assigns controllers to all the Control Points.

**Default Implementation:**

{ }

**Prototype:**

**virtual void Conform();**

**Remarks:**

Calling this method is the equivalent of pressing the Conform button in the FFD UI. Note: This method is not valid for WSMObject's.

**Default Implementation:**

**{ }**

**Prototype:**

**virtual void SelectPt(int i, BOOL sel, BOOL clearAll=FALSE);**

**Remarks:**

Selects or de-selects the specified Control Point.

**Parameters:**

**int i**

The zero based index of the Control Point.

**BOOL sel**

TRUE to select; FALSE to de-select.

**BOOL clearAll=FALSE**

TRUE to clear all the currently selected Control Points before setting the specified one; FALSE to leave the selected points alone.

**Default Implementation:**

**{ }**

## Class IParamBlock

See Also: [Class ReferenceTarget](#), [Class IParamArray](#), [Parameter Maps](#), [Class ParamBlockDescID](#), [Class Control](#).

```
class IParamBlock : public ReferenceTarget, public IParamArray
```

### Description:

This class provides methods to work with parameter blocks. For more details on parameter blocks see [Parameter Blocks](#).

### Methods:

#### Prototype:

```
virtual BOOL SetValue(int i, TimeValue t, float v)=0;
```

#### Remarks:

Implemented by the System.

Whenever the developer needs to store a value into the parameter block, the **SetValue()** method is used. There are overloaded functions for each type of value to set (int, float, Point3, and Color). Each method has three parameters. Below is the float version - the others are similar.

#### Parameters:

**int i**

This is the index into the **ParamBlockDesc** array of the parameter to set.

**TimeValue t**

The time at which to set the value.

**float v**

The value to store.

#### Return Value:

If the value was set TRUE is returned; otherwise FALSE is returned.

#### Prototype:

```
virtual BOOL SetValue(int i, TimeValue t, int v)=0;
```

#### Remarks:

This is the integer version of above.

**Prototype:**

**virtual BOOL SetValue(int i, TimeValue t, Point3& v)=0;**

**Remarks:**

This is the Point3 version of above.

**Prototype:**

**virtual BOOL SetValue(int i, TimeValue t, Color& v)=0;**

**Remarks:**

This is the Color version of above.

**Prototype:**

**virtual BOOL GetValue(int i, TimeValue t, float &v, Interval &ivalid)=0;**

**Remarks:**

Implemented by the System.

Whenever the developer needs to retrieve a value from the parameter block, the **GetValue()** method is used. There are overloaded functions for each type of value to retrieve (int, float, Point3, and Color).

**Important Note: Developers need to do range checking on values returned from a parameter block -- a spinner custom control will not necessarily ensure that the values entered by a user and stored by a parameter block are fixed to the values allowed by the spinner. For example, the spinner control ensures that it only displays, and the user is only allowed to input, values within the specified ranges. However the spinner is just a front end to a controller which actually controls the value. The user can thus circumvent the spinner constraints by editing the controller directly (via function curves in track view, key info, etc.). Therefore, when a plug-in gets a value from a controller (or a parameter block, which may use a controller) it is its responsibility to clamp the value to a valid range.**

The **GetValue()** method updates the interval passed in. This method is frequently used by developers to 'whittle' down an interval. When a parameter of a parameter block is animated, for any given time there is a interval over

which the parameter is constant. If the parameter is constantly changing the interval is instantaneous. If the parameter does not change for a certain period the interval will be longer. If the parameter never changes the interval will be **FOREVER**. By passing an interval to the `GetValue()` method you ask the parameter block to 'intersect' the interval passed in with the interval of the parameter. Intersecting two intervals means returning a new interval whose start value is the greater of the two, and whose end value is smaller of the two. In this way, the resulting interval represents a combined period of constancy for the two intervals.

This technique is used to compute a validity interval for an object. The developer starts an interval off as **FOREVER**, then intersects this interval with each of its animated parameters (by calling **GetValue()**). **GetValue()** 'whittles' down the interval with each call. When all the parameters have been intersected the result is the overall validity interval of an object at a specific time.

**Parameters:**

**int i**

Index into the **ParamBlockDesc** array of the parameter to retrieve.

**TimeValue t**

The time at which to retrieving a value.

**float v**

The value to store for the parameter at the time.

**Interval &ivalid**

The interval to update.

**Return Value:**

The return value is TRUE if a value was retrieved. Otherwise it is FALSE.

See Also: [Parameter Blocks](#) in the Advanced Topics section, [Intervals](#).

**Prototype:**

```
virtual BOOL GetValue(int i, TimeValue t, int &v, Interval  
&ivalid)=0;
```

**Remarks:**

This is the integer version of above.

**Prototype:**

**virtual BOOL GetValue(int i, TimeValue t, Point3 &v, Interval &ivalid)=0;**

**Remarks:**

This is the Point3 version of above.

**Prototype:**

**virtual BOOL GetValue(int i, TimeValue t, Color &v, Interval &ivalid)=0;**

**Remarks:**

This is the Color version of above.

**Prototype:**

**virtual SClass\_ID GetAnimParamControlType(int anim)=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the super class ID of a parameters controller.

**Parameters:**

**int anim**

Specifies the parameter whose controller super class ID is returned.

**Prototype:**

**virtual ParamType GetParameterType(int i)=0;**

**Remarks:**

Returns the type of the 'i-th' parameter.

**Parameters:**

**int i**

The zero based index of the parameter to retrieve. See [List of Parameter Types](#).

**Prototype:**

**virtual DWORD GetVersion()=0;**

**Remarks:**

Implemented by the System.  
Returns the parameter block version.

**Prototype:**

**virtual void RemoveController(int i)=0;**

**Remarks:**

Implemented by the System.  
Removes the 'i-th' controller.

**Parameters:**

**int i**

The parameter index of the controller to remove.

**Prototype:**

**virtual Control\* GetController(int i)=0;**

**Remarks:**

Implemented by the System.  
Returns the controller of the 'i-th' parameter.

**Parameters:**

**int i**

The parameter index of the controller to return.

**Prototype:**

**virtual void SetController(int i, Control \*c, BOOL  
preserveFrame0Value=TRUE)=0;**

**Remarks:**

Implemented by the System.  
Sets the 'i-th' parameter controller to **c**.

**Parameters:**

**int i**

The index of the parameter to set.

**Control \*c**

The controller to set.

**BOOL preserveFrame0Value=TRUE**

If TRUE the controllers value at frame 0 is preserved.

**Prototype:**

```
virtual void SwapControllers(int j, int k )=0;
```

**Remarks:**

Implemented by the System.

Swaps the two controllers of the parameters whose indices are passed.

**Parameters:**

**int j, int k**

The parameter indices whose controllers should be swapped.

**Prototype:**

```
virtual int GetRefNum(int paramNum)=0;
```

**Remarks:**

Implemented by the System.

Given a parameter index this method will return the reference number of that parameter.

**Parameters:**

**int paramNum**

The parameter index.

**Prototype:**

```
virtual int GetAnimNum(int paramNum)=0;
```

**Remarks:**

Implemented by the System.

Given a parameter index this method will return the anim number.

**Parameters:**

**int paramNum**

The parameter index.

**Prototype:**

```
virtual int AnimNumToParamNum(int animNum)=0;
```

**Remarks:**

Implemented by the System.

Given an anim number this method will return the parameter index.

**Parameters:**

```
int animNum
```

The anim number.

**Prototype:**

```
virtual void RescaleParam(int paramNum, float f)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This is only for use in a **ReferenceMaker::RescaleWorldUnits()**

implementation: The parameter block implementation of

**RescaleWorldUnits()** scales only tracks that have dimension type =

**stdWorldDim**. If letting the parameter block handle the rescaling is not

sufficient, call this on just the parameters you need to rescale. For additional

details on this method see the sub-section called 'Scaling Parameter Values' in

the section [Updating MAX 1.0 Plug-Ins to work with MAX 2.0](#).

**Parameters:**

```
int paramNum
```

The index into the parameter block of the parameter to rescale.

```
float f
```

The value to scale by.

**Prototype:**

```
virtual int LastNotifyParamNum()=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the index into a parameter block of the parameter that generated a notification. You can call this method when you get a **NotifyRefChanged()**

message from your parameter block to determine exactly which parameter it was that changed. When you have a very complicated dialog, for instance the Standard material, you can use this to selectively update controls in the dialog instead of updating all of them, which can feel pretty slow to the user. For smaller dialogs it's not worth the trouble.

**Sample Code:**

```
case REFMSG_CHANGE:  
if (hTarget==pblock) {  
    int np =pblock->LastNotifyParamNum();  
    // ...  
}  
break;
```

The following methods are not part of class `IParamBlock` but are available for use:

**Prototype:**

**`IParamBlock *CreateParameterBlock(ParamBlockDesc *pdesc, int count);`**

**Remarks:**

Implemented by the System.

This method is used to create a parameter block.

**Parameters:**

**`ParamBlockDesc *pdesc`**

This is an array of parameter block descriptors.

**`int count`**

This is the number in the array.

**Return Value:**

A pointer to the created parameter block. On error NULL is returned.

**Prototype:**

**`IParamBlock *CreateParameterBlock(ParamBlockDescID *pdesc, int count, DWORD version);`**

**Remarks:**

Implemented by the System.

This method is used to create a parameter block with a version number to aide in backwards compatibility.

**Parameters:**

**`ParamBlockDesc *pdesc`**

This is an array of parameter block descriptors.

**`int count`**

This is the number in the array.

**`DWORD version`**

This is used to indicate a version of the parameter block. This is used for backwards compatibility when loading 3ds max files that were saved with a previous version of the parameter block structure. There is a mechanism

which allows the older format to be converted to the newer format so the older files may still be loaded and used. See the Advanced Topics section on [Parameter Maps](#) for more information.

**Return Value:**

A pointer to the created parameter block. On error NULL is returned.

**Prototype:**

```
IParamBlock *UpdateParameterBlock(ParamBlockDescID  
*pdescOld, int oldCount, IParamBlock  
*oldPB, ParamBlockDescID *pdescNew, int newCount, DWORD  
newVersion);
```

**Remarks:**

Implemented by the System.

This creates a new parameter block, based on an existing parameter block of a later version. The new parameter block inherits any parameters from the old parameter block whose parameter IDs match.

**Parameters:**

**ParamBlockDescID \*pdescOld**

The existing parameter block descriptor.

**int oldCount**

The number of old parameters.

**IParamBlock \*oldPB**

The old parameter block.

**ParamBlockDescID \*pdescNew**

The new parameter block descriptor.

**int newCount**

The number of new parameters.

**DWORD newVersion**

The version of the new parameter block.

**Return Value:**

The new parameter block.

## Class IParamBlock2

See Also: [Class ReferenceTarget](#), [Class ParamBlockDesc2](#), [List of ParamType2 Choices](#), [Structure ParamDef](#), [Class PBValidator](#), [Class PBAccessor](#), [Class PBBitmap](#), [Class ParamBlock2PLCB](#), [Class ClassDesc2](#), [Class Animatable](#), [Structure ParamAlias](#).

```
class IParamBlock2 : public ReferenceTarget
```

### Description:

This class is available in release 3.0 and later only.

This class provides an interface for working with parameter block2s. There are methods for getting and setting parameters, descriptor access, parameter map access, etc.

### Methods Groups:

The hyperlinks below take you to the start of groups of related methods within the class:

[Version / Parameter Number / Local Name / ParamDef / BlockID Access](#)

[Descriptor Access](#)

[Index-to/from-ID Methods](#)

[Super Class ID Access](#)

[Parameter Type and Local Name](#)

[SetValue\(\) Parameter Accessors](#)

[GetValue\(\) Parameter Accessors](#)

[Shortcut Get Methods](#)

[Table \(Tab<>\) Management](#)

[Table \(Tab<>\) Insert Methods](#)

[Table \(Tab<>\) Append Methods](#)

[Keyframe Checking](#)

[Controller Access](#)

[Anim Num / Param ID Conversion](#)

[Reference Related Methods](#)

[Parameter Dimension Related Methods](#)

[Parameter Map Access](#)

[Rollout Access](#)

[MAXScript Default Parameter Initialization](#)

[Alias Maintenance](#)

[SubAnim Numbering Related Methods](#)

[Copying Parameter Values Between Blocks](#)

[Finding Parameter IDs](#)

[Reset To Default Values](#)

[PBAccessor Get / Set Methods](#)

[Validity of Parameters](#)

**Methods:**

public:

**Version / Parameter Number / Local Name / ParamDef / BlockID  
/ Owner Access**

**Prototype:**

**virtual DWORD GetVersion()=0;**

**Remarks:**

Returns the version of this parameter block.

**Prototype:**

**virtual int NumParams()=0;**

**Remarks:**

Returns the number of parameters in this parameter block.

**Prototype:**

**virtual TCHAR\* GetLocalName()=0;**

**Remarks:**

Returns the localized name for the parameter block.

**Prototype:**

**virtual ParamDef& GetParamDef(ParamID id)=0;**

**Remarks:**

Returns a reference to the ParamDef structure for this parameter block.

**Parameters:****ParamID id**

The parameter ID.

**Prototype:**

```
virtual BlockID ID()=0;
```

**Remarks:**

Returns the BlockID of the parameter block. Note: **typedef short BlockID;**

**Prototype:**

```
virtual ReferenceMaker* GetOwner()=0;
```

**Remarks:**

Returns a pointer to the owner of this parameter block.

**Descriptor Access****Prototype:**

```
virtual ParamBlockDesc2* GetDesc()=0;
```

**Remarks:**

Acquires the descriptor for this parameter block. Call **ReleaseDesc()** when done.

**Prototype:**

```
virtual void ReleaseDesc()=0;
```

**Remarks:**

Releases the descriptor for this parameter block. See **GetDesc()** above.

**Prototype:**

```
virtual void SetDesc(ParamBlockDesc2* desc)=0;
```

**Remarks:**

Sets the descriptor associated with the parameter block.

**Parameters:**

**ParamBlockDesc2\* desc**

Points to the descriptor to set.

## **Index-to/from-ID Methods**

**Prototype:**

**virtual int IDtoIndex(ParamID id)=0;**

**Remarks:**

Returns the zero based index of the parameter into the parameter definitions array of the given parameter ID or -1 if not found.

**Parameters:**

**ParamID id**

The parameter ID whose index to return.

**Prototype:**

**virtual ParamID IndextoID()=0;**

**Remarks:**

Returns the parameter ID of the parameter given its index into the parameter definitions array.

**Parameters:**

**int i**

The index of the parameter whose ID is to be returned.

## **Super Class ID Access**

**Prototype:**

**virtual SClass\_ID GetAnimParamControlType(int anim)=0;**

**Remarks:**

Returns the Super Class ID of the parameter's controller (specified by sub-anim number).

**Parameters:**

**int anim**

The sub-anim index of the parameter.

**Prototype:**

**virtual SClass\_ID GetParamControlType(ParamID id)=0;**

**Remarks:**

Returns the Super Class ID of the parameter's controller (specified by paramter ID).

**Parameters:**

**ParamID id**

The ID of the parameter.

## **Parameter Type and Local Name**

**Prototype:**

**virtual ParamType2 GetParameterType(ParamID id)=0;**

**Remarks:**

Returns the type of the specified parameter.

**Parameters:**

**ParamID id**

The ID of the parameter.

**Prototype:**

**virtual TSTR GetLocalName(ParamID id, int tabIndex = -1)=0;**

**Remarks:**

Returns the local name for the specified parameter or Tab<> parameter entry.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int tabIndex = -1**

If the parameter is a table this is the zero based index into the table of the parameter.

## **SetValue() Parameter Accessors**

### **Prototype:**

```
virtual BOOL SetValue(ParamID id, TimeValue t, float v, int  
tabIndex=0);
```

### **Remarks:**

Sets the floating point value of the specified parameter at the specified time.

### **Parameters:**

#### **ParamID id**

The permanent ID of the parameter.

#### **TimeValue t**

The time at which to set the value.

#### **float v**

The value to set.

#### **int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

### **Return Value:**

TRUE on success; otherwise FALSE.

### **Prototype:**

```
virtual BOOL SetValue(ParamID id, TimeValue t, int v, int  
tabIndex=0);
```

### **Remarks:**

Sets the integer value of the specified parameter at the specified time.

### **Parameters:**

#### **ParamID id**

The permanent ID of the parameter.

#### **TimeValue t**

The time at which to set the value.

#### **int v**

The value to set.

#### **int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL SetValue(ParamID id, TimeValue t, Point3& v, int tabIndex=0);**

**Remarks:**

Sets the Point3 value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Point3& v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL SetValue(ParamID id, TimeValue t, Color& v, int tabIndex=0);**

**Remarks:**

Sets the Color value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Color& v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL SetValue(ParamID id, TimeValue t, TCHAR* v, int  
tabIndex=0);
```

**Remarks:**

Sets the string value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**TCHAR\* v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL SetValue(ParamID id, TimeValue t, Mtl*v, int  
tabIndex=0);
```

**Remarks:**

Sets the Mtl\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Mtl\*v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL SetValue(ParamID id, TimeValue t, Texmap* v, int  
tabIndex=0);
```

**Remarks:**

Sets the Texmap\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Texmap\* v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL SetValue(ParamID id, TimeValue t, PBitmap\* v, int tabIndex=0);**

**Remarks:**

Sets the PBitmap\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**PBitmap\* v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL SetValue(ParamID id, TimeValue t, INode\* v, int tabIndex=0);**

**Remarks:**

Sets the INode\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**INode\* v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL SetValue(ParamID id, TimeValue t,  
ReferenceTarget*v, int tabIndex=0);
```

**Remarks:**

Sets the ReferenceTarget\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**ReferenceTarget\*v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL SetValue(ParamID id, TimeValue t, Matrix3& v, int  
tabIndex=0);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the Matrix3 value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Matrix3& v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**GetValue() Parameter Accessors**

**Prototype:**

**virtual BOOL GetValue(ParamID id, TimeValue t, float& v,  
Interval &ivalid, int tabIndex=0);**

**Remarks:**

Retrieves the floating point value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**float& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL GetValue(ParamID id, TimeValue t, int& v, Interval  
&ivalid, int tabIndex=0);
```

**Remarks:**

Retrieves the integer value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**int& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL GetValue(ParamID id, TimeValue t, Point3& v,  
Interval &ivalid, int tabIndex=0);
```

**Remarks:**

Retrieves the Point3 value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Point3& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL GetValue(ParamID id, TimeValue t, Color& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

Retrieves the Color value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Color& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL GetValue(ParamID id, TimeValue t, TCHAR\*& v,**

**Interval &ivalid, int tabIndex=0);**

**Remarks:**

Retrieves the string value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**TCHAR\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL GetValue(ParamID id, TimeValue t, Mtl\*& v,  
Interval &ivalid, int tabIndex=0);**

**Remarks:**

Retrieves the Mtl\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Mtl\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL GetValue(ParamID id, TimeValue t, Texmap\*& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

Retrieves the Texmap\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Texmap\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL GetValue(ParamID id, TimeValue t, PBBitmap\*& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

Retrieves the PBitmap\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**PBitmap\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**virtual BOOL GetValue(ParamID id, TimeValue t, INode\*& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

Retrieves the INode\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**INode\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL GetValue(ParamID id, TimeValue t,  
ReferenceTarget*& v, Interval &ivalid, int tabIndex=0);
```

**Remarks:**

Retrieves the ReferenceTarget\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**ReferenceTarget\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL GetValue(ParamID id, TimeValue t, Matrix3& v,  
Interval &ivalid, int tabIndex=0);
```

**Remarks:**

This method is available in release 4.0 and later only.

Retrieves the Matrix3 value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Matrix3& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

## Shortcut Get Methods

**Prototype:**

```
virtual Color GetColor(ParamID id, TimeValue t=0, int  
tabIndex=0)=0;
```

**Remarks:**

Returns the Color value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

**virtual Point3 GetPoint3(ParamID id, TimeValue t=0, int tabIndex=0)=0;**

**Remarks:**

Retrieves the value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

**virtual int GetInt(ParamID id, TimeValue t=0, int tabIndex=0)=0;**

**Remarks:**

Returns the integer value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

**virtual float GetFloat(ParamID id, TimeValue t=0, int tabIndex=0)=0;**

**Remarks:**

Returns the floating point value of the specified parameter at the specified

time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

```
virtual TimeValue GetTimeValue(ParamID id, TimeValue t=0, int  
tabIndex=0)=0;
```

**Remarks:**

Returns the TimeValue value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

```
virtual Mtl* GetMtl(ParamID id, TimeValue t=0, int  
tabIndex=0)=0;
```

**Remarks:**

Returns the Mtl\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

```
virtual TCHAR* GetStr(ParamID id, TimeValue t=0, int  
tabIndex=0)=0;
```

**Remarks:**

Returns the string value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

```
virtual Texmap* GetTexmap(ParamID id, TimeValue t=0, int  
tabIndex=0)=0;
```

**Remarks:**

Returns the Texmap\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

**virtual PBitmap\* GetBitmap(ParamID id, TimeValue t=0, int tabIndex=0)=0;**

**Remarks:**

Returns the PBitmap\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

**virtual INode\* GetINode(ParamID id, TimeValue t=0, int tabIndex=0)=0;**

**Remarks:**

Returns the INode\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

**virtual ReferenceTarget\* GetReferenceTarget(ParamID id, TimeValue t=0, int tabIndex=0)=0;**

**Remarks:**

Returns the ReferenceTarget\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

```
virtual PB2Value& GetPB2Value(ParamID id, int tabIndex=0)=0;
```

**Remarks:**

This methods is used for getting a parameter value as a PB2Value reference.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Prototype:**

```
virtual Matrix3 GetMatrix3(ParamID id, TimeValue t=0, int  
tabIndex=0)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Retrieves the value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Table (Tab<>) Management****Prototype:****virtual int Count(ParamID id)=0;****Remarks:**

Returns the number of entries being used in the table.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**Prototype:****virtual void ZeroCount(ParamID id)=0;****Remarks:**

Set the number of elements in the table that are actually used to zero.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**Prototype:****virtual void SetCount(ParamID id, int n)=0;****Remarks:**

Set the number of elements in the table that are actually used to **n**.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int n**

The number of elements to set.

**Prototype:**

**virtual int Delete(ParamID id, int start,int num)=0;**

**Remarks:**

List-type delete of **num** elements starting with **start**

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int start**

The start position for element deletion.

**int num**

The number of elements to delete.

**Return Value:**

Returns the number of items left in the table.

**Prototype:**

**virtual int Resize(ParamID id, int num)=0;**

**Remarks:**

Changes the number of allocated items to **num**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int num**

The new size of the table.

**Return Value:**

Nonzero if the array was resized; otherwise 0.

**Prototype:**

**virtual void Shrink(ParamID id)=0;**

**Remarks:**

Reallocate so there is no wasted space.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**Prototype:**

```
virtual void Sort(ParamID id, CompareFnc cmp)=0;
```

**Remarks:**

Sorts the array using the compare function.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**CompareFnc cmp**

Type of function to pass to **Sort()**. Note: **Sort()** just uses the C library **qsort** function. The developer must implement the CompareFnc function.

```
typedef int( __cdecl *CompareFnc) (const void *elem1, const void *elem2);
```

The return value of **CompareFnc** is show below in the relationship of **elem1** to **elem2**:

< 0

if **elem1** less than **elem2**

0

if **elem1** identical to **elem2**

> 0

if **elem1** greater than **elem2**

**Sample Code:**

```
static int CompTable( const void *elem1, const void *elem2 ) {  
    TCHAR *a = (TCHAR *)elem1;  
    TCHAR *b = (TCHAR *)elem2;  
    return(_tcscmp(a,b));  
}
```

## Table (Tab<>) Insert Methods

### Prototype:

**virtual int Insert(ParamID id, int at, int num, float\* el)=0;**

### Remarks:

Insert **num** float elements at position **at**.

### Parameters:

#### **ParamID id**

The permanent ID of the parameter.

#### **int at**

Zero based array index where to insert the elements.

#### **int num**

Number of elements to insert.

#### **float\* el**

Array of elements to insert.

### Return Value:

Returns **at**.

### Prototype:

**virtual int Insert(ParamID id, int at, int num, Point3\*\* el)=0;**

### Remarks:

Insert **num** Point3\* elements at position **at**.

### Parameters:

#### **ParamID id**

The permanent ID of the parameter.

#### **int at**

Zero based array index where to insert the elements.

#### **int num**

Number of elements to insert.

#### **Point3\*\* el**

Array of elements to insert.

### Return Value:

Returns **at**.

**Prototype:**

**virtual int Insert(ParamID id, int at, int num, Color\*\* el)=0;**

**Remarks:**

Insert **num** Color\* elements at position **at**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

Number of elements to insert.

**Color\*\* el**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

**virtual int Insert(ParamID id, int at, int num, TimeValue\* el)=0;**

**Remarks:**

Insert **num** TimeValue elements at position **at**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

Number of elements to insert.

**TimeValue\* el**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

```
virtual int Insert(ParamID id, int at, int num, TCHAR** vel)=0;
```

**Remarks:**

Insert **num** string (TCHAR\*) elements at position **at**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

Number of elements to insert.

**TCHAR\*\* vel**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

```
virtual int Insert(ParamID id, int at, int num, Mtl** el)=0;
```

**Remarks:**

Insert **num** Mtl\* elements at position **at**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

Number of elements to insert.

**Mtl\*\* el**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

**virtual int Insert(ParamID id, int at, int num, Texmap\*\* el)=0;**

**Remarks:**

Insert **num** Texmap\* elements at position **at**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

**Texmap\*\* el**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

**virtual int Insert(ParamID id, int at, int num, PBBitmap\*\* el)=0;**

**Remarks:**

Insert **num** PBBitmap\* elements at position **at**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

Number of elements to insert.

**PBBitmap\*\* el**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

**virtual int Insert(ParamID id, int at, int num, INode\*\* v)=0;**

**Remarks:**

Insert **num** INode\* elements at position **at**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

Number of elements to insert.

**INode\*\* v**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

**virtual int Insert(ParamID id, int at, int num, ReferenceTarget\*\* el)=0;**

**Remarks:**

Insert **num** ReferenceTarget\* elements at position **at**.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

Number of elements to insert.

**ReferenceTarget\*\* el**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

```
virtual int Insert(ParamID id, int at, int num, Matrix3** el)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Insert **num** Matrix3\* elements at position **at**.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int at**

Zero based array index where to insert the elements.

**int num**

Number of elements to insert.

**Matrix3\*\* el**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Table (Tab<>) Append Methods****Prototype:**

```
virtual int Append(ParamID id, int num, float* el, int  
allocExtra=0)=0;
```

**Remarks:**

Append **num** float elements at the end of the array.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**float\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify a non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, Point3** el, int  
allocExtra=0)=0;
```

**Remarks:**

Append **num** Point3\* elements at the end of the array.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**Point3\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify a non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, Color** el, int  
allocExtra=0)=0;
```

**Remarks:**

Append **num** Color\* elements at the end of the array.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**Color\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify an non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, TimeValue* el, int  
allocExtra=0)=0;
```

**Remarks:**

Append **num** TimeValue elements at the end of the array.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**TimeValue\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify an non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, TCHAR** el, int
```

**allocExtra=0)=0;**

**Remarks:**

Append **num** string (TCHAR\*) elements at the end of the array.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**TCHAR\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify a non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

**virtual int Append(ParamID id, int num, Mtl\*\* el, int allocExtra=0)=0;**

**Remarks:**

Append **num** Mtl\* elements at the end of the array.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**Mtl\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify a non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, Texmap** el, int  
allocExtra=0)=0;
```

**Remarks:**

Append **num** Texmap\* elements at the end of the array.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**Texmap\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify a non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, PBBitmap** el, int  
allocExtra=0)=0;
```

**Remarks:**

Append **num** PBBitmap\* elements at the end of the array.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**PBBitmap\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify an non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, INode** el, int  
allocExtra=0)=0;
```

**Remarks:**

Append **num** INode\* elements at the end of the array.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**INode\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify an non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, ReferenceTarget** el, int  
allocExtra=0)=0;
```

**Remarks:**

Append **num** ReferenceTarget\* elements at the end of the array.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**ReferenceTarget\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify an non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Prototype:**

```
virtual int Append(ParamID id, int num, Matrix3** el, int  
allocExtra=0)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Append **num** Matrix3\* elements at the end of the array.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int num**

The number of elements to append to the end of the array.

**Matrix3\*\* el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array specify an non-zero value and this many extra slots will be allocated.

**Return Value:**

Returns the number of elements in use prior to appending.

**Keyframe Checking****Prototype:**

```
virtual BOOL KeyFrameAtTime(int i, TimeValue t, int
```

**tabIndex=0);**

**Remarks:**

Checks to see if a keyframe exists for the given parameter at the given time

**Parameters:**

**int i**

Zero based index of the parameter to check.

**TimeValue t**

The time to check.

**int tabIndex=0**

If the parameter is a table this is the zero based index of the element in the table to check.

**Return Value:**

TRUE if a keyframe exists at the specified time; otherwise FALSE.

**Prototype:**

**virtual BOOL KeyFrameAtTime(ParamID id, TimeValue t, int  
tabIndex=0);**

**Remarks:**

Checks to see if a keyframe exists for the given parameter at the given time

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time to check.

**int tabIndex=0**

If the parameter is a table this is the zero based index of the element in the table to check.

**Return Value:**

TRUE if a keyframe exists at the specified time; otherwise FALSE.

**Controller Access**

**Prototype:**

**virtual void RemoveController(int i, int tabIndex)=0;**

**Remarks:**

Removes the 'i-th' controller.

**Parameters:**

**int i**

Specifies which controller using the zero based index of the parameter in the block.

**int tabIndex**

If the parameter is a table this is the zero based index of the element in the table whose controller is removed.

**Prototype:**

**virtual Control\* GetController(ParamID id, int tabIndex=0)=0;**

**Remarks:**

Returns a pointer to the controller of the specified parameter.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int tabIndex=0**

If the parameter is a table this is the zero based index of the element in the table whose controller is returned.

**Prototype:**

**virtual Control\* GetController(int i, int tabIndex)=0;**

**Remarks:**

Returns a pointer to the controller of the specified parameter.

**Parameters:**

**int i**

Specifies which controller using the zero based index of the parameter in the block.

**int tabIndex**

If the parameter is a table this is the zero based index of the element in the

table whose controller is returned.

**Prototype:**

```
virtual void SetController(int i, int tabIndex, Control *c, BOOL  
preserveFrame0Value=TRUE)=0;
```

**Remarks:**

Sets the 'i-th' parameter controller to the one specified.

**Parameters:**

**int i**

Specifies which controller using the zero based index of the parameter in the block.

**int tabIndex**

If the parameter is a table this is the zero based index of the element in the table.

**Control \*c**

The controller to set.

**BOOL preserveFrame0Value=TRUE**

If TRUE the controllers value at frame 0 is preserved.

**Prototype:**

```
virtual void SwapControllers(int i1, int tabIndex1, int i2, int  
tabIndex2)=0;
```

**Remarks:**

Swaps the two controllers of the parameters whose indices are passed.

**Parameters:**

**int i1**

The zero based index of one of the parameters in the parameter block.

**int tabIndex1**

If the parameter is a table this is the zero based index of the element in the table.

**int i2**

The zero based index of one of the other parameters in the parameter block.

## **int tabIndex2**

If the parameter is a table this is the zero based index of the element in the table.

## **Reference Related Methods**

### **Prototype:**

```
virtual int GetRefNum(int i, int tabIndex=0)=0;
```

### **Remarks:**

Given a parameter index this method will return the reference number of that parameter.

### **Parameters:**

#### **int i**

The zero based index of the parameter in the parameter block.

#### **int tabIndex=0**

If the parameter is a table this is the zero based index of the element in the table.

### **Prototype:**

```
virtual int GetControllerRefNum(int i, int tabIndex=0)=0;
```

### **Remarks:**

Returns the reference number of the specified parameter's controller or -1 if not found.

### **Parameters:**

#### **int i**

The zero based index into the parameter definitions array of the parameter.

#### **int tabIndex=0**

If the parameter is a Tab<> this is the zero based index in the table of the parameter.

### **Prototype:**

```
virtual ParamID LastNotifyParamID(int& tabIndex)=0;
```

### **Remarks:**

This method is available in release 4.0 and later only.

Like **LastNotifyParamID()**, but takes an **int& tabIndex** argument so that it can return both the ID of the changing parameter (as the result) and the changing element index for **Tab<>** parameters.

If the **ParamID** returns -1 because no parameter is currently changing, **tabIndex** is not updated. If the change to a **Tab<>** parameter is not to a single element (such as a sort), the **tabIndex** is set to -1. For multiple inserts, appends, deletes, the **tabIndex** returned is the index of the first element inserted, appended, deleted.

**Parameters:**

**int& tabIndex**

The index of the changing element for **Tab<>** parameters is returned here.

**Prototype:**

**virtual void RefDeleted(ParamID id, int tabIndex=0)=0;**

**Remarks:**

This method should be called when the parameter block owner has deleted the reference to a reference target parameter. This sets the value to NULL and invalidates the UI associated with the pblock. Note that this must only be called on **P\_OWNERS\_REF** parameters.

**Parameters:**

**ParamID id**

The ID of the reference target parameter.

**int tabIndex=0**

If the parameter is a **Tab<>** this is the zero based index into the table of the parameter.

**Prototype:**

**virtual void EnableNotifications(BOOL onOff)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Controls whether **NotifyDependents()** messages are sent when a parameter is changed, such as through

a **SetValue()** call. For example:

```
pblock->EnableNotifications(FALSE);
```

**... param change code ...**

```
pblock->EnableNotifications(TRUE);
```

Notifications are enabled by default. Note that this is a GLOBAL enable/disable, ALL paramblocks will be prevented from sending notifications while **EnableNotifications(FALSE)** is in effect.

**Parameters:**

**BOOL onOff**

TRUE to enable notifications, FALSE to disable them.

## **Anim Num / Param ID Conversion**

**Prototype:**

```
virtual int GetAnimNum(ParamID id, int tabIndex=0)=0;
```

**Remarks:**

Returns the sub-anim number of the parameter whose ID is passed or -1 if not found.

**Parameters:**

**ParamID id**

The parameter ID of the parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index in the table of the parameter.

**Prototype:**

```
virtual int AnimNumToParamNum(int animNum, int& tabIndex)=0;
```

**Remarks:**

Returns the index into the parameter definitions array of the parameter whose sub-anim index is specified or -1 if not found.

**Parameters:**

**int animNum**

The zero based sub-anim index of the parameter.

**int& tabIndex**

If the parameter is a Tab<> this is the zero based index into the table of the parameter.

**Parameter Dimension Related Methods****Prototype:**

```
virtual ParamDimension* GetParamDimension(int subAnim)=0;
```

**Remarks:**

Returns the dimension of the parameter whose sub-anim index is passed or **defaultDim** if not found.

**Parameters:****int subAnim**

The zero based sub-anim index of the parameter.

**Prototype:**

```
virtual void RescaleParam(int paramNum, int tabIndex, float f)=0;
```

**Remarks:**

This is only for use in a RescaleWorldUnits() implementation: The parameter block implementation of RescaleWorldUnits scales only tracks that have dimension type = stdWorldDim. If letting the parameter block handle the rescaling is not sufficient, call this on just the parameters you need to rescale.

**Parameters:****int paramNum**

The index into the parameter block of the parameter to rescale.

**int tabIndex**

If the parameter is a Tab<> this is the zero based index into the table of the parameter.

**float f**

The value to scale by.

## Parameter Map Access

### Prototype:

**virtual void SetMap(IParamMap2\* m, MapID map\_id = 0)=0;**

### Remarks:

This method sets the parameter map2 associated with this parameter block2.

### Parameters:

**IParamMap2\* m**

Points to the parameter map.

**MapID map\_id**

This parameter is available in release 4.0 and later only.

Specifies the ID of the map to set.

### Prototype:

**virtual IParamMap2\* GetMap(MapID map\_id = 0)=0;**

### Remarks:

Returns a pointer to the parameter map2 associated with this parameter block.

### Parameters:

**MapID map\_id**

This parameter is available in release 4.0 and later only.

Specifies the ID of the map to get.

## Rollout Access

### Prototype:

**virtual void SetRolloutOpen(BOOL open, MapID map\_id = 0)=0;**

### Remarks:

Sets the rollout state to open or closed.

Note: Normally, developers don't need to call this method (or the related ones below) explicitly; they are used internally to keep track of rollouts states.

Instead, use the **ClassDesc2** method **RestoreRolloutState()** at the end of a **BeginEditParams()** or **CreateParamDlg()** to reset the rollouts to the state last used for the current object.

**Parameters:**

**BOOL open**

TRUE for open; FALSE for closed.

**MapID map\_id**

This parameter is available in release 4.0 and later only.

Specifies the ID of the map/rollout to set open/closed state for.

**Prototype:**

**virtual BOOL GetRolloutOpen(MapID map\_id = 0)=0;**

**Remarks:**

Returns TRUE if the rollout is open; FALSE if closed. This is normally used internally -- see the note above in **SetRolloutOpen()**.

**Parameters:**

**MapID map\_id**

This parameter is available in release 4.0 and later only.

Specifies the ID of the map/rollout to get open/closed state for.

**Prototype:**

**virtual void SetRolloutScrollPos(int pos, MapID map\_id = 0)=0;**

**Remarks:**

Sets the rollout scroll position. This is normally used internally -- see the note above in **SetRolloutOpen()**.

**Parameters:**

**int pos**

The position to set.

**MapID map\_id**

This parameter is available in release 4.0 and later only.

Specifies the ID of the map/rollout to set scroll position for.

**Prototype:**

**virtual int GetRolloutScrollPos(MapID map\_id = 0)=0;**

**Remarks:**

Returns the rollout scroll position. This is normally used internally -- see the note above in **SetRolloutOpen()**.

**Parameters:**

**MapID map\_id**

This parameter is available in release 4.0 and later only.  
Specifies the ID of the map/rollout to get scroll position for.

## ParamDlg Access

**Prototype:**

**virtual IAutoMParamDlg\* GetMParamDlg()=0;**

**Remarks:**

Returns a pointer to the automatic parameter dialog object for a plug-in material or texmap (which has its interface in the materias editor). See [Class IAutoMParamDlg](#).

**Prototype:**

**virtual IAutoEParamDlg\* GetEParamDlg()=0;**

**Remarks:**

Returns a pointer to the automatic parameter dialog object for the rendering effects plug-in. See [Class IAutoEParamDlg](#).

## MAXScript Default Parameter Initialization

**Prototype:**

**virtual void InitMSParameters()=0;**

**Remarks:**

This method initializes the parameters with MAXScript defaults.

The **ParamBlockDesc2** descriptor lets you specify default values for parameters (using the tag **p\_default**), and these get installed when you first create an object and its paramblocks. Sometimes, the default value needed for interactive creation is not the one you want when creating an object via the scripter. For example, a sphere should start out with radius 0 when you create it interactively, but you want a non-zero default if you create it in the scripter,

say with sphere(), otherwise it would be invisible. There is another tag, p\_ms\_default, that lets you set a separate default for scripter-based creation (the **p\_ms\_default** for sphere radius is 25). This method is used internally by the scripter to set the **p\_ms\_default** values in after a script-based creation. It is not normally used by plug-in developers.

## Alias Maintenance

### Prototype:

```
virtual void DefineParamAlias(TCHAR* alias_name, ParamID id, int tabIndex=-1)=0;
```

### Remarks:

This is used to allow parameter 'aliases' to be set up for MAXScript use. Individual Tab<> parameter elements can have aliases. This is used to set up dynamically-varying parameters, such as the texture maps in the new Standard material, which sets up aliases for elements in the texture map arrays.

### Parameters:

**TCHAR\* alias\_name**

The name of the alias.

**ParamID id**

The permanent ID of the parameter.

**int tabIndex=-1**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

### Prototype:

```
virtual ParamAlias* FindParamAlias(TCHAR* alias_name)=0;
```

### Remarks:

Returns a pointer to a [Structure ParamAlias](#) object which describes the parameter alias whose name is passed. This includes the name, ParamID and Tab<> index.

### Parameters:

**TCHAR\* alias\_name**

The name of the alias to find.

**Prototype:**

```
virtual TCHAR* FindParamAlias(ParamID id, int  
tabIndex=-1)=0;
```

**Remarks:**

Finds the name of a parameter alias using the ID and Tab<> index passed.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int tabIndex=-1**

If the parameter is a Tab<> this is the zero based index into the table. If not a Tab<> use the default of -1.

**Return Value:**

The name of the alias or NULL if not found.

**Prototype:**

```
virtual void ClearParamAliases()=0;
```

**Remarks:**

Breaks the association between the aliases and their parameters in this block.  
The method **ParamAliasCount()** below will return 0 following this call.

**Prototype:**

```
virtual int ParamAliasCount()=0;
```

**Remarks:**

Returns the number of aliases currently defined.

**Prototype:**

```
virtual ParamAlias* GetParamAlias(int i)=0;
```

**Remarks:**

Returns a pointer to the 'i-th' alias. See [Structure ParamAlias](#).

**Parameters:****int i**

The zero based index of the alias to return.

## SubAnim Numbering Related Methods

### Prototype:

```
virtual void SetSubAnimNum(ParamID id, int subAnimNum, int  
tabIndex=0)=0;
```

### Remarks:

This method allows for the arbitrary ordering of sub-anim numbers for parameters and Tab<> parameter elements. It sets the sub-anim number for the specified parameter.

This call lets you set arbitrary sub-anim number ordering for the subAnim parameters and Tab<> parameter elements in the block. You must set numbers for ALL subAnims and take care that all numbers are used. Note that in the case of the various ReferenceTarget\* parameter types, NULL values for any parameter or Tab<> elements are **not** included the subAnim count, so if such a parameter is made non-NULL (or vice-versa), you need to reassign the subanim numbers to take that change into account.

### Parameters:

#### **ParamID id**

The permanent parameter ID

#### **int subAnimNum**

The zero based sub-anim number.

#### **int tabIndex=0**

The zero based index into the table of the parameter.

### Prototype:

```
virtual void ClearSubAnimMap()=0;
```

### Remarks:

This method clears any sub-anim map used to allow arbitrary ordering of sub-anim numbers for parameters.

## Copying Parameter Values Between Blocks

### Prototype:

```
virtual void Assign(ParamID id, IParamBlock2* src, ParamID
```

**src\_id)=0;**

**Remarks:**

This method is used for copying parameter values between parameter blocks (which is useful during old-version updating). This method copies from the 'src' block 'src\_id' parameter into this parameter block's 'id' parameter. Developers are responsible for making sure the types are the same, otherwise an assert() may occur.

**Parameters:**

**ParamID id**

This ID specifies the destination parameter.

**IParamBlock2\* src**

Points to the source parameter block 2.

**ParamID src\_id**

The source parameter ID.

## **Finding Parameter IDs**

**Prototype:**

**virtual ParamID FindRefParam(ReferenceTarget\* ref, int& tabIndex)=0;**

**Remarks:**

This method that takes a reference target object **ref** stored somewhere in this parameter block and returns the ParamID and tabIndex of the containing parameter, or -1 if not found in the parameter block.

**Parameters:**

**ReferenceTarget\* ref**

The reference target to find.

**int& tabIndex**

The table index if the parameter is a Tab<>.

## **Reset To Default Values**

**Prototype:**

**virtual void ResetAll(BOOL updateUI = TRUE, BOOL**

**callSetHandlers = TRUE)=0;**

**Remarks:**

This method resets all the parameters in the block to their default values and optionally updates any associated ParamMap2 UI that is currently displaying the contents of the block. It also optionally causing all the PBAccessor Set() methods to be called after the reset.

**Parameters:**

**BOOL updateUI = TRUE**

TRUE to update the user interface; FALSE to not update.

**BOOL callSetHandlers = TRUE**

TRUE to call **PBAccessor::Set()** for all the parameters; otherwise FALSE.

**Prototype:**

**virtual void Reset(ParamID id, int tabIndex=-1, BOOL updateUI = TRUE, BOOL callSetHandlers = TRUE)=0;**

**Remarks:**

This method resets the single parameter specified to its default value. If the parameter is a Tab<> and the tabIndex is -1, all the elements in the table are reset.

**Parameters:**

**ParamID id**

The ID of the parameter to reset.

**int tabIndex=-1**

If the parameter is a Tab<> this is the index into the table of the parameter to reset. A value of -1 causes all the elements in the table to be reset.

**BOOL updateUI = TRUE**

Determines if the user interface is updated for the parameter. TRUE to update; FALSE to not update.

**BOOL callSetHandlers = TRUE**

Determines if the method **PBAccessor::Set()** should be called on the parameter. TRUE to call it; FALSE to not call it.

**PBAccessor Get / Set Methods**

**Prototype:**

**virtual void CallSet(ParamID id, int tabIndex=-1)=0;**

**Remarks:**

This method forces a call to the **PBAccessor::Set()** method for the specified parameter. If the parameter is a Tab<> parameter and the tabIndex is -1, all the elements have the appropriate functions called.

**Parameters:****ParamID id**

The ID of the parameter.

**int tabIndex=-1**

If the parameter is a Tab<> parameter this is the zero based index into the table of the element. A value of -1 causes all the appropriate **Set()** methods to be called.

**Prototype:**

**virtual void CallGet(ParamID id, int tabIndex=-1)=0;**

**Remarks:**

This method forces a call to the **PBAccessor::Get()** method for the specified parameter. If the parameter is a Tab<> parameter and the tabIndex is -1, all the elements have the appropriate functions called.

**Parameters:****ParamID id**

The ID of the parameter.

**int tabIndex=-1**

If the parameter is a Tab<> parameter this is the zero based index into the table of the element. A value of -1 causes all the appropriate **Get()** methods to be called.

**Prototype:**

**virtual void CallSets()=0;**

**Remarks:**

This method forces a call to the **PBAccessor::Set()** method for every parameter in the block. Any parameters which are Tab<> parameters will have

**Set()** call for every appropriate element.

**Prototype:**

**virtual void CallGets()=0;**

**Remarks:**

This method forces a call to the **PBAccessor::Get()** method for every parameter in the block. Any parameters which are Tab<> parameters will have **Get()** call for every appropriate element.

## **Validity of Parameters**

**Prototype:**

**virtual void GetValidity(TimeValue t, Interval &valid)=0;**

**Remarks:**

This method updates the validity interval passed with the cumulative interval for every parameter in the parameter block.

**Parameters:**

**TimeValue t**

The time about which the interval is computed.

**Interval &valid**

The interval to update.

## Class IParamMap

See Also:, [Parameter Maps](#), [Class ParamMapUserDlgProc](#), [Class ParamUIDesc](#), [Parameter Blocks](#), [Class IParamArray](#), [Class Interface](#).

### Description:

This class provides methods to work with parameter maps. These are things like invalidating the parameter map so it gets redrawn, working with the parameter blocks associated with the parameter map, and establishing an optional dialog proc to handle controls not directly handled by the pmap. This section also documents several functions that are available for creating and destroying parameter maps but are not part of this class.

Note: The use of this class requires the explicit inclusion the **IPARAMM.H** header file.

The following functions are not part of this class but are available to create the parameter maps:

**Prototype:**

```
IParamMap *CreateCPPParamMap(ParamUIDesc *desc,int  
count,IParamArray *pb,Interface *ip,HINSTANCE  
hInst,TCHAR *dlgTemplate,TCHAR *title,DWORD flags);
```

**Remarks:**

Creates a parameter map to handle the display of parameters in the command panel. This will add the rollup page to the command panel.

**Parameters:**

**ParamUIDesc \*desc**

The array of ParamUIDescs, one element for each control to be managed.

**int count**

The number of items in the array above.

**IParamArray \*pb**

Pointer to an instance of the class IParamArray. This is the pointer to the [= 4\) BSPSPopupOnMouseOver\(event\);">virtual array](#) of parameters.

**Interface \*ip**

The interface pointer passed into the BeginEditParams() method.

**HINSTANCE hInst**

The DLL instance handle of the plug-in

**TCHAR \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A set of flags to control settings of the rollup page.

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**Return Value:**

A pointer to the parameter map that is created.

**Prototype:**

**IParamMap \*ReplaceCPPParamMap(HWND oldhw, ParamUIDesc \*desc,int count, IParamArray \*pb, Interface \*ip, HINSTANCE hInst, TCHAR \*dlgTemplate, TCHAR \*title, DWORD flags);**

**Remarks:**

This method is available in release 3.0 and later only.

This method allows one to switch rollups in the command panel. It creates the new parameter map and calls **Interface:: ReplaceRollupPage()**.

**Parameters:****HWND oldhw**

The window handle of the old rollup.

**ParamUIDesc \*desc**

The array of ParamUIDescs, one element for each control to be managed.

**int count**

The number of items in the array above.

**IParamArray \*pb**

Pointer to an instance of the class IParamArray. This is the pointer to the virtual array of parameters.

**Interface \*ip**

The interface pointer passed into the BeginEditParams() method.

**HINSTANCE hInst**

The DLL instance handle of the plug-in

**TCHAR \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A set of flags to control settings of the rollup page.

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**Return Value:**

A pointer to the parameter map that is created.

**Prototype:**

```
void DestroyCPPParamMap(IParamMap *m);
```

**Remarks:**

This function destroys a command panel parameter map. The rollup page from the command panel is removed and the parameter map is deleted.

**Parameters:**

**IParamMap \*m**

A pointer to the parameter map to destroy.

**Prototype:**

```
BOOL CreateModalParamMap(ParamUIDesc *desc,int  
count,IParamArray *pb,TimeValue t,HINSTANCE hInst,TCHAR  
*dlgTemplate,HWND hParent,ParamMapUserDlgProc  
*proc=NULL);
```

**Remarks:**

This function creates a parameter map that will handle a parameter block in a modal dialog where time does not change and the viewport is not redrawn. Note that there is no need to destroy it. It executes the dialog and then destroys itself.

**Parameters:**

**ParamUIDesc \*desc**

The array of ParamUIDescs, one element for each control to be managed.

**int count**

The number of items in the array above.

**IParamArray \*pb**

Pointer to an instance of the class IParamArray. This is the pointer to the [= 4\) BSPSPopupOnMouseOver\(event\);">virtual array](#) of parameters.

**TimeValue t**

This is just the current time when the user is bringing up the dialog.

**HINSTANCE hInst**

The DLL instance handle of the plug-in

**TCHAR \*dlgTemplate**

Dialog template for the dialog box.

## **HWND hParent**

The parent window handle.

## **ParamMapUserDlgProc \*proc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from **ParamMapUserDlgProc** and set it as the parameter map's user callback. See [Class ParamMapUserDlgProc](#).

## **Return Value:**

TRUE if the user selected OK; otherwise FALSE.

## **Prototype:**

```
IParamMap *CreateRParamMap(ParamUIDesc *desc,int  
count,IParamArray *pb, IRenderParams *ip, HINSTANCE  
hInst,TCHAR *dlgTemplate, TCHAR *title, DWORD flags);
```

## **Remarks:**

This function creates a parameter map to handle the display of render parameters or atmospheric plug-in parameters.

## **Parameters:**

### **ParamUIDesc \*desc**

The array of ParamUIDescs, one element for each control to be managed.

### **int count**

The number of items in the array above.

### **IParamArray \*pb**

Pointer to an instance of the class IParamArray. This is the pointer to the array of parameters.

### **IRenderParams \*ip**

The interface pointer passed into **CreateParamDlg()**.

### **HINSTANCE hInst**

The DLL instance handle of the plug-in

### **TCHAR \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

### **TCHAR \*title**

The title displayed in the rollup page title bar.

### **DWORD flags**

A flag to control the settings of the rollup page:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**Return Value:**

A pointer to the parameter map that is created.

**Prototype:**

**void DestroyRParamMap(IParamMap \*m);**

**Remarks:**

This function destroys a parameter map created by **CreateRParamMap()**. The rollup page is removed and the parameter map is deleted.

**Parameters:**

**IParamMap \*m**

A pointer to the parameter map to destroy.

**Prototype:**

**IParamMap \*CreateMParamMap(ParamUIDesc \*desc, int count, IParamArray \*pb, IMtlParams \*ip, HINSTANCE hInst, TCHAR \*dlgTemplate, TCHAR \*title, DWORD flags);**

**Remarks:**

This function is available in release 2.0 and later only.

This function creates a parameter map to handle the display of texture map or material parameters in the material editor.

**Parameters:**

**ParamUIDesc \*desc**

The array of ParamUIDescs, one element for each control to be managed.

**int count**

The number of items in the array above.

**IParamArray \*pb**

Pointer to an instance of the class IParamArray. This is the pointer to the array of parameters.

**IMtlParams \*ip**

The interface pointer. See [Class IMtlParams](#).

**HINSTANCE hInst**

The DLL instance handle of the plug-in

**TCHAR \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A flag to control the settings of the rollup page:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**Return Value:**

A pointer to the parameter map that is created.

**Prototype:**

**void DestroyMParamMap(IParamMap \*m);**

**Remarks:**

This function is available in release 2.0 and later only.

This function destroys a parameter map created by **CreateMParamMap()**.

The rollup page is removed and the parameter map is deleted.

**Parameters:**

**IParamMap \*m**

A pointer to the parameter map to destroy.

**Methods:**

**Prototype:**

**void Invalidate()**

**Remarks:**

Implemented by the System.

Call this method to update (redraw) the user interface controls. This marks the UI controls as needing to be updated and the parameter map will take care of it.

**Prototype:**

```
void SetParamBlock(IParamArray *pb)
```

**Remarks:**

Implemented by the System.

This method swaps the existing parameter block with a new one and updates the user interface. Consider the following example to understand how this is used: If a user is in create mode, and has created a sphere object, and then goes to create another sphere, the user interface stays up. The parameter map that manages the UI is not deleted. When the user creates the second sphere, the parameter map needs to refer to the new sphere's parameter block (not the previous one any longer). This method is used to set the parameter map to point to the new parameter block.

**Parameters:**

```
IParamArray *pb
```

A pointer to the new parameter block.

**Prototype:**

```
void SetUserDlgProc(ParamMapUserDlgProc *proc=NULL)
```

**Remarks:**

Implemented by the System.

This method allows the developer to provide special handling for a control. The developer provides a dialog proc to process the message from the control. This method is used to tell the parameter map that the developer defined method should be called. The given proc will be called **after** default processing is done. Note that if the proc is non-NULL when the ParamMap is deleted its **DeleteThis()** method will be called.

**Parameters:**

```
ParamMapUserDlgProc *proc=NULL
```

A pointer to the user dialog proc class to process the control.

**Prototype:**

```
virtual void SetPBlockIndex(int mapIndex, int blockIndex)=0;
```

**Remarks:**

Implemented by the System.

This method changes a parameter map entry to refer to a different item in the parameter block. This is used for example by the Optimize modifier. This modifier has two sets of parameters that may be adjusted (L1 and L2).

Optimize only maintains a single parameter block however. This pblock contains both sets of parameters. When the user switches between these two sets, this method is called to point the UI controls at different indices in the parameter block.

**Parameters:**

**int mapIndex**

The map entry to change.

**int blockIndex**

The new parameter block index.

**Prototype:**

**HWND GetHWnd()**

**Remarks:**

Implemented by the System.

Returns the window handle of the rollup page (or dialog).

**Prototype:**

**IParamArray \*GetParamBlock()**

**Remarks:**

Implemented by the System.

Returns a pointer to the parameter block managed by the parameter map.

## Class IParamMap2

See Also: [Class IParamBlock2](#), [Class ParamBlockDesc2](#), [Class ParamMap2UserDlgProc](#).

```
class IParamMap2 : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This class provides methods to work with parameter map2s. Methods are provided for things like invalidating the parameter map so it gets redrawn, working with the parameter blocks associated with the parameter map, and establishing an optional dialog proc to handle controls not directly handled by the parameter map. This section also documents several functions that are available for creating and destroying parameter maps but are not part of this class.

Note: The use of this class requires the explicit inclusion the IPARAMM2.H header file.

### Methods:

#### Prototype:

```
virtual void Invalidate()=0;
```

#### Remarks:

This method marks the user interface as needing to be updated. This affects the entire UI for the parameter map.

#### Prototype:

```
virtual void Validate()=0;
```

#### Remarks:

This methods un-invalidates the entire user interface.

#### Prototype:

```
virtual void Invalidate(ParamID id, int tabIndex=0)=0;
```

#### Remarks:

This methods marks a specific control in the UI as requiring an update.

**Parameters:****ParamID id**

The permanent ID of the control requiring an update.

**int tabIndex=0**

If the control is a Tab<> then this is the zero based index into the table of the value to be invalidated.

**Prototype:**

**virtual Interval& Validity()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Provides access to pmap's validity interval. Returns a reference to the actual interval so it can be modified if desired.

**Prototype:**

**virtual void UpdateUI(TimeValue t)=0;**

**Remarks:**

This method will update the user interface if the current settings are not valid at the given time (i.e., if anything is animated at that time).

**Parameters:****TimeValue t**

The time to check for the update.

**Prototype:**

**virtual void RedrawViews(TimeValue t, DWORD  
flag=REDRAW\_NORMAL)=0;**

**Remarks:**

This method may be called to cause the viewports to be redrawn.

**Parameters:****TimeValue t**

The time at which to redraw the viewports.

**DWORD flag=REDRAW\_NORMAL**

You may specify one of the following:

**REDRAW\_BEGIN**

Call this before you redraw.

**REDRAW\_INTERACTIVE**

This allows the view quality to degrade to maintain interactivity.

**REDRAW\_END**

If during interactive redraw the state degraded, this will redraw the views in the undegraded state.

**REDRAW\_NORMAL**

This redraws the views in the undegraded state.

**Prototype:**

**virtual void SetParamBlock(IParamBlock2 \*pb)=0;**

**Remarks:**

This method swaps the existing parameter block with a new one and updates user interface.

**Parameters:**

**IParamBlock2 \*pb**

Points to the new parameter block2 to use.

**Prototype:**

**virtual void SetUserDlgProc(ParamMap2UserDlgProc \*proc=NULL)=0;**

**Remarks:**

This method allows a developer to provide special handling for one or more controls. The developer provides a dialog proc to process the message from the control. This method is used to tell the parameter map that the developer defined method should be called. The given proc will be called after default processing is done. Note that if the proc is non-NULL when the ParamMap2 is deleted its DeleteThis() method will be called.

**Parameters:**

**ParamMap2UserDlgProc \*proc=NULL**

Points to the user dialog proc object to process the controls.

**Prototype:**

**virtual ParamMap2UserDlgProc \*GetUserDlgProc()=0;**

**Remarks:**

Returns a pointer to the user dialog proc for the parameter map (or NULL if none is defined). See **SetUserDlgProc()** above.

**Prototype:**

**virtual void ReplaceParam(ParamID curParam, ParamID newParam);**

**Remarks:**

This method changes a parameter map entry to refer to a different item in the parameter block.

**Parameters:**

**ParamID curParam**

The parameter ID of the item to change.

**ParamID newParam**

The new parameter ID.

**Prototype:**

**virtual HWND GetHwnd()=0;**

**Remarks:**

Returns the dialog window handle of the parameter map.

**Function:**

**virtual MapID GetMapID()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns the parameter map's ID.

**Prototype:**

**virtual IParamBlock2 \*GetParamBlock()=0;**

**Remarks:**

Returns a pointer to the parameter block2 used by this parameter map.

**Prototype:**

**virtual BOOL DlgActive()=0;**

**Remarks:**

Returns TRUE if the parameter map dialog proc is active; otherwise FALSE.

**Prototype:**

**virtual ParamBlockDesc2\* GetDesc()=0;**

**Remarks:**

Returns a pointer to the ParamBlockDesc2 instance for this parameter map.

**Prototype:**

**virtual void ActivateDlg(BOOL onOff)=0;**

**Remarks:**

This method is called to indicate the dialog is going inactive or is becoming active.

**Parameters:**

**BOOL onOff**

TRUE if becoming active; FALSE for inactive.

**Prototype:**

**virtual int FindSubTexFromHWND(HWND hw)=0;**

**Remarks:**

This method is sent to a Material Editor map to find the SubTex index corresponding to the control handle. It should return the index of the sub-  
texmap corresponding to the window whose handle is passed. If the handle is not valid return -1.

**Parameters:**

**HWND hw**

The window handle of the control.

**Prototype:**

```
virtual void Enable(ParamID id, BOOL onOff, int tabIndex=0)=0;
```

**Remarks:**

This method is used to enable or disable an individual user interface control.

**Parameters:****ParamID id**

The parameter ID of the control to enable/disable.

**BOOL onOff**

TRUE to enable; FALSE to disable.

**int tabIndex=0**

If the control is a Tab<> then this is the zero based index in the table of the item to enable/disable.

**Prototype:**

```
virtual void SetText(ParamID id, TCHAR* txt, int tabIndex=0)=0;
```

**Remarks:**

This method sets the text of a parameter user interface control.

**Parameters:****ParamID id**

The ID of the parameter whose user interface text to change.

**TCHAR\* txt**

The new string to display.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

**Prototype:**

```
virtual void SetTooltip(ParamID id, BOOL onOf, TCHAR* txt, int  
tabIndex=0)=0;
```

**Remarks:**

This method sets the tooltip text of specified parameter and can turn the tooltip on or off.

**Parameters:****ParamID id**

The ID of the parameter.

**BOOL onOf**

TRUE for on; FALSE for off.

**TCHAR\* txt**

The tool tip text.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

**Prototype:**

```
virtual void SetRange(ParamID id, float low, float high, int  
tabIndex=0)=0;
```

**Remarks:**

This method sets the range of parameter for a spinner or slider control.

**Parameters:****ParamID id**

The ID of the parameter.

**float low**

The low range for the spinner / slider.

**float high**

The high range for the spinner / slider.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

**Prototype:**

```
virtual void Show(ParamID id, BOOL showHide, int  
tabIndex=0)=0;
```

**Remarks:**

This method will show or hide the specified control.

**Parameters:****ParamID id**

The ID of the parameter.

**BOOL showHide**

TRUE to show; FALSE to hide.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

**Prototype:**

```
virtual void SetThing(ReferenceTarget *m)=0;
```

**Remarks:**

This method is called by any **IAutoXXParamDlg** when it receives a **SetThing()**.

**Parameters:****ReferenceTarget \*m**

The item which was set.

**The following functions are not part of this class but are available for use.**

**Function:**

```
BOOL CreateModalParamMap2(IParamBlock2 *pb, TimeValue t,  
HINSTANCE hInst, TCHAR *dlgTemplate, HWND hParent,  
ParamMap2UserDlgProc *proc=NULL);
```

**Remarks:**

This function creates a parameter map that will handle a parameter block in a modeless dialog where time does not change and the viewports are not redrawn. Note that there is no need to destroy it. It executes the dialog and then destroys itself.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **CreateModalParamMap2()** with default map ID of 0.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**TimeValue t**

The time at which the dialog is launched.

**HINSTANCE hInst**

The plug-ins instance handle.

**TCHAR \*dlgTemplate**

The dialog template.

**HWND hParent**

The parent window.

**ParamMap2UserDlgProc \*proc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**Return Value:**

Returns TRUE if the user selected OK, FALSE otherwise.

**Function:**

```
BOOL CreateModalParamMap2(IParamBlock2 *pb, TimeValue t,  
HINSTANCE hInst, DLGTEMPLATE *dlgTemplate, HWND  
hParent, ParamMap2UserDlgProc *proc=NULL);
```

**Remarks:**

This function is available in release 4.0 and later only.

This function creates a parameter map that will handle a parameter block in a modeless dialog where time does not change and the viewports are not redrawn. Note that there is no need to destroy it. It executes the dialog and then destroys itself. This function is currently not in use.

**Parameters:****IParamBlock2 \*pb**

Points to the parameter block2.

**TimeValue t**

The time at which the dialog is launched.

**HINSTANCE hInst**

The plug-ins instance handle.

**DLGTEMPLATE \*dlgTemplate**

The dialog template.

**HWND hParent**

The parent window.

**ParamMap2UserDlgProc \*proc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**Return Value:**

Returns TRUE if the user selected OK, FALSE otherwise.

**Function:**

```
BOOL CreateModalParamMap2(MapID map_id, IParamBlock2  
*pb, TimeValue t, HINSTANCE hInst, TCHAR *dlgTemplate,  
HWND hParent, ParamMap2UserDlgProc *proc=NULL);
```

**Remarks:**

This function is available in release 4.0 and later only.

Creates a parameter map that will handle a parameter block in a modeless dialog where time does not change and the viewports are not redrawn. This overload of **CreateModalParamMap2()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

**Function:**

```
IParamMap2 *CreateCPPParamMap2(IParamBlock2 *pb,  
Interface *ip, HINSTANCE hInst, TCHAR *dlgTemplate,  
TCHAR *title, DWORD flags, ParamMap2UserDlgProc*  
dlgProc=NULL, HWND hOldRollup=NULL, int category =  
ROLLUP_CAT_STANDARD);
```

**Remarks:**

This function creates a parameter map to handle the display of parameters in the command panel. This will add a rollup page to the command panel (or optionally replace an existing one).

Note, in version 4.0 and later, this actually maps to a call on the explicit map

ID overload of **CreateCPPParamMap2()** with default map ID of 0.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**Interface \*ip**

Pass in the plug-ins interface pointer.

**HINSTANCE hInst**

The plug-ins instance handle.

**TCHAR \*dlgTemplate**

The dialog template.

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A flag to control the settings of the rollup page:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**ParamMap2UserDlgProc\* dlgProc=NULL,**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**HWND hOldRollup=NULL**

If non-NULL specifies an existing rollup window in the current UI context that should be replaced with the newly created rollup for this map.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the

**RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

Returns a pointer to the parameter map2.

**Function:**

```
IParamMap2 *CreateCPPParamMap2(IParamBlock2 *pb,  
Interface *ip, HINSTANCE hInst, DLGTEMPLATE  
*dlgTemplate, TCHAR *title, DWORD flags,  
ParamMap2UserDlgProc* dlgProc=NULL, HWND  
hOldRollup=NULL, int category = ROLLUP_CAT_STANDARD);
```

**Remarks:**

This function is available in release 4.0 and later only.

This function creates a parameter map to handle the display of parameters in the command panel. This will add a rollup page to the command panel (or optionally replace an existing one). This function is currently not used.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**Interface \*ip**

Pass in the plug-ins interface pointer.

**HINSTANCE hInst**

The plug-ins instance handle.

**TCHAR \*dlgTemplate**

The dialog template.

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A flag to control the settings of the rollup page:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**ParamMap2UserDlgProc\* dlgProc=NULL,**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**HWND hOldRollup=NULL**

If non-NULL specifies an existing rollup window in the current UI context that should be replaced with the newly created rollup for this map.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the

**RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

Returns a pointer to the parameter map2.

**Function:**

```
IParamMap2 *CreateCPPParamMap2(MapID map_id,  
IParamBlock2 *pb, Interface *ip, HINSTANCE hInst, TCHAR  
*dlgTemplate, TCHAR *title, DWORD flags,  
ParamMap2UserDlgProc* dlgProc=NULL, HWND  
hOldRollup=NULL);
```

**Remarks:**

This function is available in release 4.0 and later only.

Creates a parameter map to handle the display of parameters in the command panel. This overload of **CreateCPPParamMap2()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

**Function:**

```
void DestroyCPPParamMap2(IParamMap2 *m);
```

**Remarks:**

This function destroys a parameter map created by **CreateCParamMap2()**. The rollup page is removed and the parameter map is deleted.

**Parameters:**

**IParamMap2 \*m**

A pointer to the parameter map2 to delete.

**Function:**

```
IParamMap2 *CreateChildCPPParamMap2(IParamBlock2 *pb,  
Interface *ip, HINSTANCE hInst, IParamMap2* parent, TCHAR  
*dlgTemplate, TCHAR *title, ParamMap2UserDlgProc*  
dlgProc=NULL);
```

**Remarks:**

This function creates a child dialog of the given parent parammap (for tabbed dialogs, etc.). This version takes an extra parent IParamMap2\* and creates a child dialog window in the parent parammap's window (rather than a new rollup) that is mapped by the new parammap. Developers need to call this explicitly once the parent parammap has been created as child parammaps are not created automatically by the **P\_AUTO\_UI** mechanisms.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **CreateChildCPPParamMap2()** with default map ID of 0.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**Interface \*ip**

Pass in the plug-ins rendering parameters interface pointer.

**HINSTANCE hInst**

The plug-ins instance handle.

**IParamMap2\* parent**

The parent parameter map.

**TCHAR \*dlgTemplate**

The dialog template.

**TCHAR \*title**

The title displayed.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**Return Value:**

Returns a pointer to the parameter map2.

**Function:**

**IParamMap2 \*CreateChildCPPParamMap2(IParamBlock2 \*pb, Interface \*ip, HINSTANCE hInst, IParamMap2\* parent, DLGTEMPLATE \*dlgTemplate, TCHAR \*title, ParamMap2UserDlgProc\* dlgProc=NULL);**

**Remarks:**

This function is available in release 4.0 and later only.

This function creates a child dialog of the given parent parammap (for tabbed dialogs, etc.). This version takes an extra parent IParamMap2\* and creates a child dialog window in the parent parammap's window (rather than a new rollout) that is mapped by the new parammap. Developers need to call this explicitly once the parent parammap has been created as child parammaps are not created automatically by the **P\_AUTO\_UI** mechanisms. This function is currently not used.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

### **Interface \*ip**

Pass in the plug-ins rendering parameters interface pointer.

### **HINSTANCE hInst**

The plug-ins instance handle.

### **IParamMap2\* parent**

The parent parameter map.

### **TCHAR \*dlgTemplate**

The dialog template.

### **TCHAR \*title**

The title displayed.

### **ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

### **Return Value:**

Returns a pointer to the parameter map2.

### **Function:**

```
IParamMap2 *CreateChildCPPParamMap2(MapID map_id,  
IParamBlock2 *pb, Interface *ip, HINSTANCE hInst,  
IParamMap2* parent, TCHAR *dlgTemplate, TCHAR *title,  
ParamMap2UserDlgProc* dlgProc=NULL);
```

### **Remarks:**

This function is available in release 4.0 and later only.

Create a child dialog of the given parent command panel parammap (for tabbed dialogs, etc.) This overload of **CreateChildCPPParamMap2()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

### **Function:**

```
void DestroyChildCPPParamMap2(IParamMap2 *m);
```

### **Remarks:**

This function destroys a parameter map created by **CreateChildCPPParamMap2()**.

**Parameters:**

**IParamMap2 \*m**

Points to the parameter map2 to destroy.

**Function:**

**IParamMap2 \*CreateRParamMap2(IParamBlock2 \*pb, IRenderParams \*ip, HINSTANCE hInst, TCHAR \*dlgTemplate, TCHAR \*title, DWORD flags, ParamMap2UserDlgProc\* dlgProc=NULL, int category = ROLLUP\_CAT\_STANDARD);**

**Remarks:**

Creates a parameter map to handle the display of render parameters or atmospheric plug-in parameters.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **CreateRParamMap2()** with default map ID of 0.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**IRenderParams \*ip**

Pass in the plug-ins rendering parameters interface pointer. See [Class IRenderParams](#).

**HINSTANCE hInst**

The plug-ins instance handle.

**TCHAR \*dlgTemplate**

The dialog template.

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A flag to control the settings of the rollup page:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

## **ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

## **int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USERREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

### **Return Value:**

Returns a pointer to the parameter map2.

### **Function:**

```
IParamMap2 *CreateRParamMap2(IParamBlock2 *pb,  
IRenderParams *ip, HINSTANCE hInst, DLGTEMPLATE  
*dlgTemplate, TCHAR *title, DWORD flags,  
ParamMap2UserDlgProc* dlgProc=NULL, int category =  
ROLLUP_CAT_STANDARD);
```

### **Remarks:**

Creates a parameter map to handle the display of render parameters or atmospheric plug-in parameters.

### **Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**IRendParams \*ip**

Pass in the plug-ins rendering parameters interface pointer. See [Class IRendParams](#).

**HINSTANCE hInst**

The plug-ins instance handle.

**DLGTEMPLATE \*dlgTemplate**

The dialog template.

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A flag to control the settings of the rollup page:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

Returns a pointer to the parameter map2.

**Function:**

```
IParamMap2 *CreateRParamMap2(MapID map_id,  
IParamBlock2 *pb, IRendParams *ip, HINSTANCE hInst,  
TCHAR *dlgTemplate, TCHAR *title, DWORD flags,  
ParamMap2UserDlgProc* dlgProc=NULL);
```

**Remarks:**

This function is available in release 4.0 and later only.

Creates a parameter map to handle the display of render parameters or atmospheric plug-in parameters. This overload of **CreateRParamMap2()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

**Function:**

```
void DestroyRParamMap2(IParamMap2 *m);
```

**Remarks:**

This function is available in release 3.0 and later only.

This function destroys a parameter map created by **CreateRParamMap2()**. The rollup page is removed and the parameter map is deleted.

**Parameters:**

```
IParamMap2 *m
```

Points to the parameter map2 to destroy.

**Function:**

```
IParamMap2* CreateChildRParamMap2(IParamBlock2 *pb,  
IRendParams *ip, HINSTANCE hInst, IParamMap2* parent,  
TCHAR *dlgTemplate, TCHAR *title, ParamMap2UserDlgProc*  
dlgProc=NULL);
```

**Remarks:**

This function create a parameter map for render or atmospheric parameters in a child dialog window of the given parammap, used typically to create tab child windows in a tabbed rollout. This version takes an extra parent **IParamMap2\*** and creates a child dialog window in the parent parammap's

window (rather than a new rollup) that is mapped by the new parammap. Developers need to call this explicitly once the parent parammap has been created as child parammaps are not created automatically by the **P\_AUTO\_UI** mechanisms.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **CreateChildRParamMap2()** with default map ID of 0.

#### **Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**IRendParams \*ip**

Pass in the plug-ins material parameters interface pointer. See [Class IRendParams](#).

**HINSTANCE hInst**

The plug-ins instance handle.

**IParamMap2\* parent**

The parent parameter map.

**TCHAR \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the dialog.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

#### **Return Value:**

Returns a pointer to the parameter map2 created.

#### **Function:**

```
IParamMap2* CreateChildRParamMap2(IParamBlock2 *pb,  
IRendParams *ip, HINSTANCE hInst, IParamMap2* parent,  
DLGTEMPLATE *dlgTemplate, TCHAR *title,  
ParamMap2UserDlgProc* dlgProc=NULL);
```

#### **Remarks:**

This function is available in release 4.0 and later only.

This function create a parameter map for render or atmospheric parameters in a child dialog window of the given parent parammap, used typically to create tab child windows in a tabbed rollout. This version takes an extra parent `IParamMap2*` and creates a child dialog window in the parent parammap's window (rather than a new rollup) that is mapped by the new parammap. Developers need to call this explicitly once the parent parammap has been created as child parammaps are not created automatically by the `P_AUTO_UI` mechanisms. This function is currently not being used.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**IRendParams \*ip**

Pass in the plug-ins material parameters interface pointer. See [Class IRendParams](#).

**HINSTANCE hInst**

The plug-ins instance handle.

**IParamMap2\* parent**

The parent parameter map.

**DLGTEMPLATE \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the dialog.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from `ParamMap2UserDlgProc` and set it as the parameter map's user callback.

**Return Value:**

Returns a pointer to the parameter map2 created.

**Function:**

**IParamMap2\* CreateChildRParamMap2(MapID map\_id, IParamBlock2 \*pb, IRendParams \*ip, HINSTANCE hInst, IParamMap2\* parent, TCHAR \*dlgTemplate, TCHAR \*title,**

**ParamMap2UserDlgProc\* dlgProc=NULL);**

**Remarks:**

This function is available in release 4.0 and later only.

Creates a parameter map for render or atmospheric parameters in a child dialog window of the given parent parammap. This overload of

**CreateChildRParamMap2()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

**Function:**

**void DestroyChildRParamMap2(IParamMap2 \*m);**

**Remarks:**

This function destroys a parameter map created by **CreateChildRParamMap2()**.

**Parameters:**

**IParamMap2 \*m**

Points to the parameter map2 to destroy.

**Function:**

**IParamMap2 \*CreateMParamMap2(IParamBlock2 \*pb, IMtlParams \*ip, HINSTANCE hInst, HWND hmedit, TexDADMgr\* dad, MtlDADMgr\* mdad, TCHAR \*dlgTemplate, TCHAR \*title, DWORD flags, ParamMap2UserDlgProc\* dlgProc=NULL, HWND hOldRollup=NULL, int category = ROLLUP\_CAT\_STANDARD);**

**Remarks:**

Creates a parameter map to handle the display of texture map or material parameters in the material editor.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **CreateMParamMap2()** with default map ID of 0.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**IMtlParams \*ip**

Pass in the plug-ins material parameters interface pointer. See [Class IMtlParams](#).

**HINSTANCE hInst**

The plug-ins instance handle.

**HWND hmedit**

The window handle to the materials editor.

**TexDADMgr\* dad**

Points to the manager used to handle drag and drop of textures. See [Class TexDADMgr](#).

**MtlDADMgr\* mdad**

Points to the manager used to handle drag and drop of materials. See [Class MtlDADMgr](#).

**TCHAR \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A flag to control the settings of the rollup page:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**HWND hOldRollup=NULL**

If non-NULL specifies an existing rollup window in the current UI context that should be replaced with the newly created rollup for this map.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there

exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

#### **Return Value:**

Returns a pointer to the parameter map2 created.

#### **Function:**

```
IParamMap2 *CreateMParamMap2(IParamBlock2 *pb,  
IMtlParams *ip, HINSTANCE hInst, HWND hmedit,  
TexDADMgr* dad, MtlDADMgr* mdad, DLGTEMPLATE  
*dlgTemplate, TCHAR *title, DWORD flags,  
ParamMap2UserDlgProc* dlgProc=NULL, HWND  
hOldRollup=NULL, int category = ROLLUP_CAT_STANDARD);
```

#### **Remarks:**

This function is available in release 4.0 and later only.

Creates a parameter map to handle the display of texture map or material parameters in the material editor. This function is currently not in use.

#### **Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**IMtlParams \*ip**

Pass in the plug-ins material parameters interface pointer. See [Class IMtlParams](#).

**HINSTANCE hInst**

The plug-ins instance handle.

**HWND hmedit**

The window handle to the materials editor.

**TexDADMgr\* dad**

Points to the manager used to handle drag and drop of textures. See [Class TexDADMgr](#).

**MtlDADMgr\* mdad**

Points to the manager used to handle drag and drop of materials. See [Class MtlDADMgr](#).

**DLGTEMPLATE \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the rollup page title bar.

**DWORD flags**

A flag to control the settings of the rollup page:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**HWND hOldRollup=NULL**

If non-NULL specifies an existing rollup window in the current UI context that should be replaced with the newly created rollup for this map.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the

**RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags

argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

Returns a pointer to the parameter map2 created.

**Function:**

```
IParamMap2 *CreateMParamMap2(MapID map_id,  
IParamBlock2 *pb, IMtlParams *ip, HINSTANCE hInst, HWND  
hmedit, TexDADMgr* dad, MtlDADMgr* mdad, TCHAR  
*dlgTemplate, TCHAR *title, DWORD flags,  
ParamMap2UserDlgProc* dlgProc=NULL, HWND  
hOldRollup=NULL);
```

**Remarks:**

This function is available in release 4.0 and later only.

Creates a parameter map to handle the display of texture map or material parameters in the material editor. This overload of **CreateMParamMap2()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

**Function:**

```
void DestroyMParamMap2(IParamMap2 *m);
```

**Remarks:**

This function destroys a parameter map created by **CreateMParamMap2()**. The rollup page is removed and the parameter map is deleted.

**Parameters:**

```
IParamMap2 *m
```

Points to the parameter map2 to destroy.

**Function:**

```
IParamMap2 *CreateChildMParamMap2(IParamBlock2 *pb,  
IMtlParams *ip, HINSTANCE hInst, IParamMap2* parent,  
TexDADMgr* tdad, MtlDADMgr* mdad, TCHAR *dlgTemplate,
```

**TCHAR \*title, ParamMap2UserDlgProc\* dlgProc=NULL);**

**Remarks:**

This function creates and returns a child dialog of the given parent parammap (for tabbed dialogs, etc.). This version takes an extra parent IParamMap2\* and create a child dialog window in the parent parammap's window (rather than a new rollup) that is mapped by the new parammap. Developers need to call this explicitly once the parent parammap has been created as child parammaps are not created automatically by the **P\_AUTO\_UI** mechanisms.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **CreateChildMParamMap2()** with default map ID of 0.

**Parameters:**

**IParamBlock2 \*pb**

Points to the parameter block2.

**IMtlParams \*ip**

Pass in the plug-ins material parameters interface pointer. See [Class IMtlParams\\_](#).

**HINSTANCE hInst**

The plug-ins instance handle.

**IParamMap2\* parent**

The parent parameter map.

**TexDADMgr\* tdad**

Points to the manager used to handle drag and drop of textures. See [Class TexDADMgr](#).

**MtlDADMgr\* mdad**

Points to the manager used to handle drag and drop of materials. See [Class MtlDADMgr\\_](#).

**TCHAR \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the dialog.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**Return Value:**

Returns a pointer to the parameter map2 created.

**Function:**

```
IParamMap2 *CreateChildMParamMap2(IParamBlock2 *pb,  
IMtlParams *ip, HINSTANCE hInst, IParamMap2* parent,  
TexDADMgr* tdad, MtlDADMgr* mdad, TCHAR *dlgTemplate,  
TCHAR *title, ParamMap2UserDlgProc* dlgProc=NULL);
```

**Remarks:**

This function is available in release 4.0 and later only.

This function creates and returns a child dialog of the given parent parammap (for tabbed dialogs, etc.). This version takes an extra parent IParamMap2\* and create a child dialog window in the parent parammap's window (rather than a new rollup) that is mapped by the new parammap. Developers need to call this explicitly once the parent parammap has been created as child parammaps are not created automatically by the **P\_AUTO\_UI** mechanisms. This function is currently not being used.

**Parameters:****IParamBlock2 \*pb**

Points to the parameter block2.

**IMtlParams \*ip**

Pass in the plug-ins material parameters interface pointer. See [Class IMtlParams](#).

**HINSTANCE hInst**

The plug-ins instance handle.

**IParamMap2\* parent**

The parent parameter map.

**TexDADMgr\* tdad**

Points to the manager used to handle drag and drop of textures. See [Class TexDADMgr](#).

**MtlDADMgr\* mdad**

Points to the manager used to handle drag and drop of materials. See [Class MtlDADMgr](#).

**DLGTEMPLATE \*dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR \*title**

The title displayed in the dialog.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**Return Value:**

Returns a pointer to the parameter map2 created.

**Function:**

**IParamMap2 \*CreateChildMParamMap2(IParamBlock2 \*pb, IMtlParams \*ip, HINSTANCE hInst, IParamMap2\* parent, TexDADMgr\* tdad, MtlDADMgr\* mdad, TCHAR \*dlgTemplate, TCHAR \*title, ParamMap2UserDlgProc\* dlgProc=NULL);**

**Remarks:**

This function is available in release 4.0 and later only.

Creates and returns a child dialog of the given material or texture map parent parammap (for tabbed dialogs, etc.) This overload of

**CreateChildMParamMap2()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

**Function:**

**void DestroyChildMParamMap2(IParamMap2 \*m);**

**Remarks:**

This function destroys a parameter map created by **CreateChildMParamMap2()**.

**Parameters:**

**IParamMap2 \*m**

Points to the parameter map2 to destroy.

**Function:**

**BOOL CreateChildModalParamMap2(MapID map\_id,**

**IParmBlock2 \*pb, HINSTANCE hInst, IParmMap2 \*parent, TCHAR \*dlgTemplate, TCHAR \*title, ParamMap2UserDlgProc \*proc=NULL);**

**Remarks:**

This function is available in release 4.0 and later only.

This function creates a child dialog of the given modal parent parammap (for tabbed dialogs, etc.). Unlike modal parent parammap, you do not need to destroy it. This version takes an extra parent IParmMap2\* and creates a child dialog window in the parent parammap's window (rather than a new rollup) that is mapped by the new parammap. Developers need to call this explicitly once the parent parammap has been created as child parammaps are not created automatically by the **P\_AUTO\_UI** mechanisms.

**Parameters:**

**MapID map\_id**

Specifies the ID of the parameter map/rollup to be created for this particular parameter block.

**IParmBlock2 \*pb**

Points to the parameter block2.

**HINSTANCE hInst**

The plug-ins instance handle.

**IParmMap2\* parent**

The parent parameter map.

**TCHAR \*dlgTemplate**

The dialog template.

**TCHAR \*title**

The title displayed in the dialog.

**ParamMap2UserDlgProc \*proc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**Return Value:**

Returns TRUE if the user selected OK, FALSE otherwise.

**Function:**

**BOOL CreateChildModalParamMap2(IParamBlock2 \*pb,  
HINSTANCE hInst, IParamMap2 \*parent, TCHAR  
\*dlgTemplate, TCHAR \*title, ParamMap2UserDlgProc  
\*proc=NULL);**

**Remarks:**

This function is available in release 4.0 and later only.

This is an overload that simply calls the explicit map ID version of **CreateChildMParamMap2()** with default map ID of 0. See that version for description.

**Function:**

**void DestroyChildModalParamMap2(IParamMap2 \*m);**

**Remarks:**

This function destroys a parameter map created by **CreateChildModalParamMap2()**.

**Parameters:**

**IParamMap2 \*m**

Points to the parameter map2 to destroy.

# Class Face

See Also: [Class Mesh](#).

class Face

## Description:

This class represents a single triangular face. The class maintains three indices into the vertex list for the face, a 32-bit smoothing group for the face, and 32-bits of face flags. The flags also store information about the visibility of the face, the visibility of the three edges, and whether or not the face has texture vertices present. The most significant 16-bits of the face flags store the material index. All methods of this class are implemented by the system.

## Data Members:

public:

**DWORD v[3];**

These are 0 based indices into a mesh object's array of vertices.

**DWORD smGroup;**

Smoothing group bits for the face.

Each bit of this 32 bit value represents membership in a smoothing group. The least significant bit represents smoothing group #1 while the most significant bit represents group #32. If two adjacent faces are assigned the same smoothing group bit, the edge between them is rendered smoothly.

**DWORD flags;**

The Face Flags:

Edge visibility bits: If the bit is 1, the edge is visible.

**EDGE\_A**

**EDGE\_B**

**EDGE\_C**

**EDGE\_ALL (EDGE\_A|EDGE\_B|EDGE\_C)**

Face visibility bit: If the bit is 1, the face is hidden.

**FACE\_HIDDEN**

Texture vertices bit: If the bit is 1, texture vertices are present. **FROM R5 THIS IS NOW OBSOLETE. YOU CAN USE MAPSUPPORT METHODS INSTEAD**

## **HAS\_TVERTS**

The material ID is stored in the HIWORD of the face flags.

## **FACE\_MATID\_SHIFT**

This is the number of bits to shift the flags to access the material.

## **FACE\_MATID\_MASK**

This is a mask used to access the material ID.

### **Methods:**

#### **Prototype:**

**Face()**

#### **Remarks:**

Constructor. The smoothing groups and face flags are initialized to zero.

#### **Prototype:**

**MtlID getMatID()**

#### **Remarks:**

Retrieves the zero based material ID for this face. Note: **typedef unsigned short MtlID;**

#### **Prototype:**

**void setMatID(MtlID id)**

#### **Remarks:**

Sets the material ID for this face.

#### **Parameters:**

**MtlID id**

Specifies the zero based material index.

#### **Prototype:**

**void setSmGroup(DWORD i)**

#### **Remarks:**

Sets the smoothing group bits for this face.

#### **Parameters:**

**DWORD i**

Specifies the smoothing group bits for this face.

**Prototype:****DWORD getSmGroup()****Remarks:**

Returns the smoothing group bits for this face.

**Prototype:****void setVerts(DWORD \*vrt)****Remarks:**

Sets the vertices of this face.

**Parameters:****DWORD \*vrt**

An array of the 3 vertices to store. These are zero based indices into the mesh object's array of vertices.

**Prototype:****void setVerts(int a, int b, int c)****Remarks:**

Sets the vertices of this face. The specified indexes are zero based indices into the mesh object's array of vertices.

**Parameters:****int a**

Specifies the first vertex.

**int b**

Specifies the second vertex.

**int c**

Specifies the third vertex.

**Prototype:****void setEdgeVis(int edge, int visFlag);**

**Remarks:**

Sets the visibility of the specified edge.

**Parameters:****int edge**

Specifies the edge to set the visibility of. You may use 0, 1, or 2.

**int visFlag**

One of the following values:

**EDGE\_VIS**

Sets the edge as visible.

**EDGE\_INVIS**

Sets the edge as invisible.

**Prototype:**

```
void setEdgeVisFlags(int va, int vb, int vc);
```

**Remarks:**

Sets the visibility of the all the edges.

**Parameters:****int va**

Specifies the visibility for edge 0. Use either **EDGE\_VIS** or **EDGE\_INVIS**.

**int vb**

Specifies the visibility for edge 1. Use either **EDGE\_VIS** or **EDGE\_INVIS**.

**int vc**

Specifies the visibility for edge 2. Use either **EDGE\_VIS** or **EDGE\_INVIS**.

**Prototype:**

```
int getEdgeVis(int edge)
```

**Remarks:**

Retrieves the edge visibility for the specified edge.

**Parameters:**

**int edge**

Specifies the edge.

**Return Value:**

Nonzero if the edge is visible, zero if the edge is invisible.

**Prototype:**

**DWORD getVert(int index)**

**Remarks:**

Returns the index into the mesh vertex array of the specified vertex.

**Parameters:**

**int index**

Specifies the vertex to retrieve. You may use 0, 1 or 2.

**Prototype:**

**DWORD \*getAllVerts()**

**Remarks:**

Retrieves a pointer to the vertex array.

**Return Value:**

A pointer to the vertex array.

**Prototype:**

**BOOL Hidden()**

**Remarks:**

Determines if the face is hidden or visible.

**Return Value:**

TRUE if the face is hidden; otherwise FALSE.

**Prototype:**

**void Hide()**

**Remarks:**

Hides this face (makes it invisible in the viewports).

**Prototype:**

**void Show()**

**Remarks:**

Shows this face (makes it visible in the viewports).

**Prototype:**

**void SetHide(BOOL hide)**

**Remarks:**

Sets the hidden state of this face.

**Parameters:**

**BOOL hide**

Specifies the hidden state for the face. Pass TRUE to hide the face; FALSE to show it.

**Prototype:**

**DWORD GetOtherIndex(DWORD v0, DWORD v1);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the first vertex in the face that isn't v0 or v1.

**Parameters:**

**DWORD v0**

The zero based index of one of the vertices to check.

**DWORD v1**

The zero based index of the other vertex to check.

**Return Value:**

The zero based index of the vertex found in the Mesh's vertex list.

**Prototype:**

**DWORD GetEdgeIndex(DWORD v0, DWORD v1);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the index of the edge in the face that goes from v0 to v1, or v1 to v0.

**Parameters:****DWORD v0**

The zero based index of the vertex at one end of the edge.

**DWORD v1**

The zero based index of the vertex at the other end of the edge.

**Return Value:**

The zero based index of the edge found in the Fesh's edge list.

**Prototype:**

```
int Direction(DWORD v0, DWORD v1);
```

**Remarks:**

This method is available in release 3.0 and later only.

Indicates order in which vertices v0 and v1 appear in the face.

**Parameters:****DWORD v0**

One vertex on this face.

**DWORD v1**

Another vertex on this face.

**Return Value:**

**1** if v1 follows v0 in sequence (This includes e.g. when Face::v[2] == v0 and Face::v[0] == v1.)

**-1** if v0 follows v1 in sequence

**0** if v0 or v1 are not on the face.

**Prototype:**

```
DWORD GetVertIndex(DWORD v0);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the index of the specified vertex in this face's vertex list (0, 1 or 2). If not found 3 is returned.

**Parameters:**

**DWORD v0**

The zero based index of the vertex to check.

**Prototype:**

```
void OrderVerts(DWORD &v0, DWORD &v1);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method switches v0,v1 if needed to put them in face-order. If v0 and v1 are in the order in which they appear in the face, or if one or both of them are not actually on the face, nothing happens. If however v0 follows v1, the values of the parameters are switched, so that they are then in the correct order for this face.

**Parameters:****DWORD &v0**

One vertex on this face.

**DWORD &v1**

Another vertex on this face.

**Sample Code:**

```
Face & f = mesh.faces[edge.f[0]];  
DWORD v0 = edge.v[0];  
DWORD v1 = edge.v[1];  
// Switch v0, v1 if needed to match orientation in selected face.  
f.OrderVerts(v0,v1);
```

## Class TVFace

See Also: [Class Mesh](#).

class TVFace

### Description:

This class is used for texture faces as well as vertex colors. The class maintains an array of three indices into the object's **tVerts** array. See the Mesh class for details on how its array of TVFaces and tVerts relate. All methods of this class are implemented by the system.

### Data Members:

public:

**DWORD t[3];**

These are indices into the mesh object's **tVerts** array.

### Methods:

#### Prototype:

**TVFace()**

#### Remarks:

Constructor. No initialization is done.

#### Prototype:

**TVFace(DWORD a, DWORD b, DWORD c);**

#### Remarks:

Constructor.

#### Parameters:

**DWORD a**

Specifies the index into the **tVerts** array for vertex 0.

**DWORD b**

Specifies the index into the **tVerts** array for vertex 1.

**DWORD c**

Specifies the index into the **tVerts** array for vertex 2.

**Prototype:**

**void setTVerts(DWORD \*vrt);**

**Remarks:**

Sets the texture vertices.

**Parameters:**

**DWORD \*vrt**

An array of indices into the **tVerts** array for vertices 0, 1, and 2.

**Prototype:**

**void setTVerts(int a, int b, int c);**

**Remarks:**

Sets the textured vertices.

**Parameters:**

**int a**

Specifies the index into the **tVerts** array for vertex 0.

**int b**

Specifies the index into the **tVerts** array for vertex 1.

**int c**

Specifies the index into the **tVerts** array for vertex 2.

**Prototype:**

**DWORD getTVert(int index);**

**Remarks:**

Retrieves one of the texture vertices.

**Parameters:**

**int index**

Specifies the index of the texture vertex to retrieve. You may use 0, 1 or 2.

**Return Value:**

The texture vertex.

**Prototype:**

**DWORD \*getAllTVerts();**

**Remarks:**

Returns a pointer to the array of texture vertices.

**Prototype:**

**DWORD GetVertIndex(DWORD v0);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the index of the specified texture vertex in this texture face's vertex list (0, 1 or 2). If not found 3 is returned.

**Parameters:**

**DWORD v0**

The zero based index of the texture vertex to check.

**Prototype:**

**DWORD GetOtherIndex(DWORD v0, DWORD v1);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the first texture vertex in this texture face that isn't v0 or v1.

**Parameters:**

**DWORD v0**

The zero based index of one of the vertices to check.

**DWORD v1**

The zero based index of the other vertex to check.

**Prototype:**

**int Direction(DWORD v0, DWORD v1);**

**Remarks:**

This method is available in release 3.0 and later only.

Indicates the order in which vertices v0 and v1 appear in the texture face.

**Parameters:****DWORD v0**

One vertex on this texture face.

**DWORD v1**

Another vertex on this texture face.

**Return Value:****1** if v1 follows v0 in sequence.**-1** if v0 follows v1 in sequence.**0** if v0 or v1 are not on the face.**Prototype:****void OrderVerts(DWORD &v0, DWORD &v1);****Remarks:**

This method is available in release 3.0 and later only.

This method switches v0,v1 if needed to put them in face-order. If v0 and v1 are in the order in which they appear in the texture face, or if one or both of them are not actually on the texture face, nothing happens. If however v0 follows v1, the values of the parameters are switched, so that they are then in the correct order for this texture face.

**Parameters:****DWORD &v0**

One vertex on this texture face.

**DWORD &v1**

Another vertex on this texture face.

# Class BitmapInfo

See Also: [Class Bitmap](#), [Working with Bitmaps](#), [List of Bitmap Error Codes](#).

class BitmapInfo

## Description:

This class describes the properties of a bitmap such as its path name or device name, width, height, gamma, number of frames, etc. Methods are available to set and retrieve these properties. All methods are implemented by the system unless noted otherwise.

## Method Groups:

The following hyperlinks jump to the start of groups of methods within the class:

[Flag Access](#)

[Get/Set Bitmap Properties \(Type, Width, Height, etc.\)](#)

[Get/Set Names, Devices.](#)

[Custom Flag Access](#)

[Get/Set Custom Bitmap Properties](#)

[Miscellaneous](#)

[Operators](#)

## Methods:

## Prototype:

**BitmapInfo();**

## Remarks:

Constructor. The following defaults are set by this constructor.

The width = 640;

The height = 480;

The custom width = 320;

The custom height = 200;

The custom flags = BMM\_CUSTOM\_RESFIT |

BMM\_CUSTOM\_FILEGAMMA;

The custom gamma = 1.0f;

The frame number = 0;

The aspect ratio = 1.0f;

The gamma setting = 1.0f;

The name and device name are set to NULL.  
The looping flag = BMM\_SEQ\_WRAP;

**Prototype:**

**BitmapInfo(TCHAR \*n);**

**Remarks:**

Constructor. The defaults are set as above excepting the bitmap name is set.

**Parameters:**

**TCHAR \*n**

The bitmap file name is set.

**Prototype:**

**BitmapInfo(const BitmapInfo &bi);**

**Remarks:**

This method is available in release 3.0 and later only.  
Copy Constructor.

**Parameters:**

**const BitmapInfo &bi**

The BitmapInfo to copy from.

## Flag Access

**Prototype:**

**DWORD Flags()**

**Remarks:**

Returns the flags of this BitmapInfo. See [Bitmap Flags](#).

**Prototype:**

**DWORD SetFlags(DWORD f)**

**Remarks:**

Sets the flags for this BitmapInfo. These are bitwise OR-ed into the current flags.

**Parameters:****DWORD f**

The flag bits to set. See [Bitmap Flags](#).

**Return Value:**

The revised flags are returned.

**Prototype:****BOOL TestFlags(DWORD f)****Remarks:**

Determines if a set of flag bits are set.

**Parameters:****DWORD f**

The flag bits to test. See [Bitmap Flags](#).

**Return Value:**

TRUE if the bits are set; otherwise FALSE.

**Get/Set Bitmap Properties****Prototype:****int Type()****Remarks:**

Returns the type property of this BitmapInfo. See [Bitmap Types](#).

**Prototype:****int SetType(int t)****Remarks:**

Sets the type property of this BitmapInfo to the specified value.

**Parameters:****int t**

Specifies the type of bitmap. See [Bitmap Types](#).

**Return Value:**

The old (previous) type setting.

**Prototype:**

**WORD Width()**

**Remarks:**

Returns the width (horizontal dimension) property of this BitmapInfo.

**Prototype:**

**WORD SetWidth(WORD w)**

**Remarks:**

Sets the width (horizontal dimension) property of this BitmapInfo.

**Parameters:**

**WORD w**

Specifies the width setting in pixels.

**Return Value:**

The old (previously set) width of the bitmap.

**Prototype:**

**WORD Height()**

**Remarks:**

Returns the height (vertical dimension) setting of this BitmapInfo.

**Prototype:**

**WORD SetHeight(WORD h)**

**Remarks:**

Sets the height (vertical dimension) property of this BitmapInfo.

**Parameters:**

**WORD h**

Specifies the height setting in pixels.

**Return Value:**

The old (previous) height setting.

**Prototype:**

## **float Gamma()**

### **Remarks:**

Returns the gamma setting property of this BitmapInfo.

### **Prototype:**

**float SetGamma(float g)**

### **Remarks:**

Sets the gamma property of this BitmapInfo to the value passed.

### **Parameters:**

**float g**

Specifies the gamma setting.

### **Return Value:**

The old (previous) gamma setting.

### **Prototype:**

**float Aspect()**

### **Remarks:**

Returns the aspect ratio property of this BitmapInfo.

### **Prototype:**

**float SetAspect(float a)**

### **Remarks:**

Set the aspect ratio property of this BitmapInfo to the specified value.

### **Parameters:**

**float a**

Specifies the aspect ratio setting.

### **Return Value:**

The old (previous) aspect ratio of the bitmap.

## **Frame Ranges**

### **Prototype:**

## **int FirstFrame()**

### **Remarks:**

Returns the first frame property of this BitmapInfo. Note that for a multi-frame bitmap some sequences may start with something other than 0.

### **Prototype:**

**int SetFirstFrame(int f)**

### **Remarks:**

Sets the first frame property of this BitmapInfo.

### **Parameters:**

**int f**

Specifies the first frame setting.

### **Return Value:**

The old (previous) first frame setting.

### **Prototype:**

**int LastFrame()**

### **Remarks:**

Returns the last frame property of this BitmapInfo.

### **Prototype:**

**int SetLastFrame(int f)**

### **Remarks:**

Sets the last frame property of this BitmapInfo.

### **Parameters:**

**int f**

Specifies the last frame.

### **Return Value:**

The old (previous) frame setting.

### **Prototype:**

## **int NumberFrames()**

### **Remarks:**

Returns the total number of frames setting of this BitmapInfo.

### **Prototype:**

## **int CurrentFrame()**

### **Remarks:**

Returns the current frame setting of this BitmapInfo.

### **Prototype:**

## **int SetCurrentFrame(int v)**

### **Remarks:**

Sets the current frame setting of this BitmapInfo.

### **Parameters:**

#### **int v**

Specifies the current frame.

### **Return Value:**

The old (previous) current frame setting.

### **Prototype:**

## **WORD SequenceOutBound()**

### **Remarks:**

When multi-frame BitmapIO loaders are reading a sequence of frames, this method is called to indicate what to do when reading beyond the end of available frames. The defaults is **BMM\_SEQ\_WRAP**.

### **Return Value:**

One of the following values:

#### **BMM\_SEQ\_WRAP**

Wraps around back to start point.

#### **BMM\_SEQ\_ERROR**

Generates an error if reading goes beyond the end.

#### **BMM\_SEQ\_PINGPONG**

This causes the sequence to turn around and goes the other direction, back and forth.

### **BMM\_SEQ\_HOLD**

When the last frame is reached it is held and used over and over.

#### **Prototype:**

**WORD SetSequenceOutBound(WORD s)**

#### **Remarks:**

Sets the sequence out of bounds property of this BitmapInfo. When reading a sequence of frames, this specifies what to do when reading beyond the end of available frames.

#### **Parameters:**

##### **WORD s**

One of the following values:

##### **BMM\_SEQ\_WRAP**

Wraps around back to start point.

##### **BMM\_SEQ\_ERROR**

Generates an error if reading goes beyond the end.

##### **BMM\_SEQ\_PINGPONG**

This causes the sequence to turn around and goes the other direction, back and forth.

##### **BMM\_SEQ\_HOLD**

When the last frame is reached it is held and used over and over.

#### **Return Value:**

The old (previous) value that was set.

### **Name/Device Access**

#### **Prototype:**

**const TCHAR \*Name()**

#### **Remarks:**

Returns the name property of this BitmapInfo. This is the full path name. See **TCHAR \*Filename()** for just the file name.

**Prototype:**

```
const TCHAR *SetName(const TCHAR *n);
```

**Remarks:**

Sets the name property of this BitmapInfo. When writing **n** should have a fully qualified filename. When reading, it only matters if the image is not in the MAP path. Note that a "feature" of the MAP path system is that if an image with same name is found more than once (in different paths), only the first one is seen.

Note: If loading an image from a device, make sure the name is empty (**bi.SetName(\_T(""))**);). This is automatic if you use

**BitmapManager::SelectDeviceInput()**. If you just create a **BitmapInfo** instance and set the device name by hand (**bi.SetDevice()**), this is also automatic as both name and device names are by default set to NULL ("").

This is only a concern if you reuse a **BitmapInfo** class previously used for image files.

**Parameters:**

```
const TCHAR *n
```

Specifies the name of the bitmap.

**Return Value:**

The old (previous) name that was set.

**Prototype:**

```
const TCHAR *Filename()
```

**Remarks:**

Returns just the file name of this BitmapInfo (not the entire path name).

**Prototype:**

```
const TCHAR *Device()
```

**Remarks:**

Returns the device name responsible for producing this image. For file types, this is just informative. For non-file types (devices) this is the way this image is identified. Therefore, it is important to save both name and device in order to properly identify an image.

**Prototype:**

**const TCHAR \*SetDevice(const TCHAR \*d);**

**Remarks:**

This method is used to set the device name.

**Parameters:**

**const TCHAR \*d**

The name to set.

**Return Value:**

The device name that was set.

**Prototype:**

**BOOL CompareName(BitmapInfo \*bi);**

**Remarks:**

This method will compare names taking in consideration both file names and device names. As devices don't have a file name, this method will first determine what type of image this is, and then perform a proper comparison.

**Parameters:**

**BitmapInfo \*bi**

The other BitmapInfo with which to compare names.

**Return Value:**

TRUE if the BitmapInfos have the same name and device name; otherwise FALSE.

## Custom Input Processing

**Prototype:**

**WORD CustWidth()**

**Remarks:**

Returns the custom width setting of this BitmapInfo.

**Prototype:**

**void SetCustWidth(WORD w)**

**Remarks:**

Sets the custom width setting for this BitmapInfo.

**Parameters:**

**WORD w**

The new custom width setting.

**Prototype:**

**WORD CustHeight()**

**Remarks:**

Returns the custom height setting of this BitmapInfo.

**Prototype:**

**void SetCustHeight(WORD h)**

**Remarks:**

Sets the custom height property of this BitmapInfo.

**Parameters:**

**WORD h**

The new custom height setting.

**Prototype:**

**int StartFrame()**

**Remarks:**

Returns the custom start frame property of this BitmapInfo.

**Prototype:**

**void SetStartFrame(int s)**

**Remarks:**

Sets the custom start frame property to the specified value.

**Parameters:**

**int s**

Specifies the start frame setting.

**Prototype:**

**int EndFrame()**

**Remarks:**

Returns the custom end frame setting of this BitmapInfo..

**Prototype:**

**void SetEndFrame(int e)**

**Remarks:**

Sets the custom end frame property of this BitmapInfo.

**Parameters:**

**int e**

The new end frame setting.

**Prototype:**

**WORD GetCustomX()**

**Remarks:**

Returns the custom x offset setting of this BitmapInfo.

**Prototype:**

**void SetCustomX(int x)**

**Remarks:**

Specifies the optional X coordinate (offset) property of this BitmapInfo. This specifies where to place the image if the image being copied from one Bitmap to another is smaller.

**Parameters:**

**int x**

Specifies the custom X offset.

**Prototype:**

**WORD GetCustomY()**

**Remarks:**

Returns the custom Y offset setting of this BitmapInfo.

**Prototype:**

**void SetCustomY(int y)**

**Remarks:**

Sets the optional Y coordinate (offset) property of this BitmapInfo. This specifies where to place the image if the image being copied from one Bitmap to another is smaller.

**Parameters:**

**int y**

Specifies the custom y offset.

**Prototype:**

**float GetCustomGamma()**

**Remarks:**

Returns the custom gamma setting of this BitmapInfo.

**Prototype:**

**void SetCustomGamma(float g)**

**Remarks:**

Sets a custom gamma setting of this BitmapInfo to the specified value.

**Parameters:**

**float g**

Specifies the custom gamma setting.

**Prototype:**

**int GetCustomStep()**

**Remarks:**

Returns the custom frame step setting of this BitmapInfo.

**Prototype:**

**void SetCustomStep(int s)**

**Remarks:**

Sets the custom frame increment setting of this BitmapInfo.

**Parameters:****int s**

Specifies the frame increment to use.

**Prototype:****int GetPresetAlignment()****Remarks:**

Returns the optional alignment setting of this BitmapInfo. This indicates where to place the image if the image being copied from one Bitmap to another is smaller.

**Return Value:**

See [List of Bitmap Alignment Positions](#).

**Prototype:****void SetPresetAlignment(int p)****Remarks:**

Establishes the optional alignment setting of this BitmapInfo. This specifies where to place the image if the image being copied from one Bitmap to another is smaller.

**Parameters:****int p**

Specifies one of the following nine values that define the position of the bitmap:

See [List of Bitmap Alignment Positions](#).

**Custom Input Flags****Prototype:****DWORD GetCustomFlags()****Remarks:**

Retrieves the custom flags setting of this BitmapInfo . See [List of Custom Bitmap Flags](#).

**Prototype:**

**void SetCustomFlag(DWORD f)**

**Remarks:**

Sets the custom flag(s) for this BitmapInfo.

**Parameters:**

**DWORD f**

Specifies the custom flags. See [List of Custom Bitmap Flags](#).

**Prototype:**

**void ResetCustomFlag(DWORD f)**

**Remarks:**

Clears the specified flag(s) of this BitmapInfo. See [List of Custom Bitmap Flags](#).

**Parameters:**

**DWORD f**

Specifies the flag bits to reset.

**Prototype:**

**BOOL TestCustomFlags(DWORD f)**

**Remarks:**

Tests the custom flags of this BitmapInfo. See [List of Custom Bitmap Flags](#).

**Parameters:**

**DWORD f**

The flag bits to test.

**Return Value:**

Returns TRUE if the specified flags were set; otherwise FALSE.

**Prototype:**

**void CopyImageInfo(BitmapInfo \*from);**

**Remarks:**

Copies the image information of the **from** BitmapInfo to this bitmap. Only the name, device and image characteristics are copied. User info, such as Custom

Width, etc. is not copied.

The following properties of the **from** BitmapInfo are copied:

**from->Name(),from->Device(),from->Width(), from->Height(),  
from->Aspect(), from->Gamma(),from->Type(), from->Flags(),  
from->FirstFrame(), from->LastFrame(), from->CurrentFrame(),  
from->GetCustomFlags()**

**Parameters:**

**BitmapInfo \*from**

The bitmap whose information will be copied.

**Prototype:**

**BOOL Validate()**

**Remarks:**

Implemented by the System.

This method is used to check the width, height, aspect ratio, and gamma settings to make sure they are within an acceptable range of values. The comparison is as follows:

```
if (width < 1 ||  
height < 1 ||  
aspect <= 0.0 ||  
gamma < MINGAMMA ||  
gamma > MAXGAMMA)  
    return (FALSE);  
else  
    return (TRUE);
```

Where:

```
#define MINGAMMA 0.2f  
#define MAXGAMMA 5.0f
```

**Return Value:**

TRUE if the BitmapInfo's settings are valid ; otherwise FALSE.

**Prototype:****HWND GetUpdateWindow()****Remarks:**

Returns the window handle to send progress or check abort messages to.

**Prototype:****void SetUpdateWindow(HWND hwnd)****Remarks:**

This is used internally - the system calls this method. This is how a window handle is sent down to device drivers and filters so they can send progress reports and check for cancel.

**Prototype:****void \*CreateFmtSpecBlock();****Remarks:**

This method is available in release 2.0 and later only.

This method provides some access to device specific data (for instance the compression ratio in a JPEG file). This method will return a buffer containing a given device specific data (or NULL if the device referenced is unknown or doesn't have "specific data"). The buffer structure will depend on the device. For all drivers shipped with the SDK, this structure is defined in their header files (which must be included in the project for which this method is used). Internally, this method validates the driver, calls its **EvaluateConfigure()** method to define the buffer size, creates this buffer and, if the returned size is greater than zero, calls the driver's **SaveConfigure()** method in order to set default values. The developer may then change whatever they want, create and write a file using this **BitmapInfo** which includes the device's specific data.

There is no need to free this buffer as this is handled by the **BitmapInfo** destructor.

Note: The name and/or device properties must be defined before using this method.

**Operators:**

**Prototype:**

**virtual BitmapInfo &operator= (BitmapInfo &from);**

**Remarks:**

Assignment operator. The data members of the specified BitmapInfo are copied to this BitmapInfo.

**Parameters:**

**BitmapInfo &from**

The source BitmapInfo.

## Class Bitmap

See Also: [Class BitmapManager](#), [Class BitmapInfo](#), [Class BitmapStorage](#), [Class GBuffer](#), [Working with Bitmaps](#).

```
class Bitmap : public BaseInterfaceServer
```

### Description:

The **Bitmap** class is the bitmap itself. All image access is done through this class. The **Bitmap** class has methods to retrieve properties of the bitmap such as image width, height, whether it is dithered, has an alpha channel, etc. There are methods to retrieve and store pixels from the image. Additional methods allow a developer to copy bitmaps. This class also has methods to open outputs and write multi-frame files. All methods of this class are implemented by the system.

### Data Members:

protected:

**float gamma;**

The gamma setting for the bitmap.

**Bitmap \*map;**

The bitmap using this output handler.

**BitmapStorage \*storage;**

The storage used by this INPUT handler

**int openMode;**

The mode the bitmap is open in.

**BitmapIO \*prevIO;**

**BitmapIO \*nextIO;**

Linked list pointers for multiple output of a single bitmap.

### Method Groups:

These hyperlinks jump to the start of groups of related methods within the class.

[Memory Deallocation](#)

[Flag Access](#)

[Bitmap Properties \(Size, Aspect, Gamma, etc.\)](#)

[Pixel Access](#)

[Palette Access](#)

[Copying / Cropping / Converting](#)

[Channel Access](#)

[Filtering / Dithering](#)

[Opening / Writing / Closing](#)

[Display \(Virtual Frame Buffer\) Methods](#)

[Storage Change Notificiation](#)

[Execute -- Generic Expansion Method](#)

**Methods:**

**Prototype:**

**inline BitmapManager \*Manager()**

**Remarks:**

Returns a pointer to the bitmap manager being used.

## **Memory Deallocation**

**Prototype:**

**void DeleteThis();**

**Remarks:**

This method is available in release 2.0 and later only.

This method should be called to free the Bitmap. Note that you should not invoke **~Bitmap()** directly by calling **delete** on the Bitmap as was done in 3ds max1.x. An assert will be raised if you call delete on a Bitmap directly. Instead use this method.

## **Flag Access**

**Prototype:**

**inline DWORD Flags()**

**Remarks:**

Returns the state of the bitmap flags. These flags describe properties of the bitmap such as if the bitmap is flipped horizontally or inverted vertically, is paletted, is dithered, etc. See [Bitmap Flags](#)

**Prototype:**

**inline void SetFlag(DWORD flag)**

**Remarks:**

Sets the specified flag bit(s).

**Parameters:**

**DWORD flag**

The flag(s) to set. See [Bitmap Flags](#).

**Prototype:**

**inline void ToggleFlag(DWORD flag)**

**Remarks:**

Toggles the specified flag bit(s) on/off.

**Parameters:**

**DWORD flag**

The flag(s) to toggle. See [Bitmap Flags](#).

**Prototype:**

**inline void ClearFlag(DWORD flag)**

**Remarks:**

Clears the specified flag bit(s) (sets them to zero).

**Parameters:**

**DWORD flag**

The flag(s) to clear. See [Bitmap Flags](#).

## **Bitmap Properties**

The following methods return properties of the bitmap. When these methods are called, they are passed through to the **BitmapStorage** instance maintained by the **Bitmap**. The **BitmapStorage** implements these methods by returning the properties of the **BitmapInfo** instance maintained by the storage. Therefore, the values returned from the methods represent what is returned from the **BitmapInfo** instance associated with the **Bitmap**.

**Prototype:**

### **inline int Width()**

**Remarks:**

Returns the width of the bitmap (the horizontal dimension).

**Return Value:**

If storage has been allocated the width of the bitmap; otherwise 0.

**Prototype:**

### **inline int Height()**

**Remarks:**

Returns the height (vertical dimension) of the bitmap.

**Return Value:**

If storage has been allocated the height of the bitmap; otherwise 0.

**Prototype:**

### **inline float Aspect()**

**Remarks:**

Returns the aspect ratio of the bitmap.

**Return Value:**

If storage has been allocated the aspect ratio of the bitmap; otherwise 0.0f.

**Prototype:**

### **inline float Gamma()**

**Remarks:**

Returns the gamma value for the bitmap.

**Return Value:**

If storage has been allocated the gamma of the bitmap; otherwise 0.0f.

**Prototype:**

### **inline int Paletted()**

**Remarks:**

Returns whether the bitmap uses a palette (is not true color).

**Return Value:**

If storage has been allocated returns nonzero if the bitmap uses a palette (returns the number of palette slots used); otherwise 0.

**Prototype:**

**inline int IsDithered()**

**Remarks:**

Returns whether the bitmap is dithered or not.

**Return Value:**

If storage has been allocated returns nonzero if the bitmap is dithered; otherwise 0.

**Prototype:**

**inline int PreMultipliedAlpha()**

**Remarks:**

Returns whether the bitmap uses pre-multiplied alpha.

**Return Value:**

If storage has been allocated returns nonzero if the bitmap uses pre-multiplied alpha; otherwise 0.

**Prototype:**

**inline int HasAlpha()**

**Remarks:**

Returns whether the bitmap has an alpha channel.

**Return Value:**

If storage has been allocated returns nonzero if the bitmap has an alpha channel; otherwise 0.

**Prototype:**

**inline int MaxRGBLevel()**

**Remarks:**

This method returns the number of bits per pixel for each color component.

For example a 24-bit TARGA has a **MaxRGBLevel()** of 8.

**Prototype:**

**virtual int MaxAlphaLevel() = 0;**

**Remarks:**

Implemented by the System.

Returns the number of bits per pixel in the alpha channel.

**Return Value:**

If storage has not been allocated returns 0.

**Prototype:**

**virtual int IsHighDynamicRange() = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns nonzero if this is a bitmap that supports high dynamic range data; zero if it doesn't.

**Prototype:**

**BMMRES GoTo(BitmapInfo \*bi);**

**Remarks:**

This method is used with multi-frame bitmaps (FLI's, AVI's, DDR devices, etc.). It is used to load a frame to replace a previously saved image. To define the desired frame, use:

**bi->SetCurrentFrame(frame);**

**Parameters:**

**BitmapInfo \*bi**

A pointer to the BitmapInfo. The frame number information is passed here.

**Return Value:**

If used with single frame drivers or if the driver doesn't support this function, it returns **BMMRES\_SINGLEFRAME**. If the return value is **BMMRES\_SUCCESS**, a new frame has been loaded into the given bitmap.

## Standard Pixel Access Methods.

Note: The following methods access pixel data one scanline at a time. Thus **x+pixels** must be less than the bitmap width.

### Prototype:

```
inline int Get16Gray(int x,int y,int pixels,float *ptr)
```

### Remarks:

Retrieves the specified 16-bit pixel values from the bitmap. Note: This method provides access to pixel data one scanline at a time.

### Parameters:

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**float \*ptr**

Pointer to storage for the retrieved pixel values.

### Return Value:

Returns nonzero if pixels were retrieved; otherwise 0. If storage has not been allocated 0 is returned.

### Prototype:

```
int Put16Gray(int x,int y,int pixels,float *ptr)
```

### Remarks:

Stores the specified 16-bit pixel values into the bitmap. The pixel value pointer you pass to this method may be freed or reused as soon as the function returns. Note: This method provides access to pixel data one scanline at a time.

### Parameters:

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**float \*ptr**

Pixel values to store.

**Return Value:**

Returns nonzero if pixels were stored; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**inline int GetPixels(int x,int y,int pixels,BMM\_Color\_fl \*ptr)**

**Remarks:**

Retrieves the specified 64-bit pixel values from the bitmap. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_fl \*ptr**

Pointer to storage for the retrieved pixel values. See [Structure BMM\\_Color\\_fl](#).

**Return Value:**

Returns nonzero if pixels were retrieved; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**int PutPixels(int x,int y,int pixels,BMM\_Color\_fl \*ptr)**

**Remarks:**

Stores the specified 64-bit pixel values into the bitmap's own local storage. The pointer you pass to this method may be freed or reused as soon as the function returns. Note: This method provides access to pixel data one scanline

at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**BMM\_Color\_fl \*ptr**

Pixel values to store. See [Structure BMM\\_Color\\_fl](#).

**Return Value:**

Returns nonzero if pixels were stored; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

```
inline int GetLinearPixels(int x,int y,int pixels,BMM_Color_64  
*ptr)
```

**Remarks:**

Retrieves the specified 64-bit pixel values from the bitmap. These pixels are NOT gamma corrected (i.e. they have linear gamma - 1.0). Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_64 \*ptr**

Pointer to storage for the retrieved pixel values. See [Structure BMM\\_Color\\_64](#).

**Return Value:**

Returns nonzero if pixels were retrieved; otherwise 0. If storage has not been

allocated 0 is returned.

**Prototype:**

**inline int GetIndexPixels(int x,int y,int pixels,BYTE \*ptr)**

**Remarks:**

Retrieves the specified pixels from the paletted bitmap. The palette for the image may be accessed using **GetPalette()**. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BYTE \*ptr**

Pointer to storage for the pixel values.

**Return Value:**

Returns nonzero if pixels were retrieved; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**inline int PutIndexPixels(int x,int y,int pixels,BYTE \*ptr)**

**Remarks:**

Stores the pixels into the specified location of the paletted bitmap. The pixel value pointer you pass to this method may be freed or reused as soon as the function returns. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**BYTE \*ptr**

Pixels to store.

**Return Value:**

Returns nonzero if pixels were stored; otherwise 0. If storage has not been allocated 0 is returned.

## Palette Access

**Prototype:**

**inline int GetPalette(int start,int count,BMM\_Color\_48 \*ptr)**

**Remarks:**

Retrieves a portion of the palette from the bitmap.

**Parameters:**

**int start**

The index into the palette of where to begin retrieving palette entries.

**int count**

The number of palette entries to retrieve.

**BMM\_Color\_48 \*ptr**

Storage for the palette entries. See [Structure BMM\\_Color\\_48](#).

**Return Value:**

Nonzero if the palette entries were retrieved; otherwise 0.

**Prototype:**

**inline int SetPalette(int start,int count,BMM\_Color\_48 \*ptr)**

**Remarks:**

Sets the specified portion of the palette of this bitmap.

**Parameters:**

**int start**

The index into the palette of where to begin storing palette entries.

**int count**

The number of palette entries to store.

**BMM\_Color\_48 \*ptr**

The palette entries to store. See [Structure BMM\\_Color\\_48](#).

**Return Value:**

Nonzero if the palette entries were stored; otherwise 0.

## Copying / Cropping Methods

**Prototype:**

```
int GetFiltered(float u, float v, float du, float dv, BMM_Color_fl  
*ptr);
```

**Remarks:**

This method uses summed area table or pyramidal filtering to compute an averaged color over the specified area. You must have a filter plugged in for this to work. See **SetFilter()** below.

**Parameters:**

**float u, float v**

The location in the bitmap to filter. These values go from 0.0 to 1.0 across the size of the bitmap.

**float du, float dv**

The size of the rectangle to sample. These values go from 0.0 to 1.0 across the size of the bitmap.

**BMM\_Color\_fl \*ptr**

The result is returned here -- the average over the specified area. See Structure BMM\_Color\_fl. **\*\*aztodo\*\*** link this

**Prototype:**

```
inline int CopyImage(Bitmap *from,int operation,BMM_Color_fl  
fillcolor, BitmapInfo *bi = NULL)
```

**Remarks:**

Copies the specified bitmap to this bitmap.

**Parameters:**

**Bitmap \*from**

The source bitmap.

### **int operation**

The type of copy to perform:

#### **COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

#### **COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

#### **COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

#### **COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

### **BMM\_Color\_fl fillcolor**

Vacant areas of the bitmap are filled with **fillcolor** pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap. See [Structure BMM\\_Color\\_fl](#).

### **BitmapInfo \*bi = NULL**

When using custom options (resize to fit, positioning, etc.) this is how the flags are passed down to the Bitmap Manager. This is an optional argument -- for simple copy operations, **\*bi** can default to NULL. If present, the code checks the option flags and acts accordingly.

### **Return Value:**

Nonzero if the copy was performed; otherwise 0.

### **Prototype:**

```
inline int CopyImage(Bitmap *from,int operation,int fillindex)
```

### **Remarks:**

Copies the specified bitmap to this bitmap.

### **Parameters:**

#### **Bitmap \*from**

The source bitmap.

#### **int operation**

The type of copy to perform. See [List of Copy Image Operations](#).

#### **int fillindex**

Vacant areas of the bitmap are filled with **fillindex** pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap.

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

**PBITMAPINFO ToDib(int depth = 24, UWORD \*gam=NULL, BOOL dither=FALSE);**

**Remarks:**

Creates a new Windows Device Independent Bitmap (DIB) and returns a pointer to it. The DIB bitmap is created from this Bitmap. The DIB is allocated using **LocalAlloc()**. The pseudo-code below show how one may be created and freed. Note that the DIB is never used or accessed inside 3ds max (the call to **ToDib()** is the first and last time that 3ds max sees this pointer):

```
PBITMAPINFO pDib;  
pDib = bitmap->ToDib();
```

...

When you are done using the DIB call:

```
LocalFree(pDib);
```

**Parameters:**

**int depth = 24**

The bitmap depth; either 24 (BGR) or 32 (BGRO). If not specified the default is 24.

**UWORD \*gam=NULL**

Specifies a pointer to an optional gamma table that is used to apply gamma correction to the color components as part of the conversion to a DIB. The table has RCOLN entries.

**BOOL dither=FALSE**

Specifies if a random dither is applied when reducing the color components from 16 bits per channel to 8 bits per channel (to reduce banding effects).

**Prototype:**

**BOOL FromDib(PBITMAPINFO pbmi);**

**Remarks:**

Converts the DIB to this bitmap's storage type. This bitmap's storage must already be allocated or the call will fail. The source must be 16, 24 or 32 bit. You cannot use an 8 bit DIB.

**Parameters:**

**PBITMAPINFO pbmi**

The source bitmap.

**Return Value:**

TRUE if the conversion was performed; otherwise FALSE.

**Prototype:**

**inline int ResizeImage(int width,int height,int newpalette)**

**Remarks:**

This method is not currently implemented.

## Channel Access

**Prototype:**

**inline void \*GetChannel(ULONG channelId, ULONG& chanType)**

**Remarks:**

Returns a pointer to the specified channel of the bitmap, and determines its type in terms of bits per pixel.

**Parameters:**

**ULONG channelId**

The channel to return a pointer to. See [List of Image Channels](#).

**ULONG& chanType**

The type of the returned channel. One of the following values:

**BMM\_CHAN\_TYPE\_UNKNOWN**

Channel not of a known type.

**BMM\_CHAN\_TYPE\_8**

1 byte per pixel

**BMM\_CHAN\_TYPE\_16**

1 word per pixel

**BMM\_CHAN\_TYPE\_32**

2 words per pixel

**BMM\_CHAN\_TYPE\_48**

3 words per pixel

**BMM\_CHAN\_TYPE\_64**

4 words per pixel

**BMM\_CHAN\_TYPE\_96**

6 words per pixel

**Prototype:**

**GBuffer \*GetGBuffer();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the GBuffer for the bitmap (or NULL if none).

**Prototype:**

**inline ULONG CreateChannels(ULONG channelIDs)**

**Remarks:**

This method creates the specified channels. After creation, these may be accessed using **void \*GetChannel()**.

**Parameters:**

**ULONG channelIDs**

Specifies the channels to create. See [List of Image Channels](#).

**Return Value:**

The channels that are present.

**Prototype:**

**inline void DeleteChannels(ULONG channelIDs)**

**Remarks:**

Delete the specified channels.

**Parameters:**

**ULONG channelIDs**

Specifies the channels to delete. See [List of Image Channels](#).

**Prototype:**

**inline ULONG ChannelsPresent()**

**Remarks:**

Returns the channels that are present. See [List of Image Channels](#).

**Prototype:**

**inline RenderInfo \*GetRenderInfo()**

**Remarks:**

This is used internally. It returns a pointer to the RenderInfo associated with the storage if available; otherwise NULL. See [Class RenderInfo](#).

**Prototype:**

**inline RenderInfo \*AllocRenderInfo()**

**Remarks:**

This is used internally. It returns a pointer to the RenderInfo instance allocated by the storage. If this could not be allocated NULL is returned. See [Class RenderInfo](#).

**Prototype:**

**BOOL PrepareGChannels(BitmapInfo \*bi);**

**Remarks:**

This method is used internally. This method will check with the plug-in (file or device) defined in the given BitmapInfo and prepare (create) the proper channels. If a given channel already exists, no new channel will be created. After creating a bitmap, use this function to define the optional channels that may be required by the given handler.

**Parameters:**

**BitmapInfo \*bi**

Points to an instance of BitmapInfo that defines the properties of the image.

**Return Value:**

TRUE if the channels were created; otherwise FALSE.

**Filtering / Dithering Methods****Prototype:**

```
int GetFiltered(float u, float v, float du, float dv, BMM_Color_fl  
*ptr);
```

**Remarks:**

This method uses summed area table or pyramidal filtering to compute an averaged color over the specified area. You must have a filter plugged in for this to work. See **SetFilter()** below.

**Parameters:**

**float u, float v**

The location in the bitmap to filter. These values go from 0.0 to 1.0 across the size of the bitmap.

**float du, float dv**

The size of the rectangle to sample. These values go from 0.0 to 1.0 across the size of the bitmap.

**BMM\_Color\_fl \*ptr**

The result is returned here -- the average over the specified area.

**Prototype:**

```
int SetFilter(UINT filterType);
```

**Remarks:**

Establishes a filtering algorithm to be used by the bitmap.

**Parameters:**

**UINT filterType**

See [List of Bitmap Filter Types](#).

**Return Value:**

Nonzero if the bitmap filtering was set; otherwise 0.

**Prototype:**

## **inline int HasFilter()**

### **Remarks:**

Determines if the bitmap has a filter.

### **Return Value:**

Returns nonzero if the bitmap has a filter; otherwise 0.

### **Prototype:**

**int SetDither(UINT ditherType);**

### **Remarks:**

Sets the type of dithering used on the bitmap.

### **Parameters:**

**UINT ditherType**

The type of dither to perform.

**BMM\_DITHER\_NONE**

Specifies no dithering is to be performed.

**BMM\_DITHER\_FLOYD**

Specifies the Floyd-Steinberg dithering algorithm.

### **Return Value:**

Nonzero if the bitmap dithering was set; otherwise 0.

### **Prototype:**

**inline BitmapFilter \*Filter()**

### **Remarks:**

This method is used internally. It returns a pointer to the bitmap filter used by the bitmap.

### **Prototype:**

**int SetStorage(BitmapStorage \*storage);**

### **Remarks:**

Establishes a bitmap storage to manage this bitmap.

### **Parameters:**

**BitmapStorage \*storage**

The storage to manage the bitmap.

**Return Value:**

Nonzero if the storage was assigned; otherwise 0.

**Prototype:**

**inline BitmapStorage \*Storage()**

**Remarks:**

Returns the storage that is managing this bitmap.

**Return Value:**

A pointer to the storage.

**Prototype:**

**inline void NullStorage();**

**Remarks:**

Sets the storage pointer to NULL.

## **Output / Writing / Closing Methods**

**Prototype:**

**BMMRES OpenOutput(BitmapInfo \*bi);**

**Remarks:**

This method will open the image for output. This allows the image to be written to. Note that you can pass a **BitmapInfo** to this method where you simply set the name 'by hand' (**bi->SetName()**). This will work and the correct driver will be selected based on the filename extension. However you won't be able to set any driver specific settings (such as compression settings for JPEGs). The alternative way is to use the **BitmapManger** methods.

**Parameters:**

**BitmapInfo \*bi**

Contains the name of the image or device to open for output.

**Return Value:**

**BMMRES\_SUCCESS**

Indicates success.

### **BMMRES\_ERRORTAKENCARE**

Indicates that 3ds max could not find a device to handle the image.

### **BMMRES\_INTERNALERROR**

Indicates the IO module handling the image could not be opened for writing.

### **Prototype:**

**BMMRES Write(BitmapInfo \*bi, DWORD frame =  
BMM\_SINGLEFRAME)**

### **Remarks:**

Write the image from the **BitmapStorage** to disk. Note that you must pass the same **BitmapInfo** used when the file was first "Opened for Output". The main reason is that any device specific settings are kept in the **BitmapInfo** object. Also, the custom options such as gamma value, optional channels, etc (if any) are kept in this **BitmapInfo** instance. These are assigned when the bitmap is first opened (for either read or write).

### **Parameters:**

#### **BitmapInfo \*bi**

Contains the name of the file or device to write to.

#### **DWORD frame = BMM\_SINGLEFRAME**

Specifies the frame number to write. If this is a single image, allow **frame** to default to single frame. This argument determines if the file will have the frame number appended to it. If you want the file to have a normal name (no frame number attached to it), you must set the frame argument to **BMM\_SINGLEFRAME**. Any other value is considered to be a frame number and it will be appended to the given filename.

### **Return Value:**

#### **BMMRES\_SUCCESS**

Indicates success.

#### **BMMRES\_ERRORTAKENCARE**

Indicates that 3ds max could not find a device to handle the image.

#### **BMMRES\_INTERNALERROR**

Indicates the IO module handling the image could not be opened for writing.

**Prototype:**

**BMMRES WriteAll(DWORD frame = BMM\_SINGLEFRAME);**

**Remarks:**

Write the image to all the open outputs.

**Parameters:**

**DWORD frame = BMM\_SINGLEFRAME**

Specifies the frame number to write. If this is a single image, allow **frame** to default to single frame.

**Return Value:**

**BMMRES\_SUCCESS**

Indicates success.

**BMMRES\_ERRORTAKENCARE**

Indicates that 3ds max could not find a device to handle the image.

**BMMRES\_INTERNALERROR**

Indicates the IO module handling the image could not be opened for writing.

**Prototype:**

**int Close(BitmapInfo \*bi, int flag = BMM\_CLOSE\_COMPLETE)**

**Remarks:**

Close the bitmap. This means the bitmap is no longer open for writing.

**Parameters:**

**BitmapInfo \*bi**

Identifies the bitmap to close.

**int flag = BMM\_CLOSE\_COMPLETE**

See [List of Bitmap Close Types](#).

**Return Value:**

Nonzero if the image output was closed without error; otherwise 0.

**Prototype:**

**int CloseAll(int flag = BMM\_CLOSE\_COMPLETE)**

**Remarks:**

Closes all the open outputs.

**Parameters:**

**int flag = BMM\_CLOSE\_COMPLETE**

See [List of Bitmap Close Types](#).

**Return Value:**

Nonzero if the image outputs were closed without error; otherwise 0.

**Prototype:**

**inline void UseScaleColors(int on);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Set whether colors are scaled (on) or clamped (off) when converting from **BMM\_Color\_fl** to **BMM\_Color\_64**. If storage is not allocated, does nothing.

**Prototype:**

**inline int ScaleColors();**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Returns the last value set by **UseScaleColors**. If storage is not allocated, returns 0.

**Prototype:**

**inline static void ClampColor(BMM\_Color\_64& out, const BMM\_Color\_fl& in);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out clamping the RGB components to 0 to 65535. The alpha component is not copied.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

```
inline static void ClampColorA(BMM_Color_64& out, const  
BMM_Color_fl& in);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out clamping the RGB components to 0 to 65535.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

```
inline void ScaleColor (BMM_Color_64& out, BMM_Color_fl in);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out clamping the RGB components to 0 to 65535. The alpha component is not copied.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

```
inline void ScaleColorA(BMM_Color_64& out, const  
BMM_Color_fl& in);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out clamping the RGB components to 0 to 65535.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

```
inline void ClampScaleColor (BMM_Color_64& out, const  
BMM_Color_fl& in);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out, using the value of **ScaleColors()** to determine the clamping or scaling. The alpha component is not copied. If the storage is not allocated, the clamping is performed.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

```
inline void ClampScaleColorA (BMM_Color_64& out, const  
BMM_Color_fl& in);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out, using the value of **ScaleColors()** to determine the clamping or scaling. If the storage is not allocated, the clamping is performed.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

## Display (Virtual Frame Buffer) Methods

**Prototype:**

```
int Display(TCHAR *title=NULL, int position=BMM_CN, BOOL  
autonomous=FALSE, BOOL savebutton=TRUE, CropCallback  
*crop=NULL, Bitmap *cloneMyVFB = NULL)
```

**Remarks:**

This method creates a window for the display of this bitmap and displays it.

**Parameters:**

**TCHAR \*title = NULL**

The title to display in the title bar of the window.

**int position = BMM\_CN**

Specifies how the bitmap should be positioned. One of the following values:

**BMM\_UL** - Upper Left

**BMM\_LL** - Lower Left

**BMM\_UR** - Upper Right

**BMM\_LR** - Lower Right

**BMM\_CN** - Center

**BMM\_RND** - Used internally. Renderer location.

**BMM\_VPP** - Used internally. Video Post Primary location.

**BMM\_VPS** - Used internally. Video Post Secondary location.

**BOOL autonomous = FALSE**

This is reserved for internal use, always let it default to FALSE.

**BOOL savebutton = TRUE**

This is reserved for internal use, always let it default to TRUE.

**CropCallback \*crop=NULL**

This parameter is available in release 2.0 and later only.

When non-NULL this will cause the VFB to display, instead of its normal toolbar, a set of sliders for adjusting cropping and also will allow interactive adjustment of the cropping rectangle in the image window. See [Class CropCallback](#).

**Bitmap \*cloneMyVFB = NULL**

This parameter is available in release 4.0 and later only.

A pointer to a bitmap to clone the VFB to.

**Return Value:**

Nonzero if the bitmap was displayed; otherwise 0.

**Prototype:**

**int UnDisplay();**

**Remarks:**

Close the display window associated with this bitmap (if any).

**Return Value:**

Always returns nonzero.

**Prototype:**

**BOOL IsAutonomousVFB();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the virtual frame buffer (VFB) is autoomous; otherwise FALSE. For instance, Video Post has an associated VFB. When Video Post is closed so is its VFB since it belongs to it. In that case the VFB is **not** autonomous. If the user does a View File command, that VFB is autonomous.

**Prototype:**

**HWND GetWindow();**

**Remarks:**

Get the window handle for the displayed bitmap.

**Return Value:**

Returns the window handle, or NULL if it's not displayed in a window.

**Prototype:**

**void RefreshWindow(RECT \*rect = NULL);**

**Remarks:**

Refreshes the interior of the display window with the bitmap contents. In release 3.0 and later this method respects the **Bitmap::ShowProgressLine()** setting. See that method for more details.

**Parameters:**

**RECT \*rect = NULL**

The region of the display window to refresh (specified in image coordinates). If the pointer is NULL the entire window is refreshed.

**Prototype:**

**void SetCroppingValues(float u, float v, float w, float h, BOOL placeImage);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used when the VFB is being displayed and you want to change the cropping rectangle from your plug-in. An example of this is available in **\MAXSDK\SAMPLES\MATERIALS\BMTEX.CPP**.

**Parameters:**

**float u**

The U value to set.

**float v**

The U value to set.

**float w**

The U value to set.

**float h**

The U value to set.

## **BOOL placeImage**

TRUE for place mode; FALSE for crop.

### **Prototype:**

```
void SetWindowTitle(TCHAR *title);
```

### **Remarks:**

Sets the title displayed in the display window's title bar.

### **Parameters:**

**TCHAR \*title**

The title to display.

## **Storage Change Notification**

### **Prototype:**

```
void SetNotify(BitmapNotify *bmnot=NULL);
```

### **Remarks:**

This method is available in release 2.0 and later only.

This method is used to set a callback to allow the developer to get notified if the storage for the Bitmap has changed.

### **Parameters:**

**BitmapNotify \*bmnot=NULL**

The pointer to the callback object implemented by the developer used to provide notification when the Bitmap's storage changes. See [Class BitmapNotify](#).

### **Prototype:**

```
BitmapNotify *GetNotify();
```

### **Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to the callback used to notify a developer when the Bitmap's storage changes.

### **Default Implementation:**

```
{ return bmNotify; };
```

## Execute -- Generic Expansion Method

### Prototype:

```
INT_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0,  
ULONG arg3=0);
```

### Remarks:

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

### Parameters:

#### **int cmd**

The index of the command to execute.

#### **ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

#### **ULONG arg2=0**

Optional argument 2.

#### **ULONG arg3=0**

Optional argument 3.

### Return Value:

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

## Handy built-in functions

### Prototype:

**int Fill(int r,int g,int b,int alpha);**

### Remarks:

Sets every pixel of the bitmap to the specified color and alpha value.

### Parameters:

**int r**

Specifies the red value to fill with.

**int g**

Specifies the green value to fill with.

**int b**

Specifies the blue value to fill with.

**int alpha**

Specifies the alpha value to fill with.

### Return Value:

Nonzero if the operation succeeded; otherwise FALSE.

### Prototype:

**void Print(bool silent = false);**

### Remarks:

This method is available in release 3.0 and later only.

This method is only supported in 3D Studio VIZ.

### Prototype:

**void ShowProgressLine(int y);**

### Remarks:

This method is available in release 3.0 and later only.

This method is used for showing a moving scanline in a virtual frame buffer displaying this bitmap. Here's how it works. If you call

**ShowProgressLine(y)**, it clears any previously set white line, and sets an internal counter: you have to call **Bitmap::RefreshWindow()** to get the

new white line to show up. From then that line will be displayed as white. To Clear it call **ShowProgressLine(-1)**.

**Parameters:**

**int y**

The scanline to display as white (the count begins at zero). Use a value of -1 to hide the line.

# Class BitmapManager

See Also: [Class BitmapInfo](#), [Class Bitmap](#), [Class BitmapIO](#), [Class BitmapStorage](#), [Working with Bitmaps](#), [List of Bitmap Error Codes](#).

```
class BitmapManager : public InterfaceServer
```

## Description:

This class is used to manage the use of bitmaps within 3ds max. There is a global instance of this class provided by 3ds max that developers may use to call these methods. It is called **TheManager**. This class provides methods for things such as creating and loading bitmaps, and access to the bitmap Map Path directories. There are also methods for displaying some general dialogs that let users select input and output files and devices, as well as dialogs for setting options for the bitmap such as its custom width, height and positioning.

Note: In the 3ds max release 3.0 SDK these methods were made virtual.

## Data Members:

public:

**BMMVfbPalette \*pal;**

This is used internally as the virtual framebuffer palette.

## Method Groups:

The hyperlinks below jump to the start of groups of related methods within the class

[New / Creating / Loading Methods](#)

[Display / VFB Related Methods](#)

[User Interface \(Dialog Box\) Methods](#)

[Host Access \(Window handles, Map path directory access, etc.\)](#)

[Error Processing](#)

[Future Expansion](#)

## Methods:

## Error Processing

### Prototype:

**BOOL SilentMode()**

**Remarks:**

Determines if silent mode is on. Silent mode specifies if developers should display error messages. If this method returns FALSE, error messages should be displayed. If TRUE, error message dialogs should not be shown.

**Return Value:**

Returns TRUE if silent mode is on; FALSE otherwise.

**Prototype:**

```
void SysLog(int type, char *format, ...);
```

**Remarks:**

This is reserved for future use.

**Prototype:**

```
BOOL SetSilentMode(BOOL s);
```

**Remarks:**

This method is used internally.

**Prototype:**

```
void SetLogLevel(DWORD level);
```

**Remarks:**

This method is used internally.

**Prototype:**

```
DWORD GetLogLevel();
```

**Remarks:**

This method is used internally.

## **Display / VFB Related Methods**

**Prototype:**

```
void RefreshAllVFBs();
```

**Remarks:**

This method is available in release 2.0 and later only.

This method refreshes the interior of all the virtual frame buffer windows with each bitmap's contents.

**Prototype:**

**void DeleteAllAutonomousVFBMaps();**

**Remarks:**

This method is available in release 2.0 and later only.

This method calls **Bitmap::DeleteThis()** on all the bitmaps whose virtual frame buffers are set to autonomous.

## Future Expansion

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:**

**int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

## Host Interface

### Prototype:

**virtual HINSTANCE AppInst();**

### Remarks:

Returns the application instance handle of 3ds max itself.

### Prototype:

**virtual HWND AppWnd();**

### Remarks:

Returns the window handle of 3ds max's main window.

### Prototype:

**virtual TCHAR \*GetDir(int i);**

### Remarks:

Implemented by the System.

Retrieves the specified standard 3ds max directory name (fonts, scenes, images, etc.).

### Parameters:

**int i**

Specifies the directory name to retrieve. See [List of Directory Names](#).

### Return Value:

The name of the specified directory.

### Prototype:

**virtual BOOL AddMapDir(TCHAR \*dir,int update);**

### Remarks:

For internal use only - This is used to add a MAP PATH to the Map path list.

### Prototype:

**virtual int GetMapDirCount();**

### Remarks:

Returns the number of map paths (used in conjunction with the method below).

**Return Value:**

The number of map paths.

**Prototype:**

**virtual TCHAR \*GetMapDir(int i);**

**Remarks:**

Map paths are accessed using a virtual array mechanism. This method returns the 'i-th' map path.

**Parameters:**

**int i**

Specifies the map path to retrieve.

**Return Value:**

The name of the 'i-th' map path.

**Prototype:**

**virtual Interface \*Max();**

**Remarks:**

Implemented by the System.

Returns an interface pointer for calling methods provided by 3ds max. See [Class Interface](#).

## **Creation / Loading**

**Prototype:**

**virtual Bitmap \*NewBitmap();**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called to allocate and return a pointer to a new instance of the Bitmap class. The default constructor is used.

**Prototype:**

**virtual Bitmap \*Create(BitmapInfo \*bi);**

**Remarks:**

This method creates a new bitmap using the properties of the BitmapInfo passed. For more details on creating bitmaps, see the section [Working with Bitmaps](#). Make sure you delete the Bitmap created when you are done using it.

**Parameters:**

**BitmapInfo \*bi**

A pointer to an instance of the class BitmapInfo describing the bitmap to create.

**Return Value:**

A pointer to a newly created instance of class Bitmap.

**Prototype:**

**virtual Bitmap \*Create(PBITMAPINFO pbmi);**

**Remarks:**

This method is used for creating a new bitmap from an existing Windows Device Independent Bitmap. Make sure you delete the Bitmap created when you are done using it.

**Parameters:**

**PBITMAPINFO pbmi**

An existing Windows DIB. For more details on creating bitmaps, see the section [Working with Bitmaps](#).

**Return Value:**

Pointer to a new instance of class Bitmap created from the DIB.

See Also: To create a Windows DIB from a Bitmap see **Bitmap::ToDib()**

**Prototype:**

**virtual Bitmap \*Load(BitmapInfo \*bi, BMMRES \*status = NULL)**

**Remarks:**

This method loads a bitmap using the parameters specified by the **BitmapInfo** pointer. Make sure you delete the Bitmap created when you are

done using it.

Note: When several plug-ins call this method to load the same image, they all receive the same pointer to one instance of the **BitmapStorage**. So if one plug-in manipulates the image, the changes will get reflected everywhere. A developer may use **BitmapManager::Create()** followed by

**Bitmap::CopyImage()** to create a unique instance of **BitmapStorage**.

Also Note: One of the methods in **BitmapInfo** returns a window handle to send progress report messages. If you want to receive these messages (for purposes of putting up a progress bar during the load), set the window handle (**bi->SetUpdateWindow(hWnd)**) and process **BMM\_PROGRESS** and **BMM\_CHECKABORT** messages.

**Parameters:**

**BitmapInfo \*bi**

Specifies the properties of the bitmap to load.

**BMMRES \*status**

The result of the bitmap load operation. See [Bitmap Error Codes](#).

**Return Value:**

A pointer to a new instance of the class Bitmap.

**Prototype:**

```
virtual BMMRES LoadInto(BitmapInfo *bi, Bitmap **map,  
    BOOL forceReload=FALSE);
```

**Remarks:**

This method loads the bitmap specified by **bi** into the bitmap pointed to by **map**. The normal **Load()** method creates a new bitmap. However, if you already have an existing bitmap and simply want to load in a new frame, this method may be used. Specify which bitmap to use using **bi** and the map to load into using **map**. For instance, if you have an AVI file and you want to load a new frame, you can simply update the frame number specified in the **BitmapInfo** and call this method passing the bitmap associates with the previous frame.

**Parameters:**

**BitmapInfo \*bi**

Specifies the properties of the bitmap to load.

**Bitmap \*\*map**

A pointer to a pointer to a bitmap. This is the bitmap that will be loaded into.

**BOOL forceReload=FALSE**

If an existing bitmap that matches **bi** is already loaded, then calling calling **LoadInto()** won't load from the disk or device. Rather it will just use the existing in memory version. If you want to force the bitmap to be reloaded from the file or device set this to TRUE.

**Return Value:**

The result of the bitmap load operation. See [Bitmap Error Codes](#).

## User Interface Methods

**Prototype:**

**virtual BMMRES GetImageInfoDlg(HWND hWnd, BitmapInfo \*bi, const TCHAR \*filename = NULL);**

**Remarks:**

This method will display information about the given bitmap in a dialog. The source of the information is either defined in **bi->Name()/bi->Device()** or explicitly in the **filename** passed). This method is an interface into **BitmapIO::GetImageInfoDlg()**. It is not normally called by developers. The default implementation is within the Bitmap Manager. There is a generic Image Info dialog that is used unless the proper BitmapIO class implements it own dialog (and notifies the system through the **BitmapIO::Capabilities()** method).

**Parameters:**

**HWND hWnd**

The parent window handle calling the dialog.

**BitmapInfo \*bi**

Defines the name of the bitmap or device (unless specified below). The image information fields of BitmapInfo \*bi are set with the information loaded from the image.

**const TCHAR \*filename = NULL**

Specifies the filename to use explicitly.

**Return Value:**

The result of the operation. See [Bitmap Error Codes](#).

**Prototype:**

```
virtual BMMRES GetImageInfo(BitmapInfo *bi, const TCHAR  
*filename = NULL);
```

**Remarks:**

This method is used to get information about an image, ie things like image resolution (**bi->Width()/bi->Height()**), number of frames, etc. This is an interface into **BitmapIO::GetImageInfo()**. Given an image definition in **bi.Name()** / **bi.Device()** or explicitly in filename (this function will place filename, if not NULL, into **bi.Name()** before calling **BitmapIO::GetImageInfo()**), the proper device will fill the data members in **BitmapInfo \*bi** with information about the image.

**Parameters:**

**BitmapInfo \*bi**

Defines the name of the bitmap or device (unless specified below).

**const TCHAR \*filename = NULL**

Specifies the filename to use explicitly.

**Return Value:**

The result of the operation. See [Bitmap Error Codes](#).

**Prototype:**

```
virtual BOOL ImageInputOptions( BitmapInfo *bi, HWND hWnd  
);
```

**Remarks:**

This method brings up the standard 3ds max Image Input Options dialog box. If the users selects OK from the dialog, the appropriate data members of **BitmapInfo \*bi** are filled specifying the user's choices. These are the 'Custom' fields accessed using methods such as **GetCustomX()**, **GetCustomGamma()**, **GetCustomStep()**, etc.

**Parameters:**

**BitmapInfo \*bi**

The instance of BitmapInfo that is updated based on the users dialog selections.

**HWND hWnd**

The parent window handle for the dialog.

**Return Value:**

Returns TRUE if the users selects OK from the dialog; otherwise FALSE.

**Prototype:**

**virtual BOOL SelectDeviceInput(BitmapInfo \*bi, HWND hWnd);**

**Remarks:**

Brings up the standard 3ds max Select Image Input Device dialog box. If the users selects OK from the dialog, then **bi->Device()** is set to the name of the users device choice.

**Parameters:**

**BitmapInfo \*bi**

Points to the instance of BitmapInfo that is updated based on the users dialog selections.

**HWND hWnd**

The parent window handle for the dialog.

**Return Value:**

TRUE if the user exited the dialog using OK; otherwise FALSE.

**Prototype:**

**virtual BOOL SelectDeviceOutput(BitmapInfo \*bi, HWND hWnd);**

**Remarks:**

Brings up the standard 3ds max Select Image Output Device dialog box. If the users selects OK from the dialog, then **bi->Device()** is set to the name of the users device choice.

**Parameters:**

**BitmapInfo \*bi**

Points to the instance of BitmapInfo that is updated based on the users dialog selections.

**HWND hWnd**

The parent window handle for the dialog.

**Return Value:**

TRUE if the user exited the dialog using OK; otherwise FALSE.

**Prototype:**

```
virtual BOOL SelectFileOutput(BitmapInfo *bi, HWND hWnd,  
TCHAR *title = NULL, ULONG *pflags = NULL);
```

**Remarks:**

Brings up the standard 3ds max Browse Images for Output dialog box. If the users selects OK from the dialog, then **bi->Name()** is set to the name of the users file choice.

**Parameters:****BitmapInfo \*bi**

Points to the instance of BitmapInfo that is updated based on the users dialog selections.

**HWND hWnd**

The parent window handle for the dialog.

**TCHAR \*title = NULL**

The optional title string to display in the title bar of the dialog.

**ULONG \*pflags = NULL**

This parameter is available in release 4.0 and later only.

One of the following:

**BMM\_ENABLE\_SAVE\_REGION**

This flag will cause the "SaveRegion" check box to appear in the dialog.

**BMM\_DO\_SAVE\_REGION**

This flag will return the state of the check box.

**Return Value:**

TRUE if the user exited the dialog using OK; otherwise FALSE.

```
virtual BOOL SelectFileOutput ( BitmapInfo *bi, HWND hWnd, TCHAR
```

\*title = NULL, ULONG \*pflags = NULL)

**Prototype:**

**virtual BOOL SelectFileInput(BitmapInfo \*bi, HWND hWnd, TCHAR \*title = NULL);**

**Remarks:**

Brings up the standard 3ds max Browse Images for Input dialog box. If the users selects OK from the dialog, then **bi->Name()** is set to the name of the users file choice.

**Parameters:**

**BitmapInfo \*bi**

The instance of BitmapInfo that is updated based on the users dialog selections.

**HWND hWnd**

The parent window handle for the dialog.

**TCHAR \*title = NULL**

The optional title string to display in the title bar of the dialog.

**Return Value:**

TRUE if the user exited the dialog using OK; otherwise FALSE.

**Prototype:**

**virtual BOOL SelectFileInputEx(BitmapInfo \*bi, HWND hWnd, TCHAR \*title = NULL, BOOL viewonly = FALSE);**

**Remarks:**

This method brings up the standard 3ds max Browse Images for Input dialog box (the same as **SelectFileInput()**) but a "Devices" button is present so the user can select both image files and image devices.

**Parameters:**

**BitmapInfo \*bi**

The instance of BitmapInfo that is updated based on the users dialog selections.

**HWND hWnd**

The parent window handle for the dialog.

**TCHAR \*title = NULL**

The optional title string to display in the title bar of the dialog.

**BOOL viewonly = FALSE**

If **viewonly** is set to TRUE, the View button is hidden in the dialog.

**Return Value:**

TRUE if the user exited the dialog using OK; otherwise FALSE.

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:**

**int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

**Prototype:**

**virtual void BeginSavingLoadErrorFiles()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used for accumulating the names of bitmap files that didn't load. Instead of having the BitmapManager display the missing file dialog, it now just collects the names (which can be retrieved using **GetLoadErrorFileList()** below).

**Prototype:**

**virtual NameTab &GetLoadErrorFileList()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method will return a list of names of bitmap files that were not found as discussed in the method above. See [Class NameTab](#).

**Prototype:**

**virtual void EndSavingLoadErrorFiles()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method ends the accumulation of a list of bitmap files that didn't load, and frees the list. See the two methods above.

**Prototype:**

**virtual bool CanImport(const TCHAR\* filename)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns true if the extension of the specified file name is one of the supported types (i.e. there is a BitmapIO module for it); otherwise false.

**Parameters:**

**const TCHAR\* filename**

The file name to check.

## Class IPoint2

See Also: [Class Point2](#).

class IPoint2

### Description:

This class describes a 2D point using int x and y coordinates. Methods are provided to add and subtract points, multiply and divide by scalars, normalize and compute the dot product of two IPoint2s. All methods are implemented by the system.

### Data Members:

public:

**int x,y;**

### Methods:

### Prototype:

**IPoint2()**

### Remarks:

Constructor.

### Prototype:

**IPoint2(int X, int Y)**

### Remarks:

Constructor. Data members are initialized to X and Y.

### Prototype:

**IPoint2(const IPoint2& a)**

### Remarks:

Constructor. Data members are initialized to a.x and a.y.

### Prototype:

**IPoint2(int af[2])**

### Remarks:

Constructor. Data members are initialized as  $x = af[0]$  and  $y = af[1]$ .

**Prototype:**

**int DotProd(const IPoint2&) const;**

**Remarks:**

Returns the dot product of two IPoint2's. This is the sum of both x values multiplied together and both y values multiplied together.

**Operators:**

**Prototype:**

**int& operator[](int i)  
const int& operator[](int i) const**

**Remarks:**

Allows access to x, y using the subscript operator.

**Return Value:**

An index of 0 will return x, 1 will return y.

**Prototype:**

**operator int\*()**

**Remarks:**

Returns the address of the IPoint2.x.

**Prototype:**

**IPoint2 operator-() const**

**Remarks:**

Unary -. Negates both x and y.

**Prototype:**

**IPoint2 operator+() const**

**Remarks:**

Unary +. Returns the Ipoint2 unaltered.

**Prototype:**

**IPoint2& operator-=(const IPoint2&);**

**Remarks:**

Subtracts a IPoint2 from this IPoint2.

**Prototype:**

**IPoint2& operator+=(const IPoint2&);**

**Remarks:**

Adds a IPoint2 to this IPoint2.

**Prototype:**

**IPoint2& operator\*=(int);**

**Remarks:**

Multiplies this IPoint2 by an integer value.

**Prototype:**

**IPoint2& operator/=(int);**

**Remarks:**

Divides this IPoint2 by an integer value.

**Prototype:**

**IPoint2 operator-(const IPoint2&) const;**

**Remarks:**

Subtracts a IPoint2 from a IPoint2.

**Prototype:**

**IPoint2 operator+(const IPoint2&) const;**

**Remarks:**

Adds a IPoint2 to a IPoint2.

**Prototype:**

**int operator\*(const IPoint2&) const;**

**Remarks:**

Returns the dot product of two IPoint2's. This is the sum of both x values multiplied together and both y values multiplied together.

**Prototype:**

**int operator==(const IPoint2& p) const**

**Remarks:**

Equality operator. Compare two IPoint2's.

**Return Value:**

Nonzero if the IPoint2's are equal; otherwise 0.

The following functions are not members of class IPoint2 but are available for use:

**Function:**

**int Length(const IPoint2& v);**

**Remarks:**

Returns the length of the IPoint2, ie:

**sqrt(v.x\*v.x+v.y\*v.y);**

**Function:**

**IPoint2 Normalize(const IPoint2&);**

**Remarks:**

Returns a unit vector. This is an IPoint2 with each component divided by the point **Length()**.

**Prototype:**

**IPoint2 operator\*(int, const IPoint2&);**

**IPoint2 operator\*(const IPoint2&, int);**

**Remarks:**

Each returns an IPoint2 multiplied by a scalar.

**Prototype:**

**IPoint2 operator/(const IPoint2&, int);**

**Remarks:**

Returns an IPoint2 whose x and y members are divided by a scalar.

**Prototype:**

**ostream &operator<<(ostream&, const IPoint2&);**

**Remarks:**

Formats the IPoint2 for output as in

**(x, y).**

**Prototype:**

**inline int MaxComponent(const IPoint2& p)**

**Remarks:**

Returns the component with the minimum abs value. 0=x, 1=y.

**Prototype:**

**inline int MinComponent(const IPoint2& p)**

**Remarks:**

Returns the component with the minimum abs value. 0=x, 1=y.

## Class IPoint3

See Also: [Class Point3](#).

class IPoint3

### Description:

This class describes a 3D point using integer x, y and z coordinates. Methods are provided to add and subtract points, multiply and divide by scalars, and element by element multiply and divide two points. All methods are implemented by the system.

### Methods:

### Data Members:

public:

**int x,y,z;**

### Prototype:

**IPoint3()**

### Remarks:

Constructor. No initialization is performed.

### Prototype:

**IPoint3(int X, int Y, int Z)**

### Remarks:

Constructor. x, y, and z are initialized to the values specified.

### Prototype:

**IPoint3(const IPoint3& a)**

### Remarks:

Constructor. x, y, and z are initialized to the IPoint3 specified.

### Prototype:

**IPoint3(int ai[3])**

**Remarks:**

Constructor. x, y, and z are initialized to. ai[0], ai[1], and ai[2] respectively.

**Prototype:**

**int DotProd(const IPoint3&) const;**

**Remarks:**

Returns the dot product of two IPoint3s.

**Prototype:**

**IPoint3 CrossProd(const IPoint3&) const;**

**Remarks:**

Returns the cross product of two IPoint3's (vectors).

**Operators:****Prototype:**

**int& operator[](int i)  
const int& operator[](int i) const**

**Remarks:**

Allows access to x, y and z using the [ ] operator.

**Return Value:**

An index of 0 will return x, 1 will return y, 2 will return z.

**Prototype:**

**operator int\*()**

**Remarks:**

Conversion function. Returns the address of the IPoint3.

**Prototype:**

**IPoint3 operator-() const**

**Remarks:**

Unary - operator. Negates x, y and z.

**Prototype:**

**IPoint3 operator+() const**

**Remarks:**

Unary +. Returns the point unaltered.

**Prototype:**

**IPoint3& operator-=(const IPoint3&);**

**Remarks:**

Subtracts a IPoint3 from this IPoint3.

**Prototype:**

**IPoint3& operator+=(const IPoint3&);**

**Remarks:**

Adds a IPoint3 to this IPoint3.

**Prototype:**

**IPoint3 operator-(const IPoint3&) const;**

**Remarks:**

Subtracts a IPoint3 from a IPoint3.

**Prototype:**

**IPoint3 operator+(const IPoint3&) const;**

**Remarks:**

Adds a IPoint3 to a IPoint3.

**Prototype:**

**int operator\*(const IPoint3&) const;**

**Remarks:**

Returns the dot product of two IPoint3s.

**Prototype:**

**IPoint3 operator^(const IPoint3&) const;**

**Remarks:**

The cross product of two IPoint3's (vectors).

**Prototype:**

**int operator==(const IPoint3& p) const**

**Remarks:**

Test for equality between two IPoint3's.

**Return Value:**

Nonzero if the IPoint3's are equal; otherwise 0.

The following functions are not members of IPoint3 but are available for use:

**Prototype:**

**int MaxComponent(const IPoint3&);**

**Remarks:**

Returns the component with the maximum absolute value. 0=x, 1=y, 2=z.

**Prototype:**

**int MinComponent(const IPoint3&);**

**Remarks:**

Returns the component with the minimum absolute value. 0=x, 1=y, 2=z.

**Prototype:**

**inline float Length(const IPoint3& v)**

**Remarks:**

Returns the 'Length' of the point. This is **sqrt(v.x\*v.x+v.y\*v.y+v.z\*v.z)**

**Prototype:**

**ostream &operator<<(ostream&, const IPoint3&);**

**Remarks:**

Formats the IPoint3 for output as in:

**(x, y, z)**

## Class Box2

See Also: [Class IPoint2](#), [Data Types](#).

```
class Box2 : public RECT
```

### Description:

This class describes a 2D rectangular region using integer coordinates. This class is sub-classed from RECT (from the Windows API). Box2 provides methods that return individual coordinates of the box, scale and translate it, retrieve its center, modify its size, expand it to include points or other boxes, and determine if points are inside the box. All methods are implemented by the system.

### Methods:

#### Prototype:

```
Box2();
```

#### Remarks:

Constructs a Box2 object. The box is initialized such that it is 'empty'. See **IsEmpty()** below.

#### Prototype:

```
Box2(const IPoint2 a, const IPoint2 b);
```

#### Remarks:

Constructs a Box2 object from the specified corners.

#### Parameters:

```
const IPoint2 a
```

The upper left corner of the box.

```
const IPoint2 b
```

The lower right corner of the box.

#### Prototype:

```
int IsEmpty();
```

#### Remarks:

Determines whether the box has been 'Set Empty' (see below). When a box is created using the default constructor it is set to 'empty'.

**Return Value:**

TRUE if the box is empty; FALSE otherwise.

**Prototype:**

```
void SetEmpty();
```

**Remarks:**

Sets the box to 'empty'. This indicates the box has not had specific values set by the developer.

**Prototype:**

```
void Rectify();
```

**Remarks:**

Adjusts the coordinates of the box such that **top<bottom** and **left<right**.

**Prototype:**

```
void Scale(float f);
```

**Remarks:**

Scales the coordinates of the box about the center of the box.

**Parameters:**

**float f**

Specifies the scale factor.

**Prototype:**

```
void Translate(IPoint2 t);
```

**Remarks:**

Translate the box by the distance specified.

**Parameters:**

**IPoint2 t**

The distance to translate the box.

**Prototype:**

```
IPoint2 GetCenter()
```

**Remarks:**

Returns the center of the box (the midpoint between the box corners).

**Prototype:**

**int x()**

**Remarks:**

Returns the minimum x coordinate of the box.

**Prototype:**

**int y()**

**Remarks:**

Returns the minimum y coordinate.

**Prototype:**

**int w()**

**Remarks:**

Returns the width of the box.

**Prototype:**

**int h()**

**Remarks:**

Returns the height of the box.

**Prototype:**

**void SetW(int w)**

**Remarks:**

Sets the box width to the width specified. The 'right' coordinate is adjusted such that:

**right = left + w - 1**

**Parameters:**

**int w**

The new width for the box.

**Prototype:**

**void SetH(int h)**

**Remarks:**

Sets the height of the box to the height specified. The 'bottom' coordinate is adjusted such that:

**bottom = top + h - 1;**

**Parameters:**

**int h**

The new height for the box.

**Prototype:**

**void SetX(int x)**

**Remarks:**

Sets the left coordinate of the box to x.

**Parameters:**

**int x**

The new value for the left coordinate.

**Prototype:**

**void SetY(int y)**

**Remarks:**

Set the top coordinate to y.

**Parameters:**

**int y**

The new value for the top coordinate.

**Prototype:**

**void SetWH(int w, int h)**

**Remarks:**

Sets both the width and height of the box.

**Parameters:**

**int w**

The new width for the box.

**int h**

The new height of the box.

**Prototype:**

**void SetXY(int x, int y)**

**Remarks:**

Sets both the left and top coordinates of the box.

**Parameters:**

**int x**

The new left coordinate.

**int y**

The new top coordinate.

**Prototype:**

**int Contains(const IPoint2& p) const**

**Remarks:**

Determines if the point passed is contained within the box. Returns nonzero if the point is inside the box; otherwise 0.

**Operators:**

**Prototype:**

**Box2& operator=(const RECT& r);**

**Box2& operator=(RECT& r);**

**Remarks:**

Assignment operators. Copies the specified source RECT into this Box2 object.

**Prototype:**

**Box2& operator+=(const Box2& b);**

**Remarks:**

Expands this **Box2** to completely include box **b**.

**Prototype:**

**Box2& operator+=(const IPoint2& p);**

**Remarks:**

Expands this **Box2** to include point **p**.

**Prototype:**

**int operator==( const Box2& b ) const**

**Remarks:**

Equality operator. Determines whether b is equal to Box2. Returns nonzero if the boxes are equal; 0 otherwise.

## Class Box3

See Also: [Class Point3](#), [Class Matrix3](#).

class Box3

### Description:

This class represents a 3D box volume described by two 3D corner coordinates. Box3 provides methods that return individual coordinates of the box, scale and translate it, retrieve its center, modify its size, expand it to include points or other boxes, and determine if points are inside the box. All methods are implemented by the system.

### Data Members:

public:

**Point3 pmin, pmax;**

The corners of the 3D box.

### Methods:

### Prototype:

**Box3();**

### Remarks:

Constructor. The corners of the box are initialized such that the box is 'empty'. See **IsEmpty()**.

### Prototype:

**Box3(const Point3& p, const Point3& q)**

### Remarks:

Constructor. The corners of the box are initialized to the points passed.  
**pmin=p; pmax = q.**

### Prototype:

**void Init();**

### Remarks:

Initializes this box such that **pmin** is a very large value while **pmax** is a small

value. Thus the box is 'empty'. See **IsEmpty()**.

**Prototype:**

**int IsEmpty() const;**

**Remarks:**

Determines if the box is empty. This indicates the box has not had specific values set by the developer.

**Return Value:**

Nonzero if the box is empty; otherwise 0.

**Prototype:**

**void MakeCube(const Point3& p, float side);**

**Remarks:**

Modifies this box such that half the side length is subtracted from **pmin** and added to **pmax**. This creates a cube with the specified center **p** and side length **side**.

**Parameters:**

**const Point3& p**

Specifies the center point of the cube.

**float side**

Specifies the side length.

**Prototype:**

**Point3 Min() const**

**Remarks:**

Returns the value of corner **pmin**.

**Prototype:**

**Point3 Max() const**

**Remarks:**

Returns the value of corner **pmax**.

**Prototype:**

**Point3 Center() const**

**Remarks:**

Returns the center of this Box3 as a Point3.

**Prototype:**

**Point3 Width() const**

**Remarks:**

Returns the width of the box as a Point3. This is **pmax-pmin**.

**Prototype:**

**void Scale(float s);**

**Remarks:**

Scales this box about its center by the specified scale.

**Parameters:**

**float s**

Specifies the scale factor for this Box3.

**Prototype:**

**void Translate(const Point3 &p);**

**Remarks:**

Translates this box by the distance specified. The point is added to each corner.

**Parameters:**

**const Point3 &p**

Specifies the distance to translate the box.

**Prototype:**

**void EnlargeBy(float s);**

**Remarks:**

Enlarges this box. A Point3 is created from **s** as Point3(s,s,s) and added to

**pmax** and subtracted from **pmin**. If the box is 'empty', the box is centered at (0,0,0) and then enlarged.

**Prototype:**

**int Contains(const Point3& p) const;**

**Remarks:**

Determines if the specified point **p** is contained in this box.

**Parameters:**

**const Point3& p**

Specifies the point to check.

**Return Value:**

Nonzero if the specified point is contained in this box; otherwise 0.

**Prototype:**

**int Contains(const Box3& b) const;**

**Remarks:**

Determines if the specified Box3 is contained totally within this box.

**Parameters:**

**const Box3& b**

Specifies the box to check.

**Return Value:**

Nonzero if the specified box is entirely contained within this box; otherwise 0.

**Operators:**

**Prototype:**

**Point3 operator[](int i) const;**

**Remarks:**

Operator[] returns the 'i-th' corner point:

**Mapping : X Y Z**

**[0] : (min,min,min)**

**[1] : (max,min,min)**

**[2] : (min,max,min)**

[3] : (max,max,min)

[4] : (min,min,max)

[5] : (max,min,max)

[6] : (min,max,max)

[7] : (max,max,max)

**Parameters:**

**int i**

Specifies the corner to retrieve ( $0 \leq i \leq 7$ )

**Return Value:**

The 'i-th' corner point as a Point3.

**Prototype:**

**Box3& operator+=(const Point3& p);**

**Remarks:**

Expands this Box3 to include the Point3 **p**.

**Parameters:**

**const Point3& p**

Specifies the point to expand the box to include.

**Prototype:**

**Box3& operator+=(const Box3& b);**

**Remarks:**

Expands this Box3 to include the Box3 **b**.

**Parameters:**

**const Box3& b**

Specifies the Box3 to expand this box to include.

**Prototype:**

**Box3 operator\*(const Matrix3& tm) const;**

**Remarks:**

Returns a box that bounds the 8 transformed corners of the input box.

**Parameters:**

**const Matrix3& tm**

Specifies the matrix to transform the box corners by.

## Class Matrix2

See Also: [Class Point2](#), [Class Point3](#), [Class Matrix3](#).

class Matrix2

### Description:

This class defines a 3x2 2D transformation matrix. Methods are provided to zero the matrix, set it to the identity matrix, translate, rotate and scale it, and compute its inverse. Operators are available for matrix addition, subtraction and multiplication. All methods of this class are implemented by the system.

### Data Members:

public:

**float m[3][2];**

### Methods:

#### Prototype:

**Matrix2()**

#### Remarks:

Constructor. No initialization is done in this constructor. Use **Zero()** or **IdentityMatrix()** to initialize the matrix.

#### Prototype:

**Matrix2(float (\*fp)[2]);**

#### Remarks:

Constructor. The matrix is initialized using **fp**.

### Operators:

#### Prototype:

**Matrix2& operator-=( const Matrix2& M);**

#### Remarks:

Subtracts a Matrix2 from this Matrix2.

#### Prototype:

**Matrix2& operator+=( const Matrix2& M);**

**Remarks:**

Adds a Matrix2 to this Matrix2.

**Prototype:**

**Matrix2& operator\*=( const Matrix2& M);**

**Remarks:**

Matrix multiplication between this Matrix2 and **M**.

**Prototype:**

**operator float\*()**

**Remarks:**

Returns the address of the Matrix2.

**Prototype:**

**void IdentityMatrix();**

**Remarks:**

Sets this Matrix2 to the Identity Matrix.

**Prototype:**

**void Zero();**

**Remarks:**

Set all elements of this Matrix2 to 0.0f

**Prototype:**

**Point2 GetRow(int i) const;**

**Remarks:**

Returns the specified row of this matrix.

**Parameters:**

**int i**

Specifies the row to retrieve (0-2).

**Prototype:**

**void SetRow(int i, Point2 p);**

**Remarks:**

Sets the specified row of this matrix.

**Parameters:**

**int i**

Specifies the row to set (0-2).

**Point2 p**

The values to set.

**Prototype:**

**Point3 GetColumn(int i);**

**Remarks:**

Returns the specified column of this matrix.

**Parameters:**

**int i**

Specifies the column to retrieve (0 or 1).

**Prototype:**

**void SetColumn(int i, Point3 col);**

**Remarks:**

Sets the specified column of this matrix.

**Parameters:**

**int i**

Specifies the column to set (0 or 1).

**Point3 col**

The values to set.

**Prototype:**

**Point2 GetColumn2(int i);**

**Remarks:**

This method returns a Point2 containing the upper two rows of the specified column.

**Parameters:**

**int i**

Specifies the column to get (0 or 1).

**Prototype:**

**void SetTrans(const Point2 p);**

**Remarks:**

Sets the translation row of the matrix to the specified values.

**Parameters:**

**const Point2 p**

The values to set.

**Prototype:**

**void SetTrans(int i, float v);**

**Remarks:**

Sets the specified element of the translation row of this matrix to the specified value.

**Parameters:**

**int i**

Specifies which column to set (0 or 1)

**float v**

The value to store.

**Prototype:**

**Point2 GetTrans();**

**Remarks:**

Returns the translation row of this matrix.

**Prototype:**

**void SetTranslate(const Point2& s);**

**Remarks:**

This method is available in release 3.0 and later only.

Initializes the matrix to the identity then sets the translation row to the specified values.

**Parameters:**

**const Point2& s**

The values to store.

**Prototype:**

**void SetRotate(float angle);**

**Remarks:**

This method is available in release 3.0 and later only.

Initializes the matrix to the identity then sets the rotation to the specified value.

**Parameters:**

**float angle**

The rotation angle in radians.

**Prototype:**

**void Invert();**

**Remarks:**

This method is available in release 3.0 and later only.

This matrix may be used to invert the matrix in place.

**Prototype:**

**void Translate(const Point2& p);**

**Remarks:**

Apply an incremental translation to this matrix.

**Parameters:**

**const Point2& p**

Specifies the amount to translate the matrix.

**Prototype:**

**void Rotate(float angle);**

**Remarks:**

Apply an incremental rotation to this matrix using the specified angle.

**Parameters:**

**float angle**

Specifies the angle of rotation.

**Prototype:**

**void Scale(const Point2& s, BOOL trans);**

**Remarks:**

Apply an incremental scaling to this matrix using the specified scale factors.

**Parameters:**

**const Point2& s**

The scale factors.

**BOOL trans = FALSE**

If set to **TRUE**, the translation component is scaled. If **trans = FALSE** the translation component is unaffected. When 3ds max was originally written there was a bug in the code for this method where the translation portion of the matrix was not being scaled. This meant that when a matrix was scaled the bottom row was not scaled. Thus it would always scale about the local origin of the object, but it would scale the world axes. When this bug was discovered, dependencies existed in the code upon this bug. Thus it could not simply be fixed because it would break the existing code that depended upon it working the incorrect way. To correct this the **trans** parameter was added. If this is set to **TRUE**, the translation component will be scaled correctly. The existing plug-ins don't use this parameter, it defaults to **FALSE**, and the code behaves the old way.

**Prototype:**

**Matrix2 operator\*(const Matrix2& B) const;**

**Remarks:**

Perform matrix multiplication.

**Prototype:**

**Matrix2 operator+(const Matrix2& B) const;**

**Remarks:**

Perform matrix addition.

**Prototype:**

**Matrix2 operator-(const Matrix2& B) const;**

**Remarks:**

Perform matrix subtraction.

The following functions are not members of Matrix2 but are available for use:

**Function:**

**Matrix2 RotateMatrix(float angle);**

**Remarks:**

Builds an identity matrix and sets the rotation components based on the specified angle.

**Parameters:**

**float angle**

Specifies the angle of rotation.

**Return Value:**

A new Matrix2 object with the specified rotation angle.

**Function**

**Matrix2 TransMatrix(const Point2& p);**

**Remarks:**

Builds an identity matrix and sets the specified translation components.

**Parameters:**

**const Point2& p**

Specifies the translation.

**Return Value:**

A new Matrix2 object with the specified translation.

**Function:**

**Matrix2 ScaleMatrix(const Point2& s);**

**Remarks:**

Builds an identity matrix and sets the specified scaling components.

**Parameters:**

**const Point2& s**

Specifies the scale factors.

**Return Value:**

A new Matrix2 object with the specified scale.

**Function:**

**Matrix2 Inverse(const Matrix2& M);**

**Remarks:**

Returns the inverse of the specified Matrix2.

**Parameters:**

**const Matrix2& M**

Specifies the matrix to return the inverse of.

**Return Value:**

The inverse of the specified Matrix2.

**Function:**

**Point2 operator\*(const Matrix2& A, const Point2& V);**

**Point2 operator\*(const Point2& V, const Matrix2& A);**

**Remarks:**

Transforms the specified Point2 with the specified Matrix2.

**Parameters:**

**const Matrix2& A**

The matrix to transform the point with.

**const Point2& V**

The point to transform.

**Return Value:**

The transformed Point2.

**Function:**

**Point2 VectorTransform(const Matrix2& M, const Point2& V);**

**Remarks:**

This method transforms a 2D point by a 2x3 matrix. This is analogous to the 3D case.

**Parameters:**

**const Matrix2& M**

The matrix to transform the point with.

**const Point2& V**

The point to transform.

**Return Value:**

The transformed Point2.

**Function:**

**ostream &operator<< (ostream& s, const Matrix2& A);**

**Remarks:**

Formats the matrix for output.

## Structure AffineParts

See Also: [Class Matrix3](#), [Class Quat](#), [Class Point](#), [Class ScaleValue](#).

### Description:

This structure and the associated functions provide a way to decompose an arbitrary Matrix3 into its translation, rotation, and scale components.

To use these APIs put the following statement in your source file:

```
#include "decomp.h"
```

For a full discussion of this decomposition see Graphics Gems IV - Polar Matrix Decomposition by Ken Shoemake. ISBN 0-12-336155-9.

**T F R U K U'**

**T** - translation matrix

**F** - either an identity matrix or negative identity matrix

**R** - rotation defined by Quat q.

**U** - rotates you into the coordinates system where the scaling or stretching is done

**K** - scaling matrix

**U'** - inverse of u.

### Structure:

```
typedef struct {
```

```
    Point3 t;
```

The translation components.

```
    Quat q;
```

The essential rotation.

```
    Quat u;
```

The stretch rotation. This is the axis system of the scaling application.

```
    Point3 k;
```

The stretch factors. These are the scale factors for x, y and z.

```
    float f;
```

Sign of the determinant.

```
} AffineParts;
```

### Functions:

**Prototype:**

```
void decomp_affine(Matrix3 A, AffineParts *parts);
```

**Remarks:**

This will decompose a matrix into the translation, rotation and scale components and store the results in the **AffineParts** structure passed. This will return correct results for off axis scale. This is a fairly computationally intensive iterative solution operation.

**Parameters:****Matrix3 A**

The input matrix to decompose.

**AffineParts \*parts**

The result. See above.

**Sample Code:**

Note: If you want to rebuild a **Matrix3** from the decomposed parts you get back from **decomp\_affine()** the important thing is the order the parts are combined.

Consider the following matrices constructed from the various affine parts:

**ptm** = position component (t)

**rtm** = "essential" rotation (q)

**srtm** = "stretch" rotation (u)

**stm** = scale component (k)

**ftm** = the flip tm -> ScaleMatrix(Point3(ap.f,ap.f,ap.f));

Here's the correct way of reassembling the decomposed matrix:

```
Matrix3 srtm, rtm, ptm, stm, ftm;
```

```
ptm.IdentityMatrix();
```

```
ptm.SetTrans(ap.t);
```

```
ap.q.MakeMatrix(rtm);
```

```
ap.u.MakeMatrix(srtm);
```

```
stm = ScaleMatrix(ap.k);
```

```
mat = Inverse(srtm) * stm * srtm * rtm * ftm * ptm;
```

**Prototype:**

**void SpectralDecomp(Matrix3 m, Point3 &s, Quat& q);**

**Remarks:**

This is another way to decompose a matrix into the scale and rotation parts (the position part can be retrieved from the bottom row of the matrix). This does not return correct results for off axis scale.

**Parameters:**

**Matrix3 m**

The input matrix to decompose.

**Point3 &s**

The scale from the matrix.

**Quat& q**

The rotation of the matrix.

**Prototype:**

**void invert\_affine(AffineParts \*parts, AffineParts \*inverse);**

**Remarks:**

This is used to take the **AffineParts** and inverts them. This gives you the equivalent parts for the inverse matrix.

**Parameters:**

**AffineParts \*parts**

The input AffineParts pointer.

**AffineParts \*inverse**

The inverse of **parts**.

## Class AngAxis

See Also: [Class Quat](#), [Class Point3](#).

class AngAxis

### Description:

This class provides a representation for orientation in three space using an angle and axis. This class is similar to a quaternion, except that a normalized quaternion only represents  $-\pi$  to  $+\pi$  rotation. This class will have the number of revolutions stored. All methods of this class are implemented by the system.

The rotation convention in the 3ds max API is the left-hand-rule. Note that this is different from the right-hand-rule used in the 3ds max user interface.

### Data Members:

public:

#### **Point3 axis;**

The axis of rotation.

#### **float angle;**

The angle of rotation about the axis in radians. This angle is left handed.

### Methods:

#### Prototype:

**AngAxis()**

#### Remarks:

Constructor. No initialization is performed.

#### Prototype:

**AngAxis(const Point3& axis, float angle)**

#### Remarks:

Constructor. Data members are initialized to the specified values.

#### Prototype:

**AngAxis(float x, float y, float z, float ang);**

#### Remarks:

This method is available in release 3.0 and later only.

Constructor. The AngAxis is initialized from the specified values.

**Parameters:**

**float x**

The x component of the axis.

**float y**

The y component of the axis.

**float z**

The z component of the axis.

**float ang**

The angle component in radians.

**Prototype:**

**AngAxis(const Matrix3& m);**

**Remarks:**

This method is available in release 3.0 and later only.

Constructor. The AngAxis is initialized with the rotation from the specified matrix.

**Parameters:**

**const Matrix3& m**

The rotation used to initialize the AngAxis.

**Prototype:**

**AngAxis(const Quat &q);**

**Remarks:**

Constructor. Data members are initialized equal to the specified Quat.

**Prototype:**

**AngAxis& Set(float x, float y, float z, float ang);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the angle and axis to the specified values.

**Parameters:****float x**

Specifies the x component of the axis.

**float y**

Specifies the xycomponent of the axis.

**float z**

Specifies the z component of the axis.

**float ang**

Specifies the angle to set in radians.

**Return Value:**

A reference to this AngAxis.

**Prototype:****AngAxis& Set(const Point3& ax, float ang);****Remarks:**

This method is available in release 3.0 and later only.

Sets the angle and axis to the specified values.

**Parameters:****const Point3& ax**

Specifies the axis to set.

**float ang**

Specifies the angle to set in radians.

**Return Value:**

A reference to this AngAxis.

**Prototype:****AngAxis& Set(const Quat& q);****Remarks:**

This method is available in release 3.0 and later only.

Sets the angle and axis based on the rotations from the specified quaternion.

**Parameters:****const Quat& q**

Specifies the angle and axis to use.

**Return Value:**

A reference to this AngAxis.

**Prototype:**

**AngAxis& Set(const Matrix3& m);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the angle and axis based on the rotations from the specified matrix.

**Parameters:**

**const Matrix3& m**

Specifies the angle and axis to use.

**Return Value:**

A reference to this AngAxis.

**Prototype:**

**int GetNumRevs();**

**Remarks:**

Returns the number of revolutions represented by the angle. This returns

**int(angle/TWOPI);**

**Prototype:**

**void SetNumRevs(int num);**

**Remarks:**

Sets the number of revolution to **num**. This modifies angle: **angle +=**

**float(num)\*TWOPI;**

# Class ScaleValue

See Also: [Class Point3](#), [Class Quat](#).

class ScaleValue

## Description:

A ScaleValue describes an arbitrary non-uniform scaling in an arbitrary axis system. The Point3 **s** gives the scaling along the x, y, and z axes, and the quaternion **q** defines the axis system in which scaling is to be applied. All methods are implemented by the system.

## Data Members:

public:

**Point3 s;**

Scale components.

**Quat q;**

The axis system of application.

## Methods:

### Prototype:

**ScaleValue()**

### Remarks:

Constructor. No initialization is performed.

### Prototype:

**ScaleValue(const Point3& as)**

### Remarks:

Constructor. The scale data member is initialized to **as**. The quaternion data member is set to the identity.

### Prototype:

**ScaleValue(const Point3& as, const Quat& aq)**

### Remarks:

Constructor. The scale data member is initialized to **as**. The quaternion data

member is set to **aq**.

**Operators:**

**Prototype:**

**ScaleValue& operator+=(const ScaleValue& s)**

**Remarks:**

Adds a ScaleValue to this ScaleValue.

**Prototype:**

**ScaleValue& operator\*=(const float s)**

**Remarks:**

Multiplies this ScaleValue by a float. This updates the scale components.

**Prototype:**

**float& operator[](int el)**

**Remarks:**

Array access operator. This allows the scale components to be accessed using the array operator.

**Parameters:**

**int el**

Specifies the element to access: 0=x, 1=y, 2=z.

**The following operators are not part of the ScaleValue class but are available for use:**

**Prototype:**

**ScaleValue operator+(const ScaleValue& s0, const ScaleValue& s1);**

**Remarks:**

Returns the sum of two ScaleValues. This still multiplies since scale values are multiplicative not additive.

**Prototype:**

**ScaleValue operator-(const ScaleValue& s0, const ScaleValue& s1);**

**Remarks:**

Returns the difference of two ScaleValues.

**Prototype:**

**ScaleValue operator\*(const ScaleValue& s, float f);**

**Remarks:**

Multiplication of a ScaleValue and a float.

**Prototype:**

**ScaleValue operator\*(float f, const ScaleValue& s);**

**Remarks:**

Multiplication of a ScaleValue and a float.

**Prototype:**

**ScaleValue operator+(const ScaleValue& s, float f);**

**Remarks:**

Returns the sum of a ScaleValue and a float. This adds f to s.x, s.y, and s.z.

**Prototype:**

**ScaleValue operator+(float f, const ScaleValue& s);**

**Remarks:**

Returns the sum of a ScaleValue and a float.

## Class GraphicsWindow

See Also: [Interactive Renderer: Graphics Window](#), [Class GWinSetup](#), [Class HitRegion](#), [List of Marker Types](#), [List of Rendering Limits](#), [Class Point3](#), [Class IPoint3](#), [Class Matrix3](#), [Class Interface](#), [Class ViewExp](#).

```
class GraphicsWindow : public InterfaceServer
```

### Description:

The abstract graphics window class. The methods here provide low-level access to 3ds max's graphics system. These methods are available for plug-ins to do any graphics work not possible using the standard high-level graphics methods of 3ds max.

These methods are for use in the existing 3ds max viewports. Note that these APIs are not for casual use, as they are not intended to be a high level graphics library. For example, many steps are required to display a single lit polygon. These APIs are optimized for speed, and not at all for plug-in programmer ease of use.

These methods are provided, however, so that developers can do things that are otherwise impossible using the high-level methods.

**Developers should use these methods with an understanding of exactly what they are doing since it's quite easy to crash 3ds max when inappropriate arguments are supplied. The calls are specifically optimized for exactly the way 3ds max uses them. In no way should the calls in GraphicsWindow be considered an "ordinary" 2D/3D API. (That's what OpenGL, D3D, and HEIDI are for.)**

One final note of warning: most graphics windows methods execute in a separate thread (or in multiple separate threads) that are owned by the graphics window. Thus, due to thread scheduling, when a bad argument or incorrect sequencing of graphics windows calls causes 3ds max to crash, it is not at all easy to figure out where the problem is. In particular, the location of the main 3ds max thread is not relevant.

All the methods of this class are implemented by the system.

### Method Groups:

The hyperlinks below jump to the start of related methods within the class:

[Driver/Configuration/Support Methods](#)  
[Window / Viewport Transformations Methods](#)

[Drawing Setup](#)

[Drawing Methods \(text, polylines, markers\)](#)

[Device Independent Bitmap Access](#)

[Position/Size/Depth/Clipping Methods](#)

[Get/Set Buffers Methods](#)

[Texture Methods](#)

[Material Methods](#)

[Lights and Camera Methods](#)

[Coordinate Transformation Methods](#)

[Hit Testing Methods](#)

[Utility Functions](#)

**Methods:**

**Driver/Configuration/Support Methods**

**Prototype:**

```
virtual void postCreate(int ct, GraphicsWindow **gw)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This is called after all four GraphicsWindows used by 3ds max are created.  
SDK users shouldn't need this call

**Prototype:**

```
virtual void shutdown()=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This is used to tell the driver that it is shutting down.

**Prototype:**

```
virtual int getVersion()=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This returns "0x200" to indicate R2.0

**Prototype:**

**virtual TCHAR \*getDriverString()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This identifies the driver (and includes manufacturer info if available)

**Prototype:**

**virtual void config(HWND hWnd)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This is called to put up the config dialog if the driver supports  
GW\_SPT\_CAN\_CONFIG

**Parameters:**

**HWND hWnd**

The parent window handle for the dialog.

**Prototype:**

**virtual int querySupport(int what) = 0;**

**Remarks:**

Determines if the driver supports the specified feature.

**Parameters:**

**int what**

One of the following values:

**GW\_SPT\_TXT\_CORRECT**

This is used to enable or gray-out the perspective correction right-click viewport menu option.

**GW\_SPT\_GEOM\_ACCEL**

This is used to indicate to 3ds max (and the mesh class in particular) that the driver wants to handle all of the 3D data natively. In this case, meshes are rendered by passing 3D world space data and letting the driver transform, clip, and light the vertices. If this returns FALSE, then the mesh class handles all xforms/clip/lighting calculations (using a lazy evaluation

algorithm) and then calls the hPolygon or hPolyline 2 1/2D calls for the driver to rasterize. (Primitives that are actually clipped are still sent to the polygon/polyline methods.)

Right now, only the OpenGL driver returns TRUE to this query, but other drivers have been developed that return TRUE, and the HEIDI and D3D drivers may change in the future.

### **GW\_SPT\_TRI\_STRIP**

If this returns TRUE, then 3ds max will try to stripify meshes before calling the rendering methods. Right now, the drivers just return the user preference that is set in the driver config dialog. It defaults to TRUE.

### **GW\_SPT\_DUAL\_PLANES**

If a driver has dual-planes support it returns TRUE. Our OpenGL display driver only returns TRUE for this if the underlying display driver has implemented a custom OpenGL extension that allows us to handle this efficiently.

### **GW\_SPT\_SWAP\_MODEL**

This returns TRUE if 3ds max has to redraw the whole scene any time the viewports are exposed.

### **GW\_SPT\_INCR\_UPDATE**

This returns TRUE if the driver can update a rectangular subset of the viewport without trashing the image outside that rectangle. This is TRUE for most drivers that blit the viewport region and FALSE for those that do page-flipping in the hardware. For OpenGL, this is TRUE if the display driver implements the Microsoft glSwapRectHintWIN extension.

### **GW\_SPT\_1\_PASS\_DECAL**

This is TRUE if the driver can handle decalling with only one pass. Right now, this is TRUE for OpenGL, but FALSE for HEIDI and D3D. (And as with all of these options, it may change in a given driver anytime in the future.)

### **GW\_SPT\_DRIVER\_CONFIG**

This is TRUE if the driver has a configuration dialog box. This is TRUE for all three of our standard drivers.

### **GW\_SPT\_TEXTURED\_BKG**

This is TRUE if the viewport background is implemented as a textured

rectangle, and FALSE if it is a blitted bitmap.

### **GW\_SPT\_VIRTUAL\_VPTS**

This is TRUE if the driver allows viewports to be made larger than the physical window they are attached to. Right now this is only TRUE for OGL.

### **GW\_SPT\_PAINT\_DOES\_BLIT**

This is TRUE if WM\_PAINT messages result in a blit of the backbuffer (as opposed to a page-flipping swap). This allows 3ds max to do quick damage region repair, and works together with the GW\_SPT\_SWAP\_MODEL flag.

### **GW\_SPT\_WIREFRAME\_STRIP**

This is TRUE if the driver wants 3ds max to send down wireframe models using triangle strips instead of a bundle of 2-pt segments. This is only used by the OGL driver, and it is there as a user-choosable performance-accuracy tradeoff (since the strips are faster and are back-culled, but they display hidden edges as though they are visible).

## **Window / Viewport / Transformations Methods**

### **Prototype:**

**virtual HWND getHWnd(void)=0;**

### **Remarks:**

This method is available in release 2.0 and later only.

This returns the "output" window handle. (Input goes to an invisible window above the viewport. The invisible window is owned by MAX.)

### **Prototype:**

**virtual int isPerspectiveView() = 0;**

### **Remarks:**

Returns TRUE if the view is in perspective projection; otherwise FALSE (orthographic projection).

### **Prototype:**

**virtual void getTextExtents(TCHAR \*text, SIZE \*sp)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the size (in pixels) that the specified text string will occupy

**Parameters:**

**TCHAR \*text**

The string to check.

**SIZE \*sp**

The size is returned here. See [Data Types](#).

**Prototype:**

**virtual void setTransform(const Matrix3 &m) = 0;**

**Remarks:**

Sets the current transformation matrix, and updates the modeling coordinates to normalized projection coordinates matrix. This routine also back-transforms each light and the eye point (so that lighting can be done in modeling coordinates).

This method may be used to set a matrix that transforms the point passed to the drawing methods (like **text()**, **marker()**, **polyline()** or **polygon()**).

Normally these methods expect world coordinates. However if this matrix is set to an objects transformation matrix you can pass objects space coordinates and they will be transformed into world space (and then put into screen space when they are drawn). If however this is set to the identity matrix you would pass world space coordinates. You can set this matrix to the objects matrix using the following code:

```
gw->setTransform(inode->GetObjectTM(t));
```

Note: For world-to-screen space conversions by the methods **text()**, **marker()**, **polyline()**, **polygon()**, etc, a developer must explicitly set this matrix to the identity. This is because the **GraphicsWindow** transform may have a non-identity matrix already in place from a previous operation.

**Parameters:**

**const Matrix3 &m**

The new current transformation matrix.

**Prototype:**

**virtual BOOL getFlipped()=0;**

**Remarks:**

This is used internally. It returns if the determinant of the current transform is positive or negative. If it's positive 0 is returned; if it's negative 1 is returned.

**Prototype:**

**virtual void setVirtualViewportParams(float zoom, float xOffset, float yOffset) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to setup a virtual viewport. Note that this is a no-op unless **GW\_SPT\_VIRTUAL\_VPTS** is TRUE. Plug-in developers should not call this method -- it's for internal use only.

**Prototype:**

**virtual void setUseVirtualViewport(int onOff) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to set a virtual viewport as active. Note that this is a no-op unless **GW\_SPT\_VIRTUAL\_VPTS** is TRUE. Plug-in developers should not call this method -- it's for internal use only.

**Parameters:**

**int onOff**

TRUE to set the virtual viewport active; FALSE to make it inactive.

**Position / Size / Depth / Clipping Methods****Prototype:**

**virtual void setPos(int x, int y, int w, int h) = 0;**

**Remarks:**

Sets the size and position of the GraphicsWindow. The coordinates are all Windows coordinates in the space of the GraphicsWindows' parent window.

(The origin is the upper left.)

**Parameters:**

**int x**

Window x origin.

**int y**

Window y origin.

**int w**

Window width.

**int h**

Window height.

**Prototype:**

**virtual void setDisplayState(int s) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

The specified value may be sent to the driver to indicate the display state of the viewport window controlled by the driver.

**Parameters:**

**int s**

The display state to set. One of the following values:

**GW\_DISPLAY\_MAXIMIZED**

**GW\_DISPLAY\_WINDOWED**

**GW\_DISPLAY\_INVISIBLE**

**Prototype:**

**virtual int getDisplayState()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the current state. One of the following values:

**GW\_DISPLAY\_MAXIMIZED**

**GW\_DISPLAY\_WINDOWED**

**GW\_DISPLAY\_INVISIBLE**

**Prototype:****virtual int getWinSizeX()=0;****Remarks:**

This method is available in release 2.0 and later only.

This method gets the current window size in X.

**Prototype:****virtual int getWinSizeY()=0;****Remarks:**

This method is available in release 2.0 and later only.

This method gets the current window size in Y.

**Prototype:****virtual DWORD getWinDepth(void)=0;****Remarks:**

This method is available in release 2.0 and later only.

This method returns the z-buffer depth (in bits)

**Prototype:****virtual DWORD getHitherCoord()=0;****Remarks:**

This method is available in release 2.0 and later only.

This method returns the largest device Z value.

**Prototype:****virtual DWORD getYonCoord()=0;****Remarks:**

This method is available in release 2.0 and later only.

This method returns the smallest device Z value.

**Buffer access methods:****Prototype:**

**virtual BOOL setBufAccess(int which, int b) = 0;**

**Remarks:**

This method is used internally. Most drivers control two image buffers. One is displayed on the screen, and the other is used to rasterize geometric primitives. When rasterization of a complete frame is done, the off-screen buffer is blitted onto the display screen. This is referred to as dual-plane mode. This method will turn dual-plane mode on or off. This is used internally by the File/Preferences... Viewport page Use Dual Planes toggle.

**Parameters:**

**int which**

Specifies which buffer should use dual-planes.

**BUF\_F\_BUFFER**

The image (Framebuffer) buffer.

**BUF\_Z\_BUFFER**

The Z buffer.

**int b**

Nonzero to enable access (toggle on); 0 to toggle off.

**Return Value:**

TRUE if the graphics window has access to the specified buffer; otherwise FALSE.

**Prototype:**

**virtual BOOL getBufAccess(int which) = 0;**

**Remarks:**

This method is used internally. It returns a boolean value indicating if dual plane mode is on or off for the specified buffer.

**Parameters:**

The buffer whose dual-planes setting will be returned.

**int which**

The buffer whose dual-planes setting will be returned. One of the following values:

**BUF\_F\_BUFFER**

The Framebuffer.

## **BUF\_Z\_BUFFER**

The Z buffer.

### **Return Value:**

TRUE if the dual-plane mode is on; otherwise FALSE.

### **Prototype:**

```
virtual BOOL getBufSize(int which, int *size) = 0;
```

### **Remarks:**

This method is used internally. It retrieves the size of the specified buffer in bytes.

### **Parameters:**

#### **int which**

One of the following values:

#### **int \*size**

The size of the buffer in bytes.

Note the following concerning the HEIDI driver. For HEIDI getBufSize() always returns 10 if dual-planes are on (and 0 otherwise). This is because HEIDI actually never returns the image - it keeps its own copy stored away. Thus the "logical" way to think is that we actually get a copy of the buffer by calling getBuf, and that we give it back by calling setBuf. But in reality (with the HEIDI driver) getBuf and setBuf only tell HEIDI to do some internal buffer manipulation.

### **Return Value:**

TRUE if the size was returned; otherwise FALSE.

### **Prototype:**

```
virtual BOOL getBuf(int which, int size, void *buf) = 0;
```

### **Remarks:**

This method is used internally. It retrieves the specified buffer.

### **Parameters:**

#### **int which**

The buffer to retrieve. One of the following values:

**BUF\_F\_BUFFER** - The image Framebuffer.

**BUF\_Z\_BUFFER** - The Z buffer.

**int size**

The number of bytes to retrieve. This must be at least the size returned from **getBufSize()**.

**void \*buf**

Storage for the buffer data.

**Return Value:**

TRUE if the buffer was retrieved; otherwise FALSE.

**Prototype:**

**virtual BOOL setBuf(int which, int size, void \*buf, RECT \*rp) = 0;**

**Remarks:**

Stores the specified buffer.

**Parameters:**

**int which**

The buffer to store. One of the following values:

**BUF\_F\_BUFFER** - The image Framebuffer.

**BUF\_Z\_BUFFER** - The Z buffer.

**int size**

The number of bytes to store.

**void \*buf**

The buffer data.

**RECT \*rp**

This allows only a subset of the saved image rect to be blitted back to the screen.

**Return Value:**

TRUE if the buffer was stored; otherwise FALSE.

## **DIB Methods**

**Prototype:**

**virtual BOOL getDIB(BITMAPINFO \*bmi, int \*size) = 0;**

**Remarks:**

This method returns the viewport image of this graphics window in a packed DIB format. A packed DIB is the standard BMI header followed immediately by all the data bytes (pixels) that make up the image. This is the standard way in Windows to pass a DIB around. See the sample code below for an example of this call in use. Note how it is called twice: once to get the size, once to get the DIB.

**Parameters:****BITMAPINFO \*bmi**

The BITMAPINFO structure defines the dimensions and color information for a Windows device-independent bitmap (DIB). Note that if this parameter is NULL, then only the size value is returned.

**int \*size**

The size of the image in bytes.

**Return Value:**

TRUE if the image was returned; otherwise FALSE.

**Sample Code:**

The following sample code saves the current 3ds max viewport to a user specified file.

```
void TestGetDIB(IObjParam *ip)  
{  
    BITMAPINFO *bmi = NULL;  
    BITMAPINFOHEADER *bmih;  
    BitmapInfo biFile;  
    Bitmap *map;  
    int size;  
    TheManager->SelectFileOutput(&biFile,  
        ip->GetMAXHwnd(), _T("Testing"));  
    if(!biFile.Name()[0])  
        return;  
    ViewExp *vpt = ip->GetActiveViewport();  
    vpt->getGW()->getDIB(NULL, &size);  
    bmi = (BITMAPINFO *)malloc(size);  
    bmih = (BITMAPINFOHEADER *)bmi;
```

```

vpt->getGW()->getDIB(bmi, &size);
biFile.SetWidth((WORD)bmi->biWidth);
biFile.SetHeight((WORD)bmi->biHeight);
biFile.SetType(BMM_TRUE_32);
map = TheManager->Create(&biFile);
map->OpenOutput(&biFile);
map->FromDib(bmi);
map->Write(&biFile);
map->Close(&biFile);
if(bmi)
    free(bmi);
ip->ReleaseViewport(vpt);
}

```

**Prototype:**

**virtual BOOL setBackgroundDIB(int width, int height, BITMAPINFO \*bmi)=0;**

**Remarks:**

This method is available in release 2.0 and later only.  
This method is used internally to zoom the viewport.

**Prototype:**

**virtual void setBackgroundOffset(int x, int y) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.  
This method is used internally to pan the viewport.

## **Texture Methods**

**Prototype:**

**virtual int getTextureSize(int bkg=FALSE) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the size of the texture bitmap that the driver wants sent in to the **getTextureHandle()** call (if **bkg** is **FALSE**). If **bkg** is **TRUE**, this returns the size of the texture that 3ds max should send to the **setBackgroundDIB()** call. In general, the return value needs to be a power of 2, though that could be driver-specific

**Parameters:**

**int bkg=FALSE**

**TRUE** to get the size for **setBackgroundDIB()**; **FALSE** to get the size for **getTextureHandle()**.

**Prototype:**

**virtual DWORD getTextureHandle(BITMAPINFO \*bmi) = 0;**

**Remarks:**

This method returns a handle for the specified texture bitmap. This handle is then used with the **setTextureByHandle()** method (there is only one current texture active at any time). The texture dimensions must be a power of 2.

When a material is on an object, and the material has a texture map, and when the button used to display the texture map is pressed, 3ds max calls this method to get the texture handle. This basically loads the texture into the hardware RAM (if available). When this mapped object gets displayed the method **setTextureHandle()** is called. When the material is deleted, or the display in viewport button is turned off, **freeTextureHandle()** is called.

**Parameters:**

**BITMAPINFO \*bmi**

The DIB image to use as a texture.

**Return Value:**

The texture handle.

**Prototype:**

**virtual BOOL setTextureByHandle(DWORD handle) = 0;**

**Remarks:**

This sets the current texture to the image whose handle is passed (see **getTextureHandle()**). The texture dimensions must be a power of 2.

**Parameters:**

**DWORD handle**

The handle of the texture to make current.

**Prototype:**

**virtual void freeTextureHandle(DWORD handle) = 0;**

**Remarks:**

When you are finished with the texture handle, call this method to free it.

**Parameters:**

**DWORD handle**

The texture handle to free.

**Return Value:**

TRUE if the texture was set; otherwise FALSE.

**Prototype:**

**virtual BOOL setTextureTiling(int u, int v, int w=GW\_TEX\_NO\_TILING) = 0;**

**Remarks:**

Sets the way in which textures are tiled across the surface of the object.

**Parameters:**

The following parameters may use one of these values:

**GW\_TEX\_NO\_TILING**

The texture clamped - Any UVW that is bigger than 1 is interpreted as being 1.

**GW\_TEX\_REPEAT**

As the UVW numbers keep getting larger than 1 the image is repeated.

**GW\_TEX\_MIRROR**

If UVW goes beyond 1, the numbers are interpreted as going backwards. So if you had 0 to 2 it would actually go 0 to 1 then 1 down to 0.

**int u**

The type of texturing in the U direction.

**int v**

The type of texturing in the V direction.

**int w=GW\_TEX\_NO\_TILING**

The type of texturing in the W direction.

**Return Value:**

TRUE if the tiling mode was set; otherwise FALSE.

**Prototype:**

**virtual int getTextureTiling(int which) = 0;**

**Remarks:**

Returns the type of texture tiling set for the particular direction.

For example, if **setTextureTiling(GW\_TEX\_NO\_TILING, GW\_TEX\_REPEAT, GW\_TEX\_MIRROR)** were called first, then

**getTextureTiling(0)** would yield **GW\_TEX\_NO\_TILING**, and **getTextureTiling(1)** would yield **GW\_TEX\_REPEAT**.

**Parameters:**

**int which**

This value is 0 or 1 and it represents the U or V direction respectively. The value 2 is not yet implemented.

**Return Value:**

**GW\_TEX\_NO\_TILING**

The texture clamped - Any UVW that is bigger than 1 is interpreted as being 1.

**GW\_TEX\_REPEAT**

As the UVW numbers keep getting larger than 1 the image is repeated.

**GW\_TEX\_MIRROR**

If UVW goes beyond 1, the numbers are interpreted as going backwards. So if you had 0 to 2 it would actually go 0 to 1 then 1 down to 0.

**Prototype:**

**virtual void setTexTransform(const Matrix3 &m) = 0;**

**Remarks:**

This method allows one to put an affine transformation on a texture. This

allows you to translate, rotate or scale a texture on an object.

**Parameters:**

**const Matrix3 &m**

The texture transformation matrix.

**Prototype:**

**virtual void beginFrame() = 0;**

**Remarks:**

If a developer is working with an existing 3ds max instance of GraphicsWindow (one of MAX's viewports) this method should NOT be called.

**Prototype:**

**virtual void endFrame() = 0;**

**Remarks:**

As above, if a developer is working with an existing 3ds max instance of GraphicsWindow (one of MAX's viewports) this method should NOT be called.

**Prototype:**

**virtual void setViewport(int x, int y, int w, int h) = 0;**

**Remarks:**

This method sets the clipping boundaries within a viewport within the graphics window. This allows more than one viewport to appear within a single graphics window. It has the side-effect of building a 4x4 viewport matrix. This routine should be called anytime the graphics window is resized, or else rendering will still occur to the old window size. (And since most drivers do not do range-checking since it is too time-costly, this could cause a system crash.)

**Parameters:**

Note: all coordinates are in Windows format, with the origin in the upper left

**int x**

Specifies the left viewport origin.

**int y**

Specifies the top viewport origin.

**int w**

Specifies the viewport width.

**int h**

Specifies the viewport height.

## **Drawing Setup**

### **Prototype:**

```
virtual void resetUpdateRect() = 0;
```

### **Remarks:**

This method resets the update rectangle. The update rectangle is the region of the screen that needs to be updated to reflect items that have changed. When the system is done rendering items, the goal is to only update the region that has actually been altered. This method sets the update rectangle (the region that will be blitted to the display) to invalid. In this way when **enlargeUpdateRect()** is later called, the RECT passed will be used as the region.

### **Prototype:**

```
virtual void enlargeUpdateRect(RECT *rp) = 0;
```

### **Remarks:**

This method enlarges the update rectangle to include the RECT passed. If **rp** is NULL, then the whole window will later be updated.

### **Parameters:**

**RECT \*rp**

Pointer to a rectangle (or NULL).

### **Prototype:**

```
virtual int getUpdateRect(RECT *rp) = 0;
```

### **Remarks:**

This method retrieves the current update rectangle.

**Parameters:****RECT \*rp**

The current update rectangle.

**Return Value:**

Zero if the update rectangle is invalid; otherwise nonzero.

**Prototype:****virtual void setRndLimits(DWORD l) = 0;****Remarks:**

Sets the rendering limits used by primitive calls.

Note: Setting the rendering limits is used in communication between the various parts of 3ds max that handle the display of objects. For example, setting this limit to **GW\_POLY\_EDGES** and then drawing a polygon won't result in a polygon drawn with edges. It only sets a flag that indicates the edge should be drawn.

What happens is as follows. Inside the upper level MAX, part of the code knows that polygon edges have been turned on. However this is not related through the object oriented architecture to the part of 3ds max that does the actual drawing. When 3ds max goes to draw objects it will see that the polygon edge flag is on. This tells it to do two drawing passes -- one to do the polygon, then it calls **outlinePass()** call with TRUE, draws a bunch of edges, then calls **outline Pass()** with FALSE. Thus, the drawing routine is responsible for looking at the flags and drawing appropriately. This method is only responsible setting the limit which can later be checked.

**Parameters:****DWORD l**

Specifies the rendering limit used by the viewport. See [Rendering Limits](#).

**Prototype:****virtual DWORD getRndLimits() = 0;****Remarks:**

Retrieves the rendering limits used by primitive calls. See [Rendering Limits](#).

**Prototype:**

```
virtual DWORD getRndMode() = 0;
```

**Remarks:**

Returns the current rendering mode used by the viewport. This is a subset of the rendering limit, in that any limits imposed by the rendering limit are forced onto the current mode. See [Rendering Limits](#).

**Prototype:**

```
virtual int getMaxStripLength();
```

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the largest number of triangles that can be in a strip

**Prototype:**

```
virtual void setSkipCount(int n) = 0;
```

**Remarks:**

Sets the number of triangles skipped when the viewport is set as a 'Fast View Display' viewport. To disable fastview, specify 1. Note that the GraphicsWindow class doesn't actually do anything with this number other than store it. Since triangles are handed down to GFX one at a time, it is up to the code that feeds triangles to the GraphicsWindow to skip the specified number of triangles. The mesh rendering in 3ds max uses the skip count in this way.

**Parameters:**

**int n**

Specifies that every 'n-th' triangle should be drawn. If set to 2, every other triangle should be drawn.

**Prototype:**

```
virtual int getSkipCount() = 0;
```

**Remarks:**

Returns the current skip count setting.

## Light and Camera Methods

### Prototype:

**virtual int getMaxLights() = 0;**

### Remarks:

Returns the maximum number of lights that may be used by the interactive renderer.

### Prototype:

**virtual void setLight(int num, const Light \*l) = 0;**

### Remarks:

Turns a light on or off, and sets the light parameters. The light's position is set through the current transformation matrix at the time of this call. A particular light is specified by its light number (-1 through **getMaxLights()-1**). Light number -1 is reserved for ambient light. If a particular field in the Light class is not needed for a particular type of light, that field's value is ignored (for example, only the color field is used for ambient light.)

### Parameters:

**int num**

The light number of the light to set. This is a value in the range -1 to **getMaxLights()-1**.

**const Light \*l**

The light class instance used to set the light parameters. If this is NULL, the light is turned off.

### Prototype:

**virtual void setLightExclusion(DWORD exclVec) = 0;**

### Remarks:

This allows a developer to control if a light is used to render an object. There is one bit per light (bits 0 through **getMaxLights()**). If the bit is set the light is NOT used to render the object. If the bit is off, the light IS used. This method allows you to set the exclusion vector controlling the lights.

### Parameters:

## **DWORD exclVec**

The exclusion vector controlling the lights.

### **Prototype:**

```
virtual void setCamera(const Camera &c) = 0;
```

### **Remarks:**

This method is no longer used.

### **Prototype:**

```
virtual void setCameraMatrix(float mat[4][4], Matrix3 *invTM,  
int persp, float hither, float yon) = 0;
```

### **Remarks:**

Sets the properties of the current camera used by the GraphicsWindow.

### **Parameters:**

**float mat[4][4]**

The transformation matrix times the projection matrix.

**Matrix3 \*invTM**

This is the inverse of the affine part of the camera transformation matrix (not the inverse of the projection part).

**int persp**

Nonzero indicates this is a perspective view; 0 is orthogonal.

**float hither**

Near clip value.

**float yon**

Far clip value.

### **Prototype:**

```
virtual void getCameraMatrix(float mat[4][4], Matrix3 *invTM,  
int *persp, float *hither, float *yon) = 0;
```

### **Remarks:**

Retrieves the properties of the current camera.

### **Parameters:**

**float mat[4][4]**

The transformation matrix times the projection matrix.

**Matrix3 \*invTM**

This is the inverse of the affine part of the camera transformation matrix (not the inverse of the projection part).

**int \*persp**

Nonzero indicates this is a perspective view; 0 is orthogonal.

**float \*hither**

Near clip value.

**float \*yon**

Far clip value.

## Material Methods

### Prototype:

```
virtual void setMaterial(const Material &m, int index=0) = 0;
```

### Remarks:

Sets the current rendering material, and modifies the rendering mode parameter for controlling the rasterizer driver. Note: You must have your rendering limit set BEFORE you set the material because the material setting may lower the rendering mode based on the material limits.

### Parameters:

**const Material &m**

The new material to instantiate

**int index=0**

Indicates which material index refers to the material which gets set.

### Prototype:

```
virtual Material *getMaterial() = 0;
```

### Remarks:

Returns the current rendering material.

**Prototype:**

**virtual void setTransparency(DWORD settings) = 0;**

**Remarks:**

Sets the current transparency flags for the current pass.

**Parameters:****DWORD settings**

This can be a combination of GW\_TRANSPARENCY and GW\_TRANSPARENT\_PASS See [Rendering Limits](#). You also use these settings in the Render limits as well.

## Coordinate Transformation Methods

**Prototype:**

**virtual DWORD hTransPoint(const Point3 \*in, IPoint3 \*out) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method converts coordinates to "h" format device coordinates. Note: This method maps points from the GraphicsWindow's current transform to device space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space. Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:****const Point3 \*in**

The input point.

**IPoint3 \*out**

The output point in integer format values in the native device coords for the device. For HEIDI and OpenGL the origin at the lower left. For Direct3D the origin is at the upper left.

**Return Value:**

DWORD containing the clipping flags for the point. If a flag is set it indicates the transformed point lies outside the view volume. See [List of Clip Flags](#).

**Prototype:**

**virtual DWORD wTransPoint(const Point3 \*in, IPoint3 \*out) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to convert coordinates to "w" format device coordinates. Note: This method maps points from the GraphicsWindow's current transform to device space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space. Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**const Point3 \*in**

The input point.

**IPoint3 \*out**

The output point in integer format with the origin at the upper left.

**Return Value:**

DWORD containing the clipping flags for the point. If a flag is set it indicates the transformed point lies outside the view volume. See [List of Clip Flags](#).

**Prototype:**

**virtual DWORD transPoint(const Point3 \*in, Point3 \*out) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to convert coordinates to "h" **floating point** coordinates. This is just a helper routine to avoid building up round-off error. 3ds max uses it just for IK. Note: This method maps points from the GraphicsWindow's current transform to device space. If the GraphicsWindow's transform is set to

the identity matrix then the mapping is done from points specified in world space. Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**const Point3 \*in**

The input point.

**Point3 \*out**

The output point in floating point format with the origin at the lower left.

**Return Value:**

DWORD containing the clipping flags for the point. If a flag is set it indicates the transformed point lies outside the view volume. See [List of Clip Flags](#).

## Drawing Methods

Note: The old device coordinate calls have been changed (or removed). In 3ds max 2.0 and later, methods that start with "**h**" take integer device coordinates with the origin at the lower-left. Calls with a "**w**" in front take Windows device coordinates with the origin at the upper left. Also Note: These "**h**" and "**w**" routines perform NO clipping unless otherwise noted (clipping at this level would be very expensive). Drawing outside the allowable region is likely to cause 3ds max to crash.

**Prototype:**

**virtual void hText(IPoint3 \*xyz, TCHAR \*s) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

Draws 2D fixed font annotation text to the specified location. Note: This routine DOES perform clipping of the text if it is off the screen.

**Parameters:**

**IPoint3 \*xyz**

This is the device coordinate for the text. The origin of the text is at the lower left corner.

**TCHAR \*s**

The text to display.

**Prototype:**

**virtual void hMarker(IPoint3 \*xyz, MarkerType type) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

Draws a marker at the specified location.

**Parameters:**

**IPoint3 \*xyz**

This is the device coordinate for the marker (with the origin at the lower left).

**MarkerType type**

See [List of Marker Types](#).

**Prototype:**

**virtual void hPolyline(int ct, IPoint3 \*xyz, Point3 \*rgb, int closed, int \*es) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method draws a multi-segment polyline.

**Parameters:**

**int ct**

The number of points in the polyline. The maximum number of points that may be used in drawing a polyline is 32.

**IPoint3 \*xyz**

Array of points. These are device coordinates with the origin at the lower left.

**Point3 \*rgb**

If the shade mode is set to smooth and these colors for the vertices are specified the polyline will be drawn Gourand shaded. This is how 3ds max draws lit wireframes for instance. If you simply want ordinary lines (drawn using the line color) pass NULL.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration.

(This is how OpenGL works.)

### **int closed**

If nonzero the first point is connected to the last point, i.e. the polyline is closed.

### **int \*es**

Edge state array. This is an array that Indicates if the 'i-th' edge is one of three state:

#### **GW\_EDGE\_SKIP**

Nonexistent - totally invisible.

#### **GW\_EDGE\_VIS**

Exists and is solid.

#### **GW\_EDGE\_INVIS**

Exists and is hidden - shown as a dotted line.

You may pass NULL for this array and the method will assume that the edges are all solid.

### **Prototype:**

```
void hPolyline(int ct, IPoint3 *xyz, Point3 *rgb, Point3 *uvw, int closed, int *es);
```

### **Remarks:**

This method is available in release 2.0 and later only.

This method draws a multi-segment polyline.

### **Parameters:**

#### **int ct**

The number of points in the polyline. The maximum number of points that may be used in drawing a polyline is 32.

#### **IPoint3 \*xyz**

Array of points. These are device coordinates with the origin at the lower left.

#### **Point3 \*rgb**

If the shade mode is set to smooth and these colors for the vertices are specified the polyline will be drawn Gourand shaded. This is how 3ds max draws lit wireframes for instance. If you simply want ordinary lines (drawn

using the line color) pass NULL.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

**Point3 \*uvw**

This is not currently used. Pass NULL.

**int closed**

If nonzero the first point is connected to the last point, i.e. the polyline is closed.

**int \*es**

Edge state array. This is an array that Indicates if the 'i-th' edge is one of three state:

**GW\_EDGE\_SKIP**

Nonexistent - totally invisible.

**GW\_EDGE\_VIS**

Exists and is solid.

**GW\_EDGE\_INVIS**

Exists and is hidden - shown as a dotted line.

You may pass NULL for this array and the method will assume that the edges are all solid.

**Prototype:**

```
virtual void hPolygon(int ct, IPoint3 *xyz, Point3 *rgb, Point3 *uvw) = 0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method draws a multi-point polygon.

**Parameters:**

**int ct**

The number of points in the polygon.

**IPoint3 \*xyz**

Array of points. These are device coordinates with the origin at the lower left.

### **Point3 \*rgb**

The color values at the vertices. The rendering mode must include **GW\_ILLUM** for these values to be used.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

### **Point3 \*uvw**

The UVW coordinates. The rendering mode must include **GW\_TEXTURE** for these values to be used.

### **Prototype:**

```
virtual void hTriStrip(int ct, IPoint3 *xyz, Point3 *rgb, Point3 *uvw) = 0;
```

### **Remarks:**

This method is available in release 2.0 and later only.

This method is used for drawing a series of triangles specified as 'strips'. It takes a count of 3 or more, and builds triangles in a strip. This sends a lot less data and the underlying graphics library has to set up a lot less data since it can use the previous information to start the rasterization. This results in a significant speed increase.

Note that this routine does no clipping so all the points passed must be within view.

### **Parameters:**

#### **int ct**

The total number of points. After the first two points, each new point is used to create a new triangle.

#### **IPoint3 \*xyz**

The point data with the origin at the lower left. For instance, to draw a quad, the first three points specify the first triangle and the next one is combined with the previous two to complete the square.

The order for these points follows the 'standard' conventions for stripping used in most graphics libraries (for example Direct3D, OpenGL and Heidi).

#### **Point3 \*rgb**

The colors for the vertices.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

**Point3 \*uvw**

The UVW texture coordinates for the vertices.

**Prototype:**

**virtual void wText(IPoint3 \*xyz, TCHAR \*s) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

Draws 2D fixed font annotation text to the specified location. Note: This routine DOES perform clipping of the text if it is off the screen.

**Parameters:**

**IPoint3 \*xyz**

This is the device coordinate for the text. The origin of the text is at the upper left corner.

**TCHAR \*s**

The text to display.

**Prototype:**

**virtual void wMarker(IPoint3 \*xyz, MarkerType type) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

Draws a marker at the specified location.

**Parameters:**

**IPoint3 \*xyz**

This is the device coordinate for the marker.

**MarkerType type**

See [List of Marker Types](#).

**Prototype:**

**virtual void wPolyline(int ct, IPoint3 \*xyz, Point3 \*rgb, int closed, int \*es) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method draws a multi-segment polyline.

**Parameters:**

**int ct**

The number of points in the polyline. The maximum number of points that may be used in drawing a polyline is 32.

**IPoint3 \*xyz**

Array of points. These are device coordinates with the origin at the upper left.

**Point3 \*rgb**

If the shade mode is set to smooth and these colors for the vertices are specified the polyline will be drawn Gourand shaded. This is how 3ds max draws lit wireframes for instance. If you simply want ordinary lines (drawn using the line color) pass NULL.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

**int closed**

If nonzero the first point is connected to the last point, i.e. the polyline is closed.

**int \*es**

Edge state array. This is an array that Indicates if the 'i-th' edge is one of three state:

**GW\_EDGE\_SKIP**

Nonexistent - totally invisible.

**GW\_EDGE\_VIS**

Exists and is solid.

**GW\_EDGE\_INVIS**

Exists and is hidden - shown as a dotted line.

You may pass NULL for this array and the method will assume that the edges are all solid.

**Prototype:**

```
void wPolyline(int ct, IPoint3 *xyz, Point3 *rgb, Point3 *uvw, int closed, int *es);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method draws a multi-segment polyline.

**Parameters:****int ct**

The number of points in the polyline. The maximum number of points that may be used in drawing a polyline is 32.

**IPoint3 \*xyz**

Array of points. These are device coordinates with the origin at the upper left.

**Point3 \*rgb**

If the shade mode is set to smooth and these colors for the vertices are specified the polyline will be drawn Gourand shaded. This is how 3ds max draws lit wireframes for instance. If you simply want ordinary lines (drawn using the line color) pass NULL.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

**Point3 \*uvw**

This is not currently used. Pass NULL.

**int closed**

If nonzero the first point is connected to the last point, i.e. the polyline is closed.

**int \*es**

Edge state array. This is an array that Indicates if the 'i-th' edge is one of three state:

**GW\_EDGE\_SKIP**

Nonexistent - totally invisible.

**GW\_EDGE\_VIS**

Exists and is solid.

### **GW\_EDGE\_INVIS**

Exists and is hidden - shown as a dotted line.

You may pass NULL for this array and the method will assume that the edges are all solid.

#### **Prototype:**

```
virtual void wPolygon(int ct, IPoint3 *xyz, Point3 *rgb, Point3 *uvw) = 0;
```

#### **Remarks:**

This method is available in release 2.0 and later only.

This method draws a multi-point polygon.

#### **Parameters:**

##### **int ct**

The number of points in the polygon.

##### **IPoint3 \*xyz**

Array of points. These are device coordinates with the origin at the upper left.

##### **Point3 \*rgb**

The color values at the vertices. The rendering mode must include **GW\_ILLUM** for these values to be used.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

##### **Point3 \*uvw**

The UVW coordinates. The rendering mode must include **GW\_TEXTURE** for these values to be used.

#### **Prototype:**

```
virtual void wTriStrip(int ct, IPoint3 *xyz, Point3 *rgb, Point3 *uvw) = 0;
```

#### **Remarks:**

This method is available in release 2.0 and later only.

This method is used for drawing a series of triangles specified as 'strips'. It takes a count of 3 or more, and builds triangles in a strip. This sends a lot less data and the underlying graphics library has to set up a lot less data since it can use the previous information to start the rasterization. This results in a significant speed increase.

Note that this routine does no clipping so all the points passed must be within view.

**Parameters:**

**int ct**

The total number of points. After the first two points, each new point is used to create a new triangle.

**IPoint3 \*xyz**

The point data with the origin at the upper left. For instance, to draw a quad, the first three points specify the first triangle and the next one is combined with the previous two to complete the square.

The order for these points follows the 'standard' conventions for stripping used in most graphics libraries (for example Direct3D, OpenGL and Heidi).

**Point3 \*rgb**

The colors for the vertices.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

**Point3 \*uvw**

The UVW texture coordinates for the vertices.

**Prototype:**

**virtual void text(Point3 \*xyz, TCHAR \*s) = 0;**

**Remarks:**

Draws 2D fixed font annotation text to the specified location. This method does perform clipping.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space. Otherwise the points given are transformed by the GraphicsWindow

transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**Point3 \*xyz**

This is the coordinate for the text.

**TCHAR \*s**

The text to display. Note: This routine DOES perform clipping of the text if it is off the screen.

**Prototype:**

**virtual void marker(Point3 \*xyz, MarkerType type) = 0;**

**Remarks:**

Draws a marker at the specified location in world space. This method does perform clipping.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space.

Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**Point3 \*xyz**

This is the coordinate for the marker.

**MarkerType type**

See [List of Marker Types](#).

**Prototype:**

**virtual void polyline(int ct, Point3 \*xyz, Point3 \*rgb, Point3 \*uvw, int closed, int \*es) = 0;**

**Remarks:**

Draws a multi-segment polyline with the coordinates specified in world space.

This method does perform clipping.

Note: The arrays of points and vertex related data all must be at least one element larger than the **ct** parameter that is passed in. The 3D clipper will use the "extra" space to clip as efficiently as possible. If room for the extra element is not provided, 3ds max may crash.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space.

Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

### Parameters:

#### **int ct**

The number of points in the polyline. The maximum number of points that may be used in drawing a polyline is 32.

#### **Point3 \*xyz**

Array of points. This array must be at least one element larger than the **ct** parameter that is passed in. The 3D clipper will use the "extra" space to clip as efficiently as possible. If room for the extra element is not provided, 3ds max will crash.

#### **Point3 \*rgb**

If the shade mode is set to smooth and these colors for the vertices are specified the polyline will be drawn Gourand shaded. This is how 3ds max draws lit wireframes for instance. If you simply want ordinary lines (drawn using the line color) pass NULL.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

#### **Point3 \*uvw**

This is not currently used. Pass NULL.

#### **int closed**

If nonzero the first point is connected to the last point, i.e. the polyline is closed.

**int \*es**

Edge state array. This is an array that Indicates if the 'i-th' edge is one of three state:

**GW\_EDGE\_SKIP**

Nonexistent - totally invisible.

**GW\_EDGE\_VIS**

Exists and is solid.

**GW\_EDGE\_INVIS**

Exists and is hidden - shown as a dotted line.

You may pass NULL for this array and the method will assume that the edges are all solid.

**Prototype:**

```
virtual void polylineN(int ct, Point3 *xyz, Point3 *nor, int closed,  
int *es) = 0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Draws a multi-segment polyline with the coordinates specified in world space. This method takes a polyline with a normal for each vertex. This is used for hardware accelerated lit wireframes (when **GW\_SPT\_GEOM\_ACCEL** is TRUE).

Note: The arrays of points and vertex related data all must be at least one element larger than the **ct** parameter that is passed in. The 3D clipper will use the "extra" space to clip as efficiently as possible. If room for the extra element is not provided, 3ds max may crash.

This method does perform clipping.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space.

Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**int ct**

The number of points in the polyline. The maximum number of points that may be used in drawing a polyline is 32.

**Point3 \*xyz**

Array of points. This array must be at least one element larger than the **ct** parameter that is passed in. The 3D clipper will use the "extra" space to clip as efficiently as possible. If room for the extra element is not provided, 3ds max will crash.

**Point3 \*nor**

The normal values at the vertices, one for each vertex.

**int closed**

If nonzero the first point is connected to the last point, i.e. the polyline is closed.

**int \*es**

Edge state array. This is an array that Indicates if the 'i-th' edge is one of three state:

**GW\_EDGE\_SKIP**

Nonexistent - totally invisible.

**GW\_EDGE\_VIS**

Exists and is solid.

**GW\_EDGE\_INVIS**

Exists and is hidden - shown as a dotted line.

You may pass NULL for this array and the method will assume that the edges are all solid.

**Prototype:**

```
virtual void polygon(int ct, Point3 *xyz, Point3 *rgb, Point3 *uvw)  
= 0;
```

**Remarks:**

Draws a multi-point polygon. Note: All arrays (**xyz**, **rgb**, **uvw**) must be at least one element larger than the **ct** parameter that is passed in. The 3D clipper will use the "extra" space to clip as efficiently as possible. If room for the extra element is not provided, 3ds max may crash.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space. Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**int ct**

The number of points in the polygon.

**Point3 \*xyz**

Array of points.

**Point3 \*rgb**

The color values at the vertices. The rendering mode must include **GW\_ILLUM** for these values to be used.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

**Point3 \*uvw**

The UVW coordinates. The rendering mode must include **GW\_TEXTURE** for these values to be used.

**Prototype:**

**virtual void startSegments() = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to begin efficiently sending a lot of 3D line segments.

First call this method, then call **segment()** many times (with two points), then call **endSegments()**.

**Prototype:**

**virtual void segment(Point3 \*xyz, int vis) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method draws a single 3D line segment between the specified points. Call **startSegments()** once before calling this method.

**Parameters:**

**Point3 \*xyz**

Points to the two line endpoints in world space.

**int vis**

Nonzero for the segment to be visible; zero for invisible.

**Prototype:**

**virtual void endSegments() = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

Call this method after sending 3D line segments with **segment()**.

**Prototype:**

**virtual void polygonN(int ct, Point3 \*xyz, Point3 \*nor, Point3 \*uvw) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

Draws a multi-point polygon. Note: All arrays (**xyz**, **nor**, **uvw**) must be at least one element larger than the **ct** parameter that is passed in. The 3D clipper will use the "extra" space to clip as efficiently as possible. If room for the extra element is not provided, 3ds max will crash.

This method sends in normal vectors instead of color for 3D accelerated rendering (when **GW\_SPT\_GEOM\_ACCEL** is TRUE)

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space.

Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:****int ct**

The number of points in the polygon.

**Point3 \*xyz**

Array of points.

**Point3 \*nor**

The normal values at the vertices, one for each vertex.

**Point3 \*uvw**

The UVW coordinates. The rendering mode must include **GW\_TEXTURE** for these values to be used.

**Prototype:**

```
virtual void triStrip(int ct, Point3 *xyz, Point3 *rgb, Point3 *uvw)  
= 0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is used for drawing a series of triangles specified as 'strips'. It takes a count of 3 or more, and builds triangles in a strip. This sends a lot less data and the underlying graphics library has to set up a lot less data since it can use the previous information to start the rasterization. This results in a significant speed increase.

**Parameters:****int ct**

The total number of points. After the first two points, each new point is used to create a new triangle.

**IPoint3 \*xyz**

The point data. For instance, to draw a quad, the first three points specify the first triangle and the next one is combined with the previous two to complete the square.

The order for these points follows the 'standard' conventions for stripping used in most graphics libraries (for example Direct3D, OpenGL and Heidi).

**Point3 \*rgb**

The colors for the vertices.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

**Point3 \*uvw**

The UVW texture coordinates for the vertices.

**Prototype:**

```
virtual void triStripN(int ct, Point3 *xyz, Point3 *nor, Point3 *uvw) = 0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is used for drawing a series of triangles specified as 'strips'. It takes a count of 3 or more, and builds triangles in a strip. This sends a lot less data and the underlying graphics library has to set up a lot less data since it can use the previous information to start the rasterization. This results in a significant speed increase. This method sends in normal vectors instead of color for 3D accelerated rendering (when **GW\_SPT\_GEOM\_ACCEL** is TRUE)

**Parameters:**

**int ct**

The total number of points. After the first two points, each new point is used to create a new triangle.

**Point3 \*xyz**

The point data. For instance, to draw a quad, the first three points specify the first triangle and the next one is combined with the previous two to complete the square.

The order for these points follows the 'standard' conventions for stripping used in most graphics libraries (for example Direct3D, OpenGL and Heidi).

**Point3 \*nor**

The normal for each vertex.

**Point3 \*uvw**

The UVW texture coordinates for the vertices.

**Prototype:****virtual void startTriangles() = 0;****Remarks:**

This method is available in release 2.0 and later only.

This method is called to begin sending a series of non-stripped triangles to render. Call this method, then any of the **triangle\*()** methods many times, then **endTriangles()** to finish.

**Prototype:****virtual void triangle(Point3 \*xyz, Point3 \*rgb) = 0;****Remarks:**

This method is available in release 2.0 and later only.

This method sends a single non-stripped triangle to render. Call **startTriangles()** first.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space.

Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:****Point3 \*xyz**

The three points for the triangle.

**Point3 \*rgb**

The color for each vertex.

Note: The use of these colors is not supported under the OpenGL driver. The rgb values are ignored. Only the current material is taken into consideration. (This is how OpenGL works.)

**Prototype:****virtual void triangleN(Point3 \*xyz, Point3 \*nor, Point3 \*uvw) = 0;****Remarks:**

This method is available in release 2.0 and later only.

This method draws a single triangle by specifying the vertex points in world space, a normal, and texture coordinates for each vertex.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space.

Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**Point3 \*xyz**

The three points for the triangle.

**Point3 \*nor**

The three normals for the triangle.

**Point3 \*uvw**

The texture coordinate for each vertex.

**Prototype:**

**virtual void triangleNC(Point3 \*xyz, Point3 \*nor, Point3 \*rgb) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method draws a single triangle by specifying the vertex points in world space, a normal, and a color for each vertex.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space.

Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**Point3 \*xyz**

The three points for the triangle.

**Point3 \*nor**

The normal for each vertex.

**Point3 \*rgb**

The color for each vertex.

**Prototype:**

**virtual void triangleNCT(Point3 \*xyz, Point3 \*nor, Point3 \*rgb,  
Point3 \*uvw) = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method draws a single triangle by specifying the vertex points in world space, a normal, a color, and a texture coordinate for each vertex.

Note: This method maps points from the GraphicsWindow's current transform to screen space. If the GraphicsWindow's transform is set to the identity matrix then the mapping is done from points specified in world space.

Otherwise the points given are transformed by the GraphicsWindow transform, and are **then** considered to be in world space. Thus, to get a world-space to screen-space conversion, you need to set the transform to the identity with **gw->setTransform(Matrix3(1))**.

**Parameters:**

**Point3 \*xyz**

The three points for the triangle.

**Point3 \*nor**

The normal for each vertex.

**Point3 \*rgb**

The color for each vertex.

**Point3 \*uvw**

The texture coordinate for each vertex.

**Prototype:**

**virtual void endTriangles() = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

Call this method to finish rendering triangles. See **startTriangles()** above.

**Prototype:**

```
virtual void lightVertex(const Point3 &pos, const Point3 &nor,  
Point3 &rgb) = 0;
```

**Remarks:**

Lights a vertex, using all the current lights. The vertex appears to be transformed using the current transformation matrix, although actually the calculations are done using back-transformed light coordinates (for speed). The vertex position and normal are passed in, and a color is returned. The rendering uses the current material.

**Parameters:**

**const Point3 &pos**

Vertex position.

**const Point3 &nor**

Vertex normal.

**Point3 &rgb**

Returned color.

**Prototype:**

```
virtual void updateScreen() = 0;
```

**Remarks:**

This method is used internally and should not be called by plug-in developers.

**Prototype:**

```
virtual void setColor(ColorType t, float r, float g, float b) = 0;
```

**Remarks:**

Sets the RGB color used for the specified drawing type (line, fill, text, clear).

**Parameters:**

**ColorType t**

One of the following values:

**LINE\_COLOR**

Line drawing color.

**FILL\_COLOR**

Polygon fill color.

**TEXT\_COLOR**

Text drawing color.

**CLEAR\_COLOR**

The color that the viewport is cleared to when you call **clearScreen()**.

**float r**

Specifies the red amount 0.0 - 1.0.

**float g**

Specifies the green amount 0.0 - 1.0.

**float b**

Specifies the blue amount 0.0 - 1.0.

**Prototype:**

```
virtual void clearScreen(RECT *rp, int useBkg = FALSE) = 0;
```

**Remarks:**

Clears the specified rectangular region of the screen.

**Parameters:****RECT \*rp**

Specifies the rectangular region to clear.

**int useBkg = FALSE**

Specifies if the background should be used to fill the cleared area. Nonzero indicate the background should be used; 0 indicates the 'clear' color should be used (see **setColor()** above).

**Hit Testing Methods**

The following methods are used for hit testing. Typically you use these methods in the following sequence:

You set a hit regions using **setHitRegion()**.

You clear the hit code using **clearHitCode()**.

You 'render' a primitive using the mode **GW\_PICK** and at whatever level you want to check at. For example, if you are interested in vertex hit testing, you would 'render' a series of markers (using **iWinMarker()** with **PLUS\_SIGN\_MRKR**) using a rendering mode that included **GW\_PICK**. The **GW\_PICK** 'rendering' mode will cause the system to perform hit testing rather than actually render the item.

After each item is 'rendered' you check to see if a hit was made using **checkHitCode()**. This method returns TRUE or FALSE based on if a hit was found. If you are looking for just a single hit, you could abort as soon as **checkHitCode()** returns TRUE. If you are interested in the closest hit, you loop through all the items calling **clearHitCode()**, **checkHitCode()** and **getHitDistance()**. After checking all the items you simply choose the smallest hit distance from all the items you've checked.

**Prototype:**

**virtual void setHitRegion(HitRegion \*rgn) = 0;**

**Remarks:**

Sets the hit region used for hit testing. See [Class HitRegion](#).

**Parameters:**

**HitRegion \*rgn**

The hit region to use.

**Prototype:**

**virtual void clearHitCode() = 0;**

**Remarks:**

This method clears the hit code. Call this method before performing a hit test.

**Prototype:**

**virtual BOOL checkHitCode() = 0;**

**Remarks:**

Returns TRUE if the hit code is set indicating a hit was made; otherwise FALSE.

**Prototype:**

```
virtual DWORD getHitDistance() = 0;
```

**Remarks:**

If **checkHitCode()** returns TRUE you may call this method to return the hit distance. In wireframe mode this is the distance to the line. In shaded mode, this is the z distance. This allows you to perform 'smart' hit testing by choosing the item with the smallest hit distance. This method only returns meaningful values when the hit region is a point.

**Prototype:**

```
virtual void setHitCode(BOOL h)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows drawing code to manually set the state of the hit code, which is returned by the **checkHitCode()** method. For more information see the topic on [Hit Testing](#).

The new methods **setHitDistance()** and **setHitCode()** make it possible to work with GraphicsWindow hit-testing in otherwise impossible situations. Why are they necessary? An example from is shown below. The patch object contains bezier spline-based edges which can consist of up to 102 vertices. Since the GraphicsWindow::polyline function can only plot lines with up to 32 vertices, it is impossible to plot these in a single call to the polyline function. Multiple calls to the polyline call do not return a proper hitcode when using a "window"-type hit region. By using the new **setHitCode()** method, code can properly handle this situation. The code below shows the function in use from the **PatchMesh::renderEdge** method:

```
int steps = GetMeshSteps();  
int segNum = steps+2;  
float fsegNum = (float) (segNum-1);  
// If steps are too high for GraphicsWindow's buffer, we must  
draw it manually  
if((steps + 2) > GW_MAX_VERTS) {  
    Point3 line[2];
```

```

Point3 prev,current(.0f,.0f,.0f);
BOOL hitAll = TRUE;
BOOL hitAny = FALSE;
DWORD hitDist = 0xffffffff;
for(int terp = 0; terp < segNum; terp++)
{
    prev = current;
    current = work.InterpCurve3D((float)terp / fsegNum);
    if (terp != 0)
    {
        line[0] = prev;
        line[1] = current;
        gw->clearHitCode();
        gw->polyline(2, line, NULL, NULL, 0, NULL);
        if(gw->checkHitCode()) {
            hitAny = TRUE;
            if(gw->getHitDistance() < hitDist)
                hitDist = gw->getHitDistance();
        }
        else hitAll = FALSE;
    }
}
if(hr && !hr->crossing && hr->type != POINT_RGN)
    gw->setHitCode(hitAll);
else
    gw->setHitCode(hitAny);
gw->setHitDistance(hitDist);
} else {
for(int terp = 0; terp < segNum; terp++)
    fixedBuf[terp] = work.InterpCurve3D((float)terp / fsegNum);
}

```

```
gw->polyline(steps+2, fixedBuf, NULL, NULL, 0, NULL);  
}
```

Note that the **gw->polyline** call is preceded by a call to **clearHitCode()**, and followed by code which checks the hit code, maintaining "hitAny" and "hitAll" flags. When all the segments are drawn, the **gw->setHitCode()** call is made, setting the hit code depending on the hit region type. When the code which called this function checks the GraphicsWindow's hit code, it will contain the proper value. This code also keeps track of the closest hit distance and places that into the GraphicsWindow when all line segments are drawn.

**Parameters:**

**BOOL h**

Set to TRUE if the hit code is set, otherwise FALSE.

**Prototype:**

```
virtual void setHitDistance(DWORD d)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows drawing code to manually set the hit distance, which is returned by the **getHitDistance()** method. For more information see the topic on [Hit Testing](#).

**Parameters:**

**DWORD d**

In wireframe mode this is the distance to the line. In shaded mode, this is the z distance.

**Prototype:**

```
virtual float interpWorld(Point3 *world1, Point3 *world2, float  
sParam, Point3 *interpPt) = 0;
```

**Remarks:**

This method is used internally.

**Utility Functions (not part of class GraphicsWindows):**

**Function:**

**GraphicsWindow \*createGW(HWND hWnd, GWinSetup &gws);**

**Remarks:**

This function is used internally to create a new graphics window. Use of this method is not supported for plug-ins.

**Function:**

**void getRegionRect(HitRegion \*hr, RECT \*rect);**

**Remarks:**

Returns a bounding rectangle that encloses the entire hit region. For example if the hit regions was a fence region, this method would return the smallest rectangle that included the entire set of fence region points.

**Parameters:**

**HitRegion \*hr**

The hit region to check.

**RECT \*rect**

The returned bounding rectangle.

**Function:**

**BOOL pointInRegion(int x, int y, HitRegion \*hr);**

**Remarks:**

Returns TRUE if the specified point is inside the region **hr**; otherwise FALSE.

**Function:**

**int distToLine(int x, int y, int \*p1, int \*p2);**

**Remarks:**

Returns the signed distance from **x,y** to the line defined by **p1->p2**.

**Function:**

**int lineCrossesRect(RECT \*rc, int \*p1, int \*p2);**

**Remarks:**

Returns nonzero if the line defined by **p1->p2** crosses into the RECT and 0

otherwise.

**Function:**

**int segCrossesCircle(int cx, int cy, int r, int \*p1, int \*p2);**

**Remarks:**

Returns nonzero if the line defined by **p1**->**p2** crosses the circle center at (**cx**, **cy**) with a radius of **r** 0 otherwise.

**Function:**

**BOOL insideTriangle(IPoint3 &p0, IPoint3 &p1, IPoint3 &p2, IPoint3 &q);**

**Remarks:**

Returns TRUE if the point passed is inside the specified triangle.

**Parameters:**

**IPoint3 &p0**

The first point of the triangle.

**IPoint3 &p1**

The second point of the triangle.

**IPoint3 &p2**

The third point of the triangle.

**IPoint3 &q**

The point to check.

**Return Value:**

Returns TRUE if the point passed is inside the specified triangle; otherwise FALSE.

**Function:**

**int getZfromTriangle(IPoint3 &p0, IPoint3 &p1, IPoint3 &p2, IPoint3 &q);**

**Remarks:**

Returns the z value of where the projected screen point **q** would intersect the triangle defined by (**p0**, **p1**, **p2**).

**Parameters:**

**IPoint3 &p0**

The first point of the triangle.

**IPoint3 &p1**

The second point of the triangle.

**IPoint3 &p2**

The third point of the triangle.

**IPoint3 &q**

The screen point to check.

**Prototype:**

**HINSTANCE GetGraphicsLibHandle(TCHAR \*driverLibName);**

**Remarks:**

This function is available in release 2.0 and later only.

This function is not supported for use in the SDK.

**Prototype:**

**BOOL GraphicsSystemIsAvailable(HINSTANCE drv);**

**Remarks:**

This function is available in release 2.0 and later only.

This function is not supported for use in the SDK.

**Prototype:**

**BOOL GraphicsSystemCanConfigure(HINSTANCE drv);**

**Remarks:**

This function is available in release 2.0 and later only.

This function is not supported for use in the SDK.

**Prototype:**

**BOOL GraphicsSystemConfigure(HWND hWnd, HINSTANCE drv);**

**Remarks:**

This function is available in release 2.0 and later only.

This function is not supported for use in the SDK.

**Prototype:**

**void FreeGraphicsLibHandle(HINSTANCE drv);**

**Remarks:**

This function is available in release 2.0 and later only.

This function is not supported for use in the SDK.

# Class CommandMode

See Also: [Class MouseCallBack](#), [Class ChangeForegroundCallback](#), [Foreground / Background Planes](#).

class CommandMode

## Description:

This base class allows the developer to create a command mode that handles processing user interaction using the mouse in the viewports.

See Also: The Advanced Topics section [Command Modes and Mouse Procs](#). There are methods in 3ds max's [Interface class](#) to set and get the current command mode.

## Methods:

## Prototype:

```
virtual int Class()=0;
```

## Remarks:

Implemented by the Plug-In.

Returns the Class of the command mode. The class describes the type of command mode this is. If the developer is defining a command mode to be used as part of the sub-object manipulation (Move, Rotate, and Scale) then one of the following pre-defined values should be used:

```
MOVE_COMMAND  
ROTATE_COMMAND  
SCALE_COMMAND  
USCALE_COMMAND  
SQUASH_COMMAND
```

If one of these other values is appropriate it may be used. If not, the developer is free to define their own (as an **int**).

```
VIEWPORT_COMMAND  
SELECT_COMMAND  
HIERARCHY_COMMAND  
CREATE_COMMAND  
MODIFY_COMMAND  
MOTION_COMMAND
```

**ANIMATION\_COMMAND**  
**CAMERA\_COMMAND**  
**NULL\_COMMAND**  
**DISPLAY\_COMMAND**  
**SPOTLIGHT\_COMMAND**  
**PICK\_COMMAND**

**Return Value:**

The Class of the command mode.

**Prototype:**

**virtual int SuperClass()**

**Remarks:**

Implemented by the Plug-In.

This method can be ignored. The default implementation should be used.

**Default Implementation:**

**{ return 0; }**

**Prototype:**

**virtual int ID()=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the ID of the command mode. This value should be the constant **CID\_USER** plus some random value chosen by the developer.

As an example, this method could be implemented as:

**{ CID\_USER+0x1423; }**

In the special case of the developer implementing a custom command mode to be used as part of sub-object manipulation (Move, Rotate or Scale) the value for ID() should be one of the following values:

**CID\_SUBOBJMOVE**  
**CID\_SUBOBJROTATE**  
**CID\_SUBOBJSCALE**  
**CID\_SUBOBJUSCALE**

## **CID\_SUBOBSQUASH**

Note: if two command modes conflict in this ID value, it is **not** a problem, so the uniqueness is not strictly required. However, this **ID()** method is often used to check which mode is active, so unless the value for your command mode is identifiable via this ID, you may not be able to recognize if your mode is indeed the active one.

### **Prototype:**

```
virtual MouseCallback *MouseProc(int *numPoints)=0;
```

### **Remarks:**

Implemented by the Plug-In.

This method establishes the number of points required by the command mode and returns a pointer to the mouse callback object that is used to process the user input.

### **Parameters:**

**int \*numPoints**

This is where to store the number of points used by the CommandMode. If the plug-in needs to use an undetermined number of points it can specify a large number for this value. When the mouse proc has finished processing points it returns FALSE to stop the point processing before the number of points specified here have been entered.

### **Return Value:**

A pointer to the mouse callback object that is used to process the user input.

See Also: [Class MouseCallback](#).

### **Prototype:**

```
virtual ChangeForegroundCallback *ChangeFGProc()=0;
```

### **Remarks:**

Implemented by the Plug-In.

Returns a pointer to a callback procedure that flags nodes that belong in the foreground plane. Plug-ins typically use a standard callback object provided by the system that flags all nodes dependent on the plug-in object. This ensures that when the plug-in object changes, any nodes that change as a result

will be in the foreground plane, making redraw time faster.

These constants may be specified to use one of the standard callback objects instead of an actual FG proc. For example **{return**

**CHANGE\_FG\_SELECTED;}**

**CHANGE\_FG\_SELECTED**

Selected nodes are flagged.

**CHANGE\_FG\_ANIMATED**

Nodes that are animated are flagged.

**Return Value:**

A pointer to a callback procedure that flags nodes that belong in the foreground plane.

See Also: For additional information see the Advanced Topics section [Foreground / Background Planes](#).

**Prototype:**

**virtual BOOL ChangeFG( CommandMode \*oldMode )=0;**

**Remarks:**

Implemented by the Plug-In.

This method returns TRUE if the command mode needs to change the foreground proc (using **ChangeFGProc()**) and FALSE if it does not. A command mode that does not involve any redrawing of the viewports can just return FALSE.

**Parameters:**

**CommandMode \*oldMode**

This is the command mode that is currently in place. This may be used for comparison with a potential new mode. See the sample code below.

**Sample Code:**

The sample code below checks to see if the command mode is already **CHANGE\_FG\_SELECTED**. If it is there is no reason to change to foreground proc to this mode so the method returns FALSE. If a different mode is in place TRUE is returned.

```
BOOL ChangeFG( CommandMode *oldMode )  
{ return (oldMode->ChangeFGProc() !=
```

```
CHANGE_FG_SELECTED); }  
ChangeForegroundCallback *ChangeFGProc()  
{ return CHANGE_FG_SELECTED; }
```

**Prototype:**

```
virtual void EnterMode()=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called when a command mode becomes active. Usually a developer responds by changing the state of a control to indicate to the user that they have entered the mode. Typically this means pushing in a tool button. When the mode is finished the button should be returned to normal (see **ExitMode()** below).

Note: A developer should use the standard color **GREEN\_WASH** for check buttons that instigate a command mode. While the command mode is active the button should be displayed in **GREEN\_WASH**. See [Class ICustButton](#) (specifically the method **SetHighlightColor()**) for more details.

**Sample Code:**

```
iPick->SetHighlightColor(GREEN_WASH);  
iPick->SetCheck(TRUE);
```

**Prototype:**

```
virtual void ExitMode()=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called when the active command mode is replaced by a different mode. Typically a developer implements this method to set the state of the control that invoked the command mode to the 'out' state. See [Class ICustButton](#) (specifically the method **SetCheck()**).

**Sample Code:**

```
iPick->SetCheck(FALSE);
```

# Class MouseCallBack

See Also: [Class CommandMode](#).

```
class MouseCallBack : public BaseInterfaceServer
```

## Description:

This class is used to allow a developer to capture the mouse events entered by the user and process them. To create a mouse call back, derive a sub class of this class and implement the **proc()** method.

## Methods:

### Prototype:

```
virtual int proc(HWND hwnd, int msg, int point, int flags, IPoint2  
m );
```

### Remarks:

Implemented by the Plug-In.

This method is called when a mouse event takes place to handles its processing.

### Parameters:

#### **HWND hwnd**

The window handle of the window the user clicked in. This is one of the viewports. An interface to the viewport can be obtained from the system, given this window handle. See [Class Interface](#) and review the methods below:

#### **ViewExp \*GetViewport( HWND hwnd )**

This method gets a viewport interface given the window handle.

#### **void ReleaseViewport( ViewExp \*vpt )**

When the developer is done with the viewport interface they should call this method to release it.

#### **int msg**

This message describes the type of event that occurred. See [List of Mouse Callback Messages](#).

#### **int point**

The point number. this is 0 for the first click, 1 for the second, etc.

#### **int flags**

These flags describe the state of the mouse buttons. See [List of Mouse Callback Flags](#).

### **IPoint2 m**

The 2D screen point that the user clicked on. Methods in the viewport interface allow this point to be converted into a world space ray or a 3D view space point. A world space ray can be intersected with the active construction plane which results in a point on the active construction plane. See [Class ViewExp](#).

### **Return Value:**

TRUE indicates the proc should continue to process points; FALSE means stop processing points. If a plug-in supplied a large number of points in a command mode that uses this mouse proc, FALSE can be returned to abort the processing before all the points have been entered.

### **Prototype:**

**virtual int override(int mode)**

### **Remarks:**

Implemented by the Plug-In.

This method is used to override the default drag mode. Most plug-in will not need to replace the default implementation of this method. What this does is change the way the messages are sent relative to the mouse clicking.

Normally the messages are sent as follows: When the user clicks down this generates a **MOUSE\_POINT** message. Then the user drags the mouse with the button down and a series of **MOUSE\_MOVE** messages are sent. When they let up on the mouse button a **MOUSE\_POINT** messages is generated. Then as the mouse is moved a series of **MOUSE\_MOVE** messages are sent. Then they click down on the mouse again, but this time a point message is not generated until the button is released. All future points are then only sent after the mouse button has been pressed and released.

### **Parameters:**

**int mode**

The current drag mode. See below.

### **Return Value:**

One of the following drag modes should be returned:

## **CLICK\_MODE\_DEFAULT**

Returned to indicate the use of the system mouse mode.

## **CLICK\_DRAG\_CLICK**

This is the default behavior as described above.

## **CLICK\_MOVE\_CLICK**

In this mode, the first point is entered by clicking the mouse button down and then letting it up. This generates point 0. In other words, a **MOUSE\_POINT** message is only generated after the user has pressed and released the mouse button.

## **CLICK\_DOWN\_POINT**

In this mode, point messages are sent on mouse-down only.

### **Default Implementation:**

```
{ return mode; }
```

### **Sample Code:**

A sample program that uses the override method is **\MAXSDK\SAMPLES\OBJECTS\SPLINE.CPP**. It uses **CLICK\_DOWN\_POINT**.

### **Prototype:**

```
virtual BOOL SupportTransformGizmo();
```

### **Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method returns TRUE if the mouse proc supports a transform gizmo; otherwise FALSE.

This method, and **DeactivateTransformGizmo()** below, are normally implemented by the selection processor and the existing mouse procs. In case a special implementation is done, the following describes what needs to be done:

Since the transform gizmo is dependent on the Command Mode, the **MouseCallback** itself decides if it supports the use of the transform gizmo. When the node axis (or transform gizmo) is redrawn, the system will ask the command mode's mouse proc if it supports transform gizmos, and if it does, it

will draw a gizmo, instead of the regular node axis.

The same things happens with the main selection processor in 3ds max. When the mouse is moved, the selection processor itself asks if the MouseCallback supports transform gizmos, and if it does, it will hit test the gizmo in a **MOUSE\_FREEMOVE** or **MOUSE\_POINT** message. If any of the transform gizmos hit test flags are passed into the mouse procs hit tester, the transform gizmo should be hit tested as well (using **Interface::HitTestTransformGizmo()**)

When hit testing the gizmo, different flags will be passed in:

**HIT\_TRANSFORMGIZMO** is passed in on a **MOUSE\_FREEMOVE** message so that the axis is hit tested and it highlights if it is hit, but it doesn't actually switch the transform mode.

In case of a **MOUSE\_POINT**, the flag will be **HIT\_SWITCH\_GIZMO**, and if the axis is hit, the 'hit' transform mode will be pushed on the transform mode stack.

When the mouse is released (**MOUSE\_POINT** (pt==1), or **MOUSE\_ABORT**, then the axis constraint should pop back to the existing one, and **DeactivateTransformGizmo()** is called. Inside **DeactivateTransformGizmo()** the normal implementation is to pop the axis mode back. It should also maintain a flag (set it if **HitTestTransformGizmo()** returns true and the **HIT\_SWITCH\_GIZMO** is set, and clear it in **DeactivateTransformGizmo()**). The flag is needed because you should not call **Interface::PopAxisMode()** unless the axis mode was previously popped.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void DeactivateTransformGizmo();
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method is called to deactivate the transform gizmo. See the note above in

**SupportTransformGizmo().**

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL SupportAutoGrid();
```

**Remarks:**

This method is available in release 3.0 and later only.

Mouse procs wishing to utilize the AutoGrid feature should override this method by returning TRUE and then make the appropriate calls to the **ViewExp** methods **TrackImpliciGrid()**, **CommitImplicitGrid()** and **ReleaseImplicitGrid()** from the body of their classes **proc()** method. For sample code see

```
\\MAXSDK\\SAMPLES\\MODIFIERS\\SURFWRAP\\SURFWRAP.CI
```

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL TolerateOrthoMode();
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method returns TRUE if Ortho Mode makes sense for this creation; FALSE if it doesn't. In general this is only TRUE for splines and such.

**Default Implementation:**

```
{return FALSE; }
```

**Prototype:**

```
void setMouseManager(MouseManager *mm)
```

**Remarks:**

This method is used internally.

**Prototype:**

**MouseManager \*getManager()**

**Remarks:**

This method is used internally.

**The following functions are not part of this class but are available for use:**

**Function:**

**BOOL GetInMouseAbort();**

**Remarks:**

This global function is available in release 2.0 and later only.

This function is not part of this class but is available for use. It returns TRUE if **any** mouse proc is currently in the process of aborting a mouse proc; otherwise FALSE.

For instance, a developer may be using the **Animatable::MouseCycleCompleted()** method to put up a dialog box, but need to not put it up if the mouse proc was aborting. This method provides a way to know if indeed the mouse proc is aborting so the dialog won't be displayed inside **MouseCycleCompleted()**.

**Return Value:**

TRUE if aborting; otherwise FALSE.

**Function:**

**float GetPerspMouseSpeed();**

**Remarks:**

This global function is not part of class MouseCallBack but is available for use.

Returns the mouse speed value used in perspective viewports. This value corresponds to the value that the user may set using the 3ds max command in the File / Preferences . . . Move / Rotate Transform area.

**Function:**

**void SetPerspMouseSpeed(float speed);**

**Remarks:**

This global function is not part of class MouseCallBack but is available for use.

Sets the mouse speed value used in perspective viewports. This value corresponds to the value that the user may set using the 3ds max command in the File / Preferences . . . Move / Rotate Transform area.

**Parameters:**

**float speed**

The value to set.

**Function:**

**void SetMoveModeType(int moveType);**

**Remarks:**

This global function is not part of class MouseCallBack but is available for use.

Sets the mouse move mode. This value corresponds to the value that the user may set using the 3ds max command in the File / Preferences . . . Move / Rotate Transform area.

**Parameters:**

**int moveType**

One of the following values:

**MOVE\_INTERSECTION**

**MOVE\_PROJECTION**

**Function:**

**int GetMoveModeType();**

**Remarks:**

This global function is not part of class MouseCallBack but is available for use.

Returns the mouse move mode. This value corresponds to the value that the user may set using the 3ds max command in the File / Preferences . . . Move / Rotate Transform area.

**Return Value:**

One of the following values:

**MOVE\_INTERSECTION**

**MOVE\_PROJECTION**

## Class IParamArray

See Also: [Class IParamBlock](#), [Class IParamMap](#).

class IParamArray

### Description:

This class represents a virtual array of parameters. Parameter are accessed using an integer index and **GetValue()** and **SetValue()** methods. Parameter blocks are one such implementation of this class, but it can also be useful to implement a class that abstracts non-parameter block variables. The ParamMap class uses this base class so that a ParamMap can be used to control UI for not only parameter blocks but also variables stored outside of parameter blocks. The Advanced Topics section on [Parameter Maps](#) discusses how this is done.

### Methods:

Default implementation of these methods are provided which simply return FALSE.

### Prototype:

**virtual BOOL SetValue(int i, TimeValue t, float v);**

### Remarks:

Implemented by the Plug-In.

This method is called when the system needs to store a value into a variable. There are overloaded functions for each type of value to set (int, float, and Point3). Each method has three parameters. Below is the float version - the others are similar.

### Parameters:

**int i**

This is the index into the virtual array of the value to set.

**TimeValue t**

This is the time at which to set the value.

**float v**

This is the value to set.

### Return Value:

This is not currently used.

**Prototype:**

**virtual BOOL SetValue(int i, TimeValue t, int v)**

**Remarks:**

Implemented by the Plug-In.

This is the integer version of above.

**Prototype:**

**virtual BOOL SetValue(int i, TimeValue t, Point3& v)**

**Remarks:**

Implemented by the Plug-In.

This is the Point3 version of above.

**Prototype:**

**virtual BOOL GetValue(int i, TimeValue t, float &v, Interval &ivalid)**

**Remarks:**

Implemented by the Plug-In.

Whenever the developer needs to retrieve a value from the parameter block, the **GetValue()** method is used. There are overloaded functions for each type of value to retrieve (int, float, Point3, and Color). Each method has four parameters:

**Parameters:**

**int i**

This is the index into the virtual array of the value to retrieve.

**TimeValue t**

This is the time at which to retrieve the value. For constants pass 0.

**float &v**

This is the value to retrieve.

**Interval &ivalid**

This is the validity interval to update to reflect the validity of this parameter.

**Return Value:**

If the value was retrieved TRUE is returned; otherwise FALSE is returned.

**Prototype:**

**virtual BOOL GetValue(int i, TimeValue t, int &v, Interval &ivalid)**

**Remarks:**

This is the integer version of above.

**Prototype:**

**virtual BOOL GetValue(int i, TimeValue t, Point3 &v, Interval &ivalid)**

**Remarks:**

This is the Point3 version of above.

**Prototype:**

**IParamBlock \*GetParamBlock()**

**Remarks:**

Implemented by the System.

If the array uses a parameter block, this method will return a pointer to it, otherwise it will return NULL. Note that casting won't work because of multiple inheritance.

**Return Value:**

Returns a pointer to the parameter block if one is used; NULL otherwise.

**Prototype:**

**virtual BOOL KeyFrameAtTime(int i, TimeValue t);**

**Remarks:**

This method is available in release 2.0 and later only.

Checks to see if a keyframe exists for the given parameter at the given time. Returns TRUE if a keyframe exists at the specified time; otherwise FALSE.

**Parameters:**

**int i**

Zero based index of the parameter to check.

**TimeValue t**

The time to check.

**Default Implementation:**

```
{return FALSE;}
```

# Class ICustomControl

See Also: [Custom Controls](#).

class ICustomControl : public InterfaceServer

## Description:

This is the base class from which the 3ds max custom controls are derived. All methods of this class are implemented by the system.

## Methods:

### Prototype:

**virtual HWND GetHwnd()=0;**

### Remarks:

Returns the handle of the control.

### Prototype:

**virtual void Enable(BOOL onOff=TRUE)=0;**

### Remarks:

This method is used to enable the control so it may be operated by the user.

### Parameters:

**BOOL onOff=TRUE**

This parameter is available in release 2.0 and later only.

TRUE to enable; FALSE to disable.

### Prototype:

**virtual void Enable2(BOOL onOff=TRUE)=0;**

### Remarks:

This method is available in release 2.0 and later only.

This method is used internally and should not be called by plug-in developers.

This second enable function is used to disable and enable custom controls when the associated parameter has a non-keyframable parameter. The effective enable state is the AND of these two enable bits.

For example, when a parameter has a controller plugged into it, and the

controller is not keyframable, any spinner control associated with it won't be effective. That's because the controller doesn't take input -- it only outputs values. To prevent the user from being confused by the ineffectiveness of the spinner the control it's automatically disabled by the system using this method.

**Parameters:**

**BOOL onOff=TRUE**

TRUE to enable; FALSE to disable.

**Prototype:**

**virtual void Disable()=0;**

**Remarks:**

This method is used to disable the control so it may not be selected or used. When disabled, the control usually appears grayed out.

**Prototype:**

**virtual BOOL IsEnabled()=0;**

**Remarks:**

This returns TRUE if the control is enabled and FALSE if it is disabled.

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);**

**Remarks:**

This method is available in release 3.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

**Parameters:**

**int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value (defined uniquely for each **cmd**).

**Default Implementation:**

```
{ return 0; }
```

## Class ICustEdit

See Also: [Class ICustomControl](#), [Custom Controls](#).

```
class ICustEdit : public ICustomControl
```

### Description:

This control is a simple text input control. The user may type any string into the field and the plug-in is notified when the user presses the ENTER key. There are also methods to parse and return integer and floating point values entered in the control.



To initialize the pointer to the control call:

### Prototype:

```
ICustEdit *GetICustEdit(HWND hCtrl);
```

To release the control call:

### Prototype:

```
ReleaseICustEdit(ICustEdit *ice);
```

The value to use in the Class field of the Custom Control Properties dialog is:

### CustEdit

The following messages may be sent by the edit control:

This message is sent when the control gets focus or when the user presses the ENTER key while using the control.

#### **WM\_CUSTEDIT\_ENTER**

**wParam** contains the custom edit control resource ID.

**lParam** contains the **HWND** of custom edit control.

### Methods:

### Prototype:

```
virtual void GetText( TCHAR *text, int ct )=0;
```

### Remarks:

This retrieves the text entered into the control.

### Parameters:

**TCHAR \*text**

Storage for the text to retrieve.

**int ct**

Specifies the maximum length of the string returned.

**Prototype:**

**virtual void SetText( TCHAR \*text )=0;**

**Remarks:**

This method places the text into the control for editing.

**Parameters:**

**TCHAR \*text**

The text to place in the control.

**Prototype:**

**virtual void SetText( int i )=0;**

**Remarks:**

This method allows you to pass in an integer value to the control. The integer is converted to a string and displayed in the control.

**Parameters:**

**int i**

This value is converted to a string and displayed in the control.

**Prototype:**

**virtual void SetText( float f, int precision=3 )=0;**

**Remarks:**

This method allows you to pass in a floating point value to the control. The float is converted to a string and displayed in the control.

**Parameters:**

**float f**

This value is converted to a string and displayed in the control.

**int precision=3**

The precision argument is simply the number of decimal places that get represented in the string that appears in the edit field. So if the arguments were

(1.0f/3.0f, 3) then the string "0.333" would appear in the edit field.

**Prototype:**

**virtual int GetInt(BOOL \*valid=NULL)=0;**

**Remarks:**

This method parses and returns an integer value from the control.

**Parameters:**

**BOOL \*valid=NULL**

This pointer, if passed, is set to TRUE if the input is 'valid'; otherwise FALSE. FALSE indicates that something caused the parsing of the input to terminate improperly. An example is a non-numeric character. So for example, if the user entered "123jkfk sdf" into the field the valid pointer would be set to FALSE.

**Prototype:**

**virtual float GetFloat(BOOL \*valid=NULL)=0;**

**Remarks:**

This method parses and returns a floating point value from the control.

**Parameters:**

**BOOL \*valid=NULL**

This pointer, if passed, is set to TRUE if the input is 'valid'; otherwise FALSE. FALSE indicates that something caused the parsing of the input to terminate improperly. An example is a non-numeric character. So for example, if the user entered "123jkfk sdf" into the field this pointer would be set to FALSE.

**Prototype:**

**virtual void SetLeading(int lead)=0;**

**Remarks:**

A developer doesn't normally need to call this method. This offsets the text vertically in the edit control.

**Parameters:**

**int lead**

This parameter specifies the number of pixels to offset.

**Prototype:**

```
virtual void WantReturn(BOOL yesNo)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

This method allows custom handling of the RETURN key. If you pass TRUE to this method an **EN\_CHANGE** message will be sent to the control when the RETURN key is pressed. The **EN\_CHANGE** message is sent when the user has taken any action that may have altered text in an edit control so developer need to also call **GotReturn()** (documented below) to see if it was indeed a RETURN keypress.

**Parameters:**

**BOOL yesNo**

If TRUE, then when the user presses the RETURN key in that control, the edit field will send an **EN\_CHANGE** message to the owner, and calling **GotReturn()** will return TRUE.

**Sample Code:**

Below is the way this is handled by the Hit By Name dialog. In that dialog, when the user enters a wild card pattern into the name match field and presses RETURN, the dialog is exited with the items matching the pattern selected. The way this is accomplished is by pass TRUE to **WantReturn()** and then processing the **EN\_CHANGE** message on the control. If **GotReturn()** is TRUE the Win32 function **PostMessage()** is used to send the **IDOK** message to exit the dialog. If this wasn't done, pressing RETURN in the edit control would only enter the text -- the user would have to move the mouse over the OK button and press it.

```
case IDC_HBN_PATTERN:  
if (HIWORD(wParam)==EN_CHANGE) {  
    iName = GetICustEdit( GetDlgItem(hDlg,IDC_HBN_PATTERN) );  
    iName->GetText(buf,256);  
    ct = _tcslen(buf);  
    if(ct && buf[ct-1] != _T('*'))  
        _tcscat(buf, _T("*"));  
    SendMessage(sbn->hList, LB_RESETCONTENT, 0, 0);
```

```
sbn->SetPattern(GetDlgItem(hDlg, IDC_HBN_PATTERN), buf);
sbn->BuildHitList(ct);
if(iName->GotReturn())
    PostMessage(hDlg, WM_COMMAND, IDOK, 0);
ReleaseICustEdit(iName);
}
break;
```

**Prototype:**

**virtual BOOL GotReturn()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method should be called on receipt of an **EN\_CHANGE** message. It return TRUE if pressing the RETURN key generated the message; otherwise FALSE.

**Prototype:**

**virtual void GiveFocus()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Calling this method gives the control the focus to receive input.

**Prototype:**

**virtual BOOL HasFocus()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if the control has the focus to receive input; otherwise FALSE.

**Prototype:**

**virtual void WantDlgNextCtl(BOOL yesNo)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Determines whether the TAB key may be used to jump to the next control in

the tab sequence.

**Parameters:**

**BOOL yesNo**

TRUE to enable the TAB key to move to the next control; FALSE to disable the TAB key from moving the focus.

**Prototype:**

**virtual void SetNotifyOnKillFocus(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Normally when a user exits an edit field the notification

**WM\_CUSTEDIT\_ENTER** is sent. Many plug-ins key off this message to finalize the input of values. For instance, if the user is entering a value into an edit field and they hit the TAB key to leave the field the value should be entered. Normally this is the desired behavior. However, as a special case condition, if a developer does not want to update the value, this method may be called so the **WM\_CUSTEDIT\_ENTER** notification won't be sent when the edit control loses focus.

**Parameters:**

**BOOL onOff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual void SetBold(BOOL onOff)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the text font in the edit control to display in a bold format or normal.

**Parameters:**

**BOOL onOff**

TRUE to turn bolding on; FALSE to turn off.

## Class ISpinnerControl

See Also: [Class ICustomControl](#), [Custom Controls](#).

```
class ISpinnerControl : public ICustomControl
```

### Description:

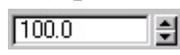
The spinner control is used (usually in conjunction with the custom edit control) to provide input of values limited to a fixed type. For example, the control may be limited to the input of only positive integers. The input options are integer, float, universe (world space coordinates), positive integer, positive float, positive universal, and time. This control allows the user to increment or decrement a value by clicking on the up or down arrows. The user may also click and drag on the arrows to interactively adjust the value. The Ctrl key may be held to accelerate the value changing speed, while the Alt key may be held to decrease the value changing speed.

The standard size used by 3ds max for the spinner control is 7 wide by 10 high. If you use a larger size, the spinner control arrows will be position in the upper left corner of the control.

**Important Note: The spinner control ensures that it only displays, and the user is only allowed to input, values within the specified ranges. However the spinner is just a front end to a controller which actually controls the value. The user can thus circumvent the spinner constraints by editing the controller directly (via function curves in track view, key info, etc.).**

**Therefore, when a plug-in gets a value from a controller (or a parameter block, which may use a controller) it is its responsibility to clamp the value to a valid range.**

 (Spinner Control)

 (Spinner and Edit control)

To initialize the pointer to the control call:

```
ISpinnerControl *GetISpinner(HWND hCtrl);
```

To release the control call:

```
ReleaseISpinner(ISpinnerControl *isc);
```

The value to use in the Class field of the Custom Control Properties dialog is:

**SpinnerControl**

The following messages may be sent by the spinner control:

This message is sent when the value of a spinner changes.

### **CC\_SPINNER\_CHANGE**

**lParam** contains a pointer to the spinner control. You can cast this pointer to a **ISpinnerControl** type and then call methods of the control.

**LOWORD(wParam)** contains the ID of the spinner. This is the named established in the ID field of the Custom Control Properties dialog.

**HIWORD(wParam)** is TRUE if the user is dragging the spinner interactively.

This message is sent when the user presses down on the spinner buttons.

### **CC\_SPINNER\_BUTTONDOWN**

**lParam** contains a pointer to the spinner control. You can cast this pointer to a **ISpinnerControl** type and then call methods of the control.

**LOWORD(wParam)** contains the ID of the spinner. This is the named established in the ID field of the Custom Control Properties dialog.

This message is sent when the user releases a spinner button.

### **CC\_SPINNER\_BUTTONUP**

**lParam** contains a pointer to the spinner control. You can cast this pointer to a **ISpinnerControl** type and then call methods of the control.

**LOWORD(wParam)** contains the ID of the spinner. This is the named established in the ID field of the Custom Control Properties dialog.

**HIWORD(wParam)** is FALSE if the user canceled and TRUE otherwise.

For example, if the user is interactively dragging the spinner, then does a right click to cancel, the following messages are sent:

- 1 A **CC\_SPINNER\_BUTTONDOWN** message indicating the user has pressed the spinner button.
- 2 A series of **CC\_SPINNER\_CHANGE** where **HIWORD(wParam) = TRUE**. This indicates that the spinner is being dragged interactively.
- 3 A **CC\_SPINNER\_CHANGE** where **HIWORD(wParam) = FALSE**.
- 4 A **CC\_SPINNER\_BUTTONUP** message where **HIWORD(wParam) = FALSE**. This indicates the user has cancelled.

**Methods:**

**Prototype:**

**virtual float GetFVal()=0;**

**Remarks:**

Returns the floating point value of the control.

**Prototype:**

**virtual int GetIVal()=0;**

**Remarks:**

This method returns the integer value of the control.

**Prototype:**

**virtual void SetAutoScale(BOOL on=TRUE)=0;**

**Remarks:**

This method sets the scale for the spinner based on the current value of the spinner. This allows the spinner to cover a larger range of values with less mouse motion. If you wish to use auto scale, pass TRUE to this method.

**Parameters:**

**BOOL on=TRUE**

If you wish to use auto scale pass TRUE to this method; otherwise FALSE.

**Prototype:**

**virtual void SetScale(float s)=0;**

**Remarks:**

This method sets the value which is added to or subtracted from the current control value as the arrow buttons are pressed, or the user interactively drags the spinner.

**Parameters:**

**float s**

The value is added to or subtracted from the current control value.

**Prototype:**

**virtual void SetValue(float v, int notify)=0;**

**Remarks:**

This method sets the value of the control to the specific floating point number passed. You may pass FALSE as the notify parameter so the control won't send a message when you set the value.

**Parameters:**

**float v**

The new value for the control.

**int notify**

If TRUE a message is sent indicating the control has changed.

Note that sometimes the **SetValue()** method is used to update the display of parameters in the user interface. For example, if the user changes the current time and the UI parameters are animated, the user interface controls must be updated to reflect the value at the new time. The programmer calls

**SetValue()** to update the value displayed in the control. This is an example of when to pass FALSE as the notify parameter. If you were to pass TRUE, a message would be sent as if the user had actually enter a new value at this time. These are of course very different conditions.

**Prototype:**

**virtual void SetValue(int v, int notify)=0;**

**Remarks:**

This method sets the value to the specific integer passed. You may pass FALSE as the notify parameter so the control won't send a message when you set the value.

**Parameters:**

**int v**

The new value for the control.

**int notify**

If TRUE a message is sent indicating the control has changed.

**Prototype:**

**virtual void SetLimits(int min, int max, int limitCurValue =**

**TRUE)=0;**

**Remarks:**

This method establishes the allowable limits for integer values entered.

**Parameters:**

**int min**

The minimum allowable value.

**int max**

The maximum allowable value.

**int limitCurValue = TRUE**

You may pass FALSE to the this parameter so the control will not send a spinner changed message when the limits are set.

**Prototype:**

**virtual void SetLimits(float min, float max, int limitCurValue = TRUE)=0;**

**Remarks:**

This method establishes the allowable limits for floating point values entered.

**Parameters:**

**float min**

The minimum allowable value.

**float max**

The maximum allowable value.

**int limitCurValue = TRUE**

You may pass FALSE to the this parameter so the control will not send a spinner changed message when the limits are set.

**Prototype:**

**virtual void LinkToEdit(HWND hEdit, EditSpinnerType type)=0;**

**Remarks:**

When an edit control is used in conjunction with the spinner control, this method is used to link the two, so values entered using the spinner are displayed in the edit control. This method is also used to set the type of value which may be entered.

**Parameters:****HWND hEdit**

The handle of the edit control to link.

**EditSpinnerType type**

The type of value that may be entered. One of the following values:

**EDITTYPE\_INT**

Any integer value.

**EDITTYPE\_FLOAT**

Any floating point value.

**EDITTYPE\_UNIVERSE**

This is a value in world space units. It respects the system's unit settings (for example feet and inches).

**EDITTYPE\_POS\_INT**

Any integer  $\geq 0$

**EDITTYPE\_POS\_FLOAT**

Any floating point value  $\geq 0.0$

**EDITTYPE\_POS\_UNIVERSE**

This is a positive value in world space units. It respects the system's unit settings (for example feet and inches) .

**EDITTYPE\_TIME**

This is a time value. It respects the system time settings (SMPTE for example).

**Prototype:**

```
virtual void SetIndeterminate(BOOL i=TRUE)=0;
```

**Remarks:**

This method is used to show commonality. When several different values are being reflected by the spinner, the value is indeterminate. When TRUE, the value field of the spinner appears empty.

**Parameters:****BOOL i=TRUE**

Pass TRUE to this method to set the value to indeterminate.

**Prototype:**

**virtual BOOL IsIndeterminate()=0;**

**Remarks:**

This method returns TRUE if the current state of the spinner is indeterminate. See **SetIndeterminate()** above.

**Prototype:**

**virtual void SetResetValue(float v)=0;**

**Remarks:**

A 3ds max user may right click on the spinner buttons to reset them to their 'reset' value (after they have been changed). This method specifies the value used when the reset occurs.

**Parameters:**

**float v**

The reset value.

**Prototype:**

**virtual void SetResetValue(int v)=0;**

**Remarks:**

A 3ds max user may right click on the spinner buttons to reset them to their 'reset' value (after they have been changed). This method specifies the value used when the reset occurs.

**Parameters:**

**int v**

The reset value.

**Prototype:**

**virtual void SetKeyBrackets(BOOL onOff)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the display of the brackets surrounding the spinner control to on. This is used to indicate if a key exists for the parameter controlled by the spinner at

the current time. These brackets turned on and off automatically if you are using a parameter map and parameter block to handle the control. If not you'll need to use this method.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Sample Code:**

This example shows how you do this if you only use a parameter block.

```
case CC_SPINNER_CHANGE:
switch (LOWORD(wParam)) {
case IDC_LENSPINNER:
th->SetLength(th->ip->GetTime(),th->lengthSpin->GetFVal());
th->lengthSpin->SetKeyBrackets(th->pblock->
KeyFrameAtTime(PB_LENGTH,th->ip->GetTime()));
break;
}
return TRUE;
```

The following functions are not part of class **ISpinnerControl** but are available for use with spinner controls.

**Function:**

**void SetSpinnerPrecision(int p);**

**Remarks:**

Sets the precision (number of decimal places displayed) used by the spinner control. Note that this function also affects slider controls. See [Class ISliderControl](#).

**Parameters:**

**int p**

The number of decimal places to display in the edit box linked to the spinner control.

**Function:**

**int GetSpinnerPrecision();**

**Remarks:**

Returns the number of decimal places displayed in the edit box linked to a spinner control. Note that this function also affects slider controls. See [Class ISliderControl](#).

Spinner controls have a global snap setting. This is set in 3ds max using File/Preferences... in the General page by changing the Spinner Snap setting. When enabled this specifies an increment that is applied to the current spinner value each time the UP or DOWN buttons are pressed on the spinner control.

**Prototype:**

**void SetSnapSpinner(BOOL b);**

**Remarks:**

This activates or de-activates the global spinner snap toggle.

**Parameters:**

**BOOL b**

TRUE to activate; FALSE to de-activate.

**Prototype:**

**BOOL GetSnapSpinner();**

**Remarks:**

Returns the global spinner snap setting; TRUE if on; FALSE if off.

**Prototype:**

**void SetSnapSpinValue(float f);**

**Remarks:**

This sets the global spinner snap increment or decrement value.

**Parameters:**

**float f**

The value that is added to or subtracted from the current spinner value when the arrow buttons are pressed.

**Prototype:**

**float GetSnapSpinValue();**

**Remarks:**

Returns the global spinner snap increment or decrement value.

**Function:**

**ISpinnerControl \*SetupFloatSpinner(HWND hwnd, int idSpin, int idEdit, float min, float max, float val, float scale = 0.1f);**

**Remarks:**

This global function (not part of class ISpinnerControl) is used for setting up Spinners. It performs the equivalent of the **GetISpinner()**, **SetLimits()**, **SetValue()**, and **LinkToEdit()**.

**Parameters:**

**HWND hwnd**

The handle of the dialog box in which the spinner appears.

**int idSpin**

The ID of the spinner.

**int idEdit**

The ID of the edit control.

**float min**

The minimum allowable value.

**float max**

The maximum allowable value.

**float val**

The initial value for the spinner.

**float scale = 0.1f**

The initial scale value for the spinner.

**Return Value:**

A pointer to the spinner control.

**Sample Code:**

Sample code to initialize a spinner / edit control.

```
to->custCtrlSpin = GetISpinner(GetDlgItem(hDlg,
IDC_SPIN_SPINNER));
to->custCtrlSpin->SetLimits(0.0f, 100.0f, FALSE);
to->custCtrlSpin->SetValue(100.0f, FALSE);
to->custCtrlSpin->LinkToEdit(GetDlgItem(hDlg,
```

```
IDC_SPIN_EDIT),  
    EDITTYPE_FLOAT);
```

The above code could be replaced with the following simplified code:

```
to->custCtrlSpin = SetupFloatSpinner(hDlg,  
    IDC_SPIN_SPINNER,  
    IDC_SPIN_EDIT, 0.0f, 100.0f, 100.0f);
```

**Function:**

```
ISpinnerControl *SetupIntSpinner(HWND hwnd, int idSpin, int  
idEdit, int min, int max, int val);
```

**Remarks:**

This global function (not part of class ISpinnerControl) is used for setting up Spinners. It performs the equivalent of the **GetISpinner()**, **SetLimits()**, **SetValue()**, and **LinkToEdit()**.

**Parameters:**

**HWND hwnd**

The handle of the dialog box in which the spinner appears.

**int idSpin**

The ID of the spinner.

**int idEdit**

The ID of the edit control.

**int min**

The minimum allowable value.

**int max**

The maximum allowable value.

**int val**

The initial value for the spinner.

**Return Value:**

A pointer to the spinner control.

**Function:**

```
void SetSpinDragNotify(BOOL onOff);
```

**Remarks:**

This global function is available in release 2.0 and later only.

This function controls whether or not spinners send

**CC\_SPINNER\_CHANGE** notifications while the user adjusts them with the mouse.

**Parameters:**

**BOOL onOff**

TRUE to turn on; FALSE to turn off.

**Function:**

**BOOL GetSpinDragNotify();**

**Remarks:**

This global function is available in release 2.0 and later only.

Returns TRUE if **CC\_SPINNER\_CHANGE** notifications are sent by spinners while the user adjusts them with the mouse; FALSE if they are not sent.

## Class ICustImage

See Also: [Class ICustomControl](#), [Custom Controls](#).

```
class ICustImage : public ICustomControl
```

### Description:

The custom image control provides a recessed area in the dialog to display a bitmap image.



To initialize the pointer to the control call:

### Prototype:

```
ICustImage *GetICustImage(HWND hCtrl);
```

To release the control call:

### Prototype:

```
ReleaseICustImage(ICustImage *ici);
```

The value to use in the Class field of the Custom Control Properties dialog is:

**CustImage**

### Methods:

### Prototype:

```
virtual void SetImage(HIMAGELIST hImage, int index, int w, int h)=0;
```

### Remarks:

This method sets the image to display.

### Parameters:

#### **HIMAGELIST hImage**

An image list. An image list is a collection of same-sized images, each of which can be referred to by its index. Image lists are used to efficiently manage large sets of icons or bitmaps in Windows. All images in an image list are contained in a single, wide bitmap in screen device format. An image list may also include a monochrome bitmap that contains masks used to draw

images transparently (icon style). The Windows API provides image list functions, which enable you to draw images, create and destroy image lists, add and remove images, replace images, and merge images.

**int index**

This is the index of the image to display in the image list.

**int w**

The image width.

**int h**

The image height.

## Class ICustStatus

See Also: [Class ICustomControl](#), [Custom Controls](#), [Class ICustStatusEdit](#).

class ICustStatus : public ICustomControl

### Description:

The custom status control provide a recessed area of the dialog which the developer may use as a status prompt display.



(0, 255, 128)

To initialize the pointer to the control call:

### Prototype:

**ICustStatus \*GetICustStatus(HWND hCtrl);**

To release the control call:

### Prototype:

**ReleaseICustStatus(ICustStatus \*ics);**

The value to use in the Class field of the Custom Control Properties dialog is:

**CustStatus**

### Methods:

#### Prototype:

**virtual void SetText(TCHAR \*text)=0;**

#### Remarks:

This method specifies the text message to display.

#### Parameters:

**TCHAR \*text**

Points to the text to display.

#### Prototype:

**virtual void GetText(TCHAR \*text, int ct)=0;**

#### Remarks:

This method is available in release 2.0 and later only.

Retrieves the text currently displayed in the custom status control.

**Parameters:**

**TCHAR \*text**

A pointer to storage for the text to return.

**int ct**

The maximum length of the string to return.

**Prototype:**

**virtual void SetTextFormat(StatusTextFormat f)=0;**

**Remarks:**

This methods controls the formatting of the text in the status control.

**Parameters:**

**StatusTextFormat f**

One of the following options:

**STATUSTEXT\_LEFT**

Left justified in the control.

**STATUSTEXT\_CENTERED**

Centered in the control.

**STATUSTEXT\_RIGHT**

Right justified in the control.

**Prototype:**

**virtual void SetTooltip(BOOL onOff, LPSTR text)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Specifies the tooltip text to use for a custom status control and enables or disables its availability.

**Parameters:**

**BOOL onOff**

If TRUE the tooltip text may appear; if FALSE the tooltip will not appear.

**LPSTR text**

The text to use for the tooltip.

## Class IColorSwatch

See Also: [Class ICustomControl](#), [Custom Controls](#), [COLORREF](#).

```
class IColorSwatch : public ICustomControl
```

### Description:

The color swatch control puts up the standard 3ds max modeless color selector when the control is clicked on. The plug-in may be notified as the user interactively selects new colors.



To initialize the pointer to the control call:

### Prototype:

```
IColorSwatch *GetIColorSwatch(HWND hCtrl, COLORREF col,  
TCHAR *name);
```

For example:

```
custCSw = GetIColorSwatch(GetDlgItem(hDlg,  
IDC_CSWATCH), RGB(255,255,255), _T("New Wireframe  
Color"));
```

This returns the pointer to the control, sets the initial color selected, and displays the text string passed in the title bar of the selection dialog.

To release the control call:

### Prototype:

```
ReleaseIColorSwatch(IColorSwatch *ics);
```

The value to use in the Class field of the Custom Control Properties dialog is:

### ColorSwatch

This message is sent as the color is being adjusted in the ColorPicker.

### CC\_COLOR\_CHANGE

**lParam** = pointer to ColorSwatchControl

**LOWORD(wParam)** contains the ID of the control. This is the named established in the ID field of the Custom Control Properties dialog.

**HIWORD(wParam)** contains 1 if button UP, or 0 if mouse drag.

This message is sent if the color has been clicked on, before bringing up the color picker.

## **CC\_COLOR\_SEL**

**lParam** contains a pointer to the ColorSwatch Control.

**LOWORD(wParam)** contains the ID of the control. This is the named established in the ID field of the Custom Control Properties dialog.

**HWORD(wParam)** contains 0.

This message is sent if another color swatch has been dragged and dropped on this swatch.

## **CC\_COLOR\_DROP**

**lParam** contains a pointer to the ColorSwatch Control.

**LOWORD(wParam)** contains the ID of the control. This is the named established in the ID field of the Custom Control Properties dialog.

**HWORD(wParam)** contains 0.

This message is sent when the color picker is closed.

## CC\_COLOR\_CLOSE

**lParam** contains a pointer to the ColorSwatch Control.

**LOWORD(wParam)** contains the ID of the control. This is the named established in the ID field of the Custom Control Properties dialog.

**HWORD(wParam)** contains 0.

### Methods:

#### Prototype:

```
virtual COLORREF SetColor(COLORREF c, int  
notify=FALSE)=0;
```

#### Remarks:

This method sets the current color value.

#### Parameters:

**COLORREF c**

You can pass specific RGB values in using the RGB macro. For example, to pass in pure blue you would use **RGB(0,0,255)**.

**int notify=FALSE**

If you pass TRUE for this parameter, the dialog proc for the dialog will receive the **CC\_COLOR\_CHANGE** message each time the color is changed.

#### Return Value:

This method returns the old color.

#### Prototype:

```
virtual COLORREF GetColor()=0;
```

#### Remarks:

This method may be used to retrieve the color selected by the user.

#### Return Value:

The **COLORREF** structure returned may be broken down into individual

RGB values by using the **GetRValue(color)**, **GetGValue(color)**, and **GetBValue(color)** macros.

**Prototype:**

**virtual void ForceDitherMode(BOOL onOff)=0;**

**Remarks:**

This method sets if the color shown in the color swatch is dithered on not.

**Parameters:**

**BOOL onOff**

TRUE to force the color to be dithered; otherwise FALSE.

**Prototype:**

**virtual void SetModal()=0;**

**Remarks:**

Call this method to have the color selector comes up in a modal dialog. This forces the user to select OK before the user may operate the rest of the program.

**Prototype:**

**virtual void Activate(int onOff)=0;**

**Remarks:**

This method is called to indicate that the color swatch is in a dialog that has been become active or inactive. A color swatch that is in an inactive dialog will be drawn as dithered due to the limited number of color registers available on an 8-bit display.

**Parameters:**

**int onOff**

If TRUE the color swatch is in an active dialog. If FALSE the control is in an inactive dialog.

**Prototype:**

**virtual void EditThis(BOOL startNew=TRUE)=0;**

**Remarks:**

If there is already a color picker up for a color swatch, this method switches it over to edit the color swatch on which **EditThis()** was called.

**Parameters:****BOOL startNew=TRUE**

If there was no color picker up, if this parameter is set to TRUE, then a color picker is created. If this parameter is set to FALSE, and there was no color picker up, then nothing happens.

## Class ICustToolBar

See Also: [Custom Controls](#). [Class ToolItem](#), [Class MacroButtonData](#), [Class CUIFrameMsgHandler](#), [Class ICustStatusEdit](#), [Class ICustStatus](#), [Class ICustButton](#).

```
class ICustToolBar : public ICustomControl
```

### Description:

This control allows the creation of toolbars containing buttons (push, check, and fly-offs), status fields, separators (spacers), and other Windows or user defined controls. Note: The standard size for 3ds max toolbar button icons is 16x15.

In 3ds max 3.0 and later toolbars may have multiple rows, or appear vertically. They may also have macro buttons (added with the MacroButtonData class) which may have icons or text.



To initialize the pointer to the control call:

### Prototype:

```
ICustToolBar *GetICustToolBar(HWND hCtrl);
```

To release the control call:

### Prototype:

```
ReleaseICustToolBar(ICustToolBar *ict);
```

The value to use in the Class field of the Custom Control Properties dialog is:

**CustToolBar**

Note: The **TB\_RIGHTCLICK** message is sent when the user right clicks in open space on a toolbar:

Also Note: To add tooltips to the toolbar controls you can do so by capturing the **WM\_NOTIFY** message in the dialog proc. For complete sample code see **\MAXSDK\SAMPLES\HOWTO\CUSTCTRL\CUSTCTRL.CPP**. The specific message is processed as shown below.

```
case WM_NOTIFY:
```

```
// This is where we provide the tooltip text for the
```

```
// toolbar buttons...
```

```
if(((LPNMHDR)lParam)->code == TTN_NEEDTEXT) {
```

```

LPTOOLTIPTEXT lpttt;
lpttt = (LPTOOLTIPTEXT)lParam;
switch (lpttt->hdr.idFrom) {
  case ID_TB_1:
    lpttt->lpszText = _T("Do Nothing Up");
    break;
  case ID_TB_2:
    lpttt->lpszText = _T("Do Nothing Down");
    break;
  case ID_TB_3:
    lpttt->lpszText = _T("Do Nothing Lock");
    break;
  case IDC_BUTTON1:
    if (to->custCtrlButtonC->IsChecked())
      lpttt->lpszText = _T("Button Checked");
    else
      lpttt->lpszText = _T("Button Un-Checked");
    break;
  };
}
break;

```

**Methods:**

**Prototype:**

**virtual void SetImage(HIMAGELIST hImage)=0;**

**Remarks:**

This method establishes the image list used to display images in the toolbar.

**Parameters:**

**HIMAGELIST hImage**

The image list. An image list is a collection of same-sized images, each of which can be referred to by an index. Image lists are used to efficiently manage large sets of icons or bitmaps in Windows. All images in an image list are contained in a single, wide bitmap in screen device format. An image list may also include a monochrome bitmap that contains masks used to draw

images transparently (icon style). The Windows API provides image list functions, which enable you to draw images, create and destroy image lists, add and remove images, replace images, and merge images.

**Prototype:**

**virtual void AddTool(const ToolItem& entry, int pos=-1)=0;**

**Remarks:**

The developer calls this method once for each item in the toolbar. The items appear in the toolbar from left to right in the order that they were added using this method. (Note that this method adds tools to the custom toolbar and not the 3ds max toolbar).

**Parameters:**

**const ToolItem& entry**

Describes the item to add to the toolbar.

**int pos=-1**

Controls where the added tool is inserted. The default of -1 indicates the control will be added at the right end of the toolbar.

**Prototype:**

**virtual void AddTool2(ToolItem& entry, int pos=-1)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Currently this method is identical to the AddTool() method.

**Parameters:**

**ToolItem& entry**

Describes the item to add to the toolbar.

**int pos=-1**

Controls where the added tool is inserted. The default of -1 indicates the control will be added at the right end of the toolbar.

**Prototype:**

**virtual void DeleteTools(int start, int num=-1)=0;**

**Remarks:**

This method is used to delete tools from the toolbar.

**Parameters:****int start**

Specifies which tool is the first to be deleted.

**int num=-1**

Specifies the number of tools to delete. If this parameter is -1 (the default) it deletes 'start' through count-1 tools.

**Prototype:**

**virtual void SetTopBorder(BOOL on)=0;**

**Remarks:**

Passing TRUE to this method draws a border above the toolbar. You can see the appearance of the top border in the sample toolbar shown above. If this is set to FALSE, the border is not drawn

**Parameters:****BOOL on**

TRUE to draw the border; FALSE for no border.

**Prototype:**

**virtual void SetBottomBorder(BOOL on)=0;**

**Remarks:**

Passing TRUE to this method draws a border beneath the toolbar. You can see the appearance of the bottom border in the sample toolbar shown above. If this is set to FALSE, the border is not drawn.

**Parameters:****BOOL on**

TRUE to draw the border; FALSE for no border.

**Prototype:**

**virtual int GetNeededWidth(int rows)=0;**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the width needed for specified number of rows.

**Parameters:**

**int rows**

The number of rows.

**Prototype:**

**virtual void SetNumRows(int rows)=0;**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the number of rows that the toolbar may hold.

**Parameters:**

**int rows**

The number of rows to set.

**Prototype:**

**virtual ICustButton \*GetICustButton(int id)=0;**

**Remarks:**

This method is used to return a pointer to one of the toolbar's buttons. Using this pointer you can call methods on the button. If you use this method, you must release the control after you are finished with it.

**Parameters:**

**int id**

Specifies the id of the toolbar button.

**Return Value:**

A pointer to one of the toolbar's buttons. If the button is not found it returns NULL. See [Class ICustButton](#).

**Prototype:**

**virtual ICustStatus \*GetICustStatus(int id)=0;**

**Remarks:**

This method is used to return a pointer to one of the toolbars status controls.

Using this pointer you can call methods on the status control. If you use this method, you must release the control after you are finished with it.

**Parameters:**

**int id**

Specifies the id of the toolbar button.

**Return Value:**

A pointer to one of the toolbars status controls. See [Class ICustStatus](#).

**Prototype:**

**virtual ICustStatusEdit \*GetICustStatusEdit(int id)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to return a pointer to the custom status edit control whose id is passed. If you use this method, you must release the control after you are finished with it. See [Class ICustStatusEdit](#).

**Parameters:**

**int id**

Specifies the id of the toolbar button.

**Prototype:**

**virtual HWND GetItemHwnd(int id)=0;**

**Remarks:**

Returns the handle to the toolbar item whose ID is passed.

**Parameters:**

**int id**

Specifies the id of the toolbar button.

**Prototype:**

**virtual int GetNumItems()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the number of items in the toolbar.

**Prototype:**

**virtual int GetItemID(int index)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Each item in the toolbar has an ID. When items are programatically added to the toolbar via Class `ToolButtonItem` an ID is passed to the `ToolButtonItem` constructor. This method returns the ID for the specified item in the toolbar.

**Parameters:**

**int index**

Specifies which toolbar item to return the id of. This is an index between **0** and **GetNumItems()-1**.

**Return Value:**

When the button is added using [Class ToolButtonItem](#) this is the id that is part of that structure. When the user operates a tool the dialog proc get a **WM\_COMMAND** message and this is also the id in **LOWORD(wParam)**.

**Prototype:**

**virtual int FindItem(int id)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the index into the list of toolbar entries of the item whose id is passed.

**Parameters:**

**int id**

The id of the control to find.

**Prototype:**

**virtual void DeleteItemByID(int id)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Deletes the toolbar item whose id is passed.

**Parameters:**

**int id**

The id of the control to delete.

**Prototype:**

```
virtual void LinkToCUIFrame(HWND hCUIFrame,  
CUIFrameMsgHandler *msgHandler)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

This method links this toolbar to the CUI frame whose window handle and message handler are passed.

**Parameters:**

**HWND hCUIFrame**

The window handle of the CUI frame to link this toolbar to.

**CUIFrameMsgHandler \*msgHandler**

Points to the message handler for the CUI frame. See [Class CUIFrameMsgHandler](#).

**Prototype:**

```
virtual void GetFloatingCUIFrameSize(SIZE *sz, int rows=1)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to obtain the size of the floating CUI frame.

**Parameters:**

**SIZE \*sz**

The size of the floating frame.

**int rows=1**

The number of rows displayed in the floating frame.

**Prototype:**

```
virtual void ResetIconImages() = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This resets the icons in the toolbar. This tells all the buttons in this toolbar to delete their icon image cache. If a plug-in has created a toolbar with any MaxBmpFileIcons on it, it should register a callback for color changing, and call this method on the toolbar. See [Structure NotifyInfo](#) for registering the color change callback.

## Class IRollupWindow

See Also: [Class ICustomControl](#), [Class IRollupPanel](#), [Custom Controls](#), [Class Interface](#).

```
class IRollupWindow : public ICustomControl
```

### Description:

This control is used to access existing rollup pages or if you are creating a dialog box which will not be used in the command panel. This control may be used to add a container area for rollup pages to be added to the dialog, and provides a scroll bar just like the command panel itself.

Note that this is a special case. Normally, adding rollup pages to the command panel is done using the simple **AddRollupPage()** method of the Interface class. This control is only used when you want to have a scrolling region for rollup pages in a dialog box.

To initialize the pointer to the control call:

```
IRollupWindow *GetIRollup(HWND hCtrl);
```

To release the control call:

```
void ReleaseIRollup(IRollupWindow *irw);
```

The value to use in the Class field of the Custom Control Properties dialog is:  
**RollupWindow**

### Methods:

#### Prototype:

```
virtual void Show()=0;
```

#### Remarks:

This causes all the rollup windows to be visible.

#### Prototype:

```
virtual void Hide()=0;
```

#### Remarks:

This causes all the rollup windows to become invisible.

#### Prototype:

**virtual void Show(int index)=0;**

**Remarks:**

This will make the rollup window whose index is passed visible.

**Parameters:**

**int index**

The index of the rollup to show.

**Prototype:**

**virtual void Hide(int index)=0;**

**Remarks:**

This will make the rollup window whose index is passed invisible.

**Parameters:**

**int index**

The index of the rollup to hide.

**Prototype:**

**virtual HWND GetPanelDlg(int index)=0;**

**Remarks:**

Returns the handle of the rollup page whose index is passed.

**Parameters:**

**int index**

The index of the rollup whose handle is to be returned.

**Prototype:**

**virtual int GetPanelIndex(HWND hWnd)=0;**

**Remarks:**

Returns an index to the rollup page given its handle.

**Parameters:**

**HWND hWnd**

The handle of the rollup.

**Prototype:**

**virtual void SetPanelTitle(int index,TCHAR \*title)=0;**

**Remarks:**

This method sets the title text displayed in the rollup page whose index is passed.

**Parameters:**

**int index**

Specifies the rollup whose title is to be set.

**TCHAR \*title**

The title string.

**Prototype:**

**virtual int AppendRollup(HINSTANCE hInst, TCHAR \*dlgTemplate, DLGPROC dlgProc, TCHAR \*title, LPARAM param=0,DWORD flags=0, int category = ROLLUP\_CAT\_STANDARD)=0;**

**Remarks:**

This method is used to add a rollup page.

**Parameters:**

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**TCHAR \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here.

**DWORD flags=0**

Append rollup page flags:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The index of the new page is returned.

**Prototype:**

```
virtual int AppendRollup(HINSTANCE hInst, DLGTEMPLATE *dlgTemplate, DLGPROC dlgProc, TCHAR *title, LPARAM param=0, DWORD flags=0, int category = ROLLUP_CAT_STANDARD)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is used to add a rollup page, but is currently not used.

**Parameters:**

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**DLGTEMPLATE \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here.

**DWORD flags=0**

Append rollup page flags:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The index of the new page is returned.

**Prototype:**

```
virtual int ReplaceRollup(int index, HINSTANCE hInst, TCHAR *dlgTemplate, DLGPROC dlgProc, TCHAR *title, LPARAM param=0,DWORD flags=0)=0;
```

**Remarks:**

This method is used to replace the rollup page whose index is passed.

**Parameters:****int index**

Specifies the rollup whose to be replaced.

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**TCHAR \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here.

**DWORD flags=0**

Append rollup page flags:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**Return Value:**

The index of the replacement page is returned.

**Prototype:**

```
virtual int ReplaceRollup(int index, HINSTANCE hInst,  
DLGTEMPLATE *dlgTemplate, DLGPROC dlgProc, TCHAR  
*title, LPARAM param=0,DWORD flags=0, int category =  
ROLLUP_CAT_STANDARD)=0;;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is used to replace the rollup page whose index is passed, but is currently not used.

**Parameters:****int index**

Specifies the rollup whose to be replaced.

**HINSTANCE hInst**

The DLL instance handle of the plug-in.

**DLGTEMPLATE \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here.

**DWORD flags=0**

Append rollup page flags:

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The index of the replacement page is returned.

**Prototype:**

**virtual void DeleteRollup(int index, int count)=0;**

**Remarks:**

This method deletes the rollup pages starting at the index passed. The count parameter controls how many pages are deleted.

**Parameters:****int index**

The starting index.

**int count**

The number of pages.

**Prototype:**

```
virtual void SetPageDlgHeight(int index,int height)=0;
```

**Remarks:**

This method is used to change the height of a rollup page.

**Parameters:****int index**

The index of the rollup to change.

**int height**

The new height of the dialog in pixels.

**Prototype:**

```
virtual void SaveState(RollupState *hState)=0;
```

**Remarks:**

This method saves the state of the rollup (the position of the scroll bars, which pages are open, etc...).

**Parameters:****RollupState \*hState**

Pointer to storage for the rollup state. Note: **typedef void \*RollupState;**

**Prototype:**

```
virtual void RestoreState(RollupState *hState)=0;
```

**Remarks:**

This methods restores a saved state.

**Parameters:**

**RollupState \*hState**

Pointer to storage for the rollup state. Note: **typedef void \*RollupState;**

**Prototype:**

**virtual int GetNumPanels()=0;**

**Remarks:**

This method returns the number of panels used in the rollup.

**Prototype:**

**virtual void DlgMouseMove(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)=0;**

**Remarks:**

Passing **WM\_LBUTTONDOWN**, **WM\_MOUSEMOVE**, and **WM\_LBUTTONUP** to this function allows hand cursor scrolling with unused areas in the dialog.

**Parameters:**

**HWND hDlg**

The handle of the dialog.

**UINT message**

The message to pass along: **WM\_LBUTTONDOWN**, **WM\_MOUSEMOVE**, or **WM\_LBUTTONUP**.

**WPARAM wParam**

**LPARAM lParam**

These are passed as part of the message sent in. Pass them along to this method.

**Prototype:**

**virtual BOOL IsPanelOpen(int index) = 0;**

**Remarks:**

This method return TRUE if the rollup page whose index is passed is open and FALSE if it is closed.

**Prototype:**

**virtual void SetPanelOpen(int index, BOOL isOpen, BOOL ignoreFlags = TRUE) =0;**

**Remarks:**

This causes the page whose index is passed to either open or close. If **isOpen** is passed a value of TRUE, the page is opened.

**Parameters:**

**int index**

The page to open or close.

**BOOL isOpen**

If TRUE, the page is opened, if FALSE it is closed.

**BOOL ignoreFlags = TRUE**

The method would close the panel if the **DONTAUTOCLOSE** flag is not set on the rollup. This flag indicates if it should be closed anyway, even if the flag is set.

**Prototype:**

**virtual int GetScrollPos()=0;**

**Remarks:**

This method returns the scroll position of the window.

**Prototype:**

**virtual void SetScrollPos(int spos)=0;**

**Remarks:**

This method sets the scroll position of the window.

**Parameters:**

**int spos**

The scroll position to set.

**Prototype:**

**virtual void MoveRollupPanelFrom(IRollupWindow \*from, HWND hPanel, BOOL top)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This methods moves a RollupPanel to another RollupWindow. It either inserts it at the top, or appends it at the end (depending on the top parameter)

**Parameters:****IRollupWindow \*from**

A pointer to the rollup window you are moving from.

**HWND hPanel**

The handle to the destination panel.

**BOOL top**

TRUE to insert at the top; FALSE to append at the end.

**Prototype:**

```
virtual int GetPanelHeight(int index)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns the height of the specified RollupPanel.

**Parameters:****int index**

The zero based index of the rollup panel.

**Prototype:**

```
virtual int GetScrollHeight()=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns the height of a RollupWindow, that it is longer than the visible area

**Prototype:**

```
virtual IRollupPanel *GetPanel(HWND hWnd)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the rollup panel for the specified window handle. An `IRollupPanel` describes the properties of a single rollup.

**Parameters:**

**HWND hWnd**

The window handle to get the rollup for.

**Prototype:**

**virtual void RegisterRollupCallback(IRollupCallback \*callb)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to register a rollup callback function to handle any custom handling for dragging and dropping rollouts.

**Parameters:**

**IRollupCallback \*callb**

A pointer to the callback function you wish to register.

**Prototype:**

**virtual void UnRegisterRollupCallback(IRollupCallback \*callb)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to unregister a rollup callback function.

**Parameters:**

**IRollupCallback \*callb**

A pointer to the callback function you wish to unregister.

**Prototype:**

**virtual void RegisterRCMenuItem(IRollupRCMenuItem \*item)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to register a rollup right-click menu item which will

be added to the list of items. For rollups that support Drag and Drop this is used to register a ResetCategories RightClickMenu. Reset Categories will get rid of all the changes that have been made through drag and drop and restore the default.

**Parameters:**

**IRollupRCMenuItem \*item**

A pointer to the right-click menu item you wish to register.

**Prototype:**

**virtual void UnRegisterRCMenuItem(IRollupRCMenuItem \*item)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to unregister a rollup right-click menu item.

**Parameters:**

**IRollupRCMenuItem \*item**

A pointer to the right-click menu item you wish to unregister.

**Prototype:**

**virtual void ResetCategories(bool update = true)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method will reset the category information on all the panels in the rollup window. The plugin will have to be reloaded (EndEditParams, BeginEditparams) in order to show this in the UI.

**Parameters:**

**bool update = true**

TRUE to update the layout, otherwise FALSE. Leave this on TRUE.

**Prototype:**

**virtual void UpdateLayout()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used internally

**This function is not part of this class but is available for use:**

**Function:**

**BOOL IsRollupPanelOpen(HWND hDlg);**

**Remarks:**

This function returns TRUE if a particular rollup panel is open given a handle to the dialog window in the panel.

**Parameters:**

**HWND hDlg**

Handle to the dialog window in the panel.

## Class IOffScreenBuf

See Also: [Custom Controls](#).

class IOffScreenBuf

### Description:

This control provides an off screen buffer which the developer may draw into, then quickly blit onto the actual display for flicker free image updates.

To initialize the pointer to the control call:

### Function:

**IOffScreenBuf \*CreateIOffScreenBuf(HWND hCtrl);**

To release the control call:

### Function:

**DestroyIOffScreenBuf(IOffScreenBuf \*iBuf);**

### Methods:

### Prototype:

**virtual HDC GetDC()=0;**

### Remarks:

Returns a handle to the display device context (DC) for the off screen buffer. The display device context can be used in subsequent GDI functions to draw in the buffer.

### Prototype:

**virtual void Erase(Rect \*rct=NULL)=0;**

### Remarks:

This method is used to erase the buffer.

### Parameters:

**Rect \*rct=NULL**

Specifies the rectangular region to erase. If NULL the entire buffer is erased.

### Prototype:

**virtual void Blit(Rect \*rct=NULL)=0;**

**Remarks:**

This method blits (transfers the image from) the buffer to the display.

**Parameters:**

**Rect \*rct=NULL**

Specifies the rectangular region to blit. If NULL the entire buffer is blitted.

**Prototype:**

**virtual void Resize()=0;**

**Remarks:**

This method is used to resize the buffer.

**Prototype:**

**virtual void SetBkColor(COLORREF color)=0;**

**Remarks:**

This sets the buffer to the specified color.

**Parameters:**

**COLORREF color**

The color to set. You may use the RGB macro to set the color.

**Prototype:**

**virtual COLORREF GetBkColor()=0;**

**Remarks:**

This methods retrieves the background color of the buffer.

**Return Value:**

The background color of the buffer.

# Class BitArray

See Also: [Template Class Tab](#), [Class BitArrayCallback](#).

Class BitArray

## Description:

This class allows the developer to define a set of bit flags that may be treated as a virtual array and are stored in an efficient manner. The class has methods to set, clear and return the i-th bit, resize the BitArray, etc. All methods are implemented by the system.

## Methods:

### Prototype:

**BitArray()**

### Remarks:

Default constructor. Sets the number of bits to 0.

### Prototype:

**BitArray(int n);**

### Remarks:

Constructor.

### Parameters:

**int i**

The size of the BitArray in bits.

### Prototype:

**BitArray(const BitArray& b);**

### Remarks:

Constructor. Duplicates the BitArray passed.

### Parameters:

**const BitArray& b**

The BitArray to duplicate.

**Prototype:**

**void SetSize(int n, int save=0)**

**Remarks:**

Sets the number of bits used.

**Parameters:**

**int n**

The number of bits in the array.

**int save=0**

If passed as 1, the old bit values will be preserved when the array is resized.

**Prototype:**

**int GetSize()**

**Remarks:**

Returns the size of the bit array in bits.

**Prototype:**

**void ClearAll()**

**Remarks:**

Clears all the bits in the array (sets them to 0).

**Prototype:**

**void SetAll()**

**Remarks:**

Sets all the bits in the array to 1.

**Prototype:**

**void Set(int i)**

**Remarks:**

Set the i-th bit to 1.

**Parameters:**

**int i**

The array index of the bit to set.

**Prototype:**

**void Clear(int i)**

**Remarks:**

Sets the i-th bit to 0.

**Parameters:**

**int i**

The array index of the bit to clear.

**Prototype:**

**void Set(int i, int b);**

**Remarks:**

Set the i-th bit to b.

**Parameters:**

**int i**

The index of the bit to set.

**int b**

The value to set, either 1 or 0.

**Prototype:**

**int NumberSet()**

**Remarks:**

Returns the number of bits set to 1.

**Return Value:**

The number of bits set to 1.

**Prototype:**

**BOOL IsEmpty();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if no bits are set; otherwise FALSE. This method is much faster than checking if **NumberSet()** returns 0.

**Prototype:**

**void Compress()**

**Remarks:**

This is not currently implemented and is reserved for future use.

**Prototype:**

**void Expand()**

**Remarks:**

This is not currently implemented and is reserved for future use.

**Prototype:**

**void Reverse(BOOL keepZero = FALSE);**

**Remarks:**

This method is available in release 2.0 and later only.

Reverses the bits in the BitArray.

**Parameters:**

**BOOL keepZero = FALSE**

If TRUE the zero bit is kept where it is.

**Prototype:**

**void Rotate(int direction, int count);**

**Remarks:**

This method is available in release 2.0 and later only.

Rotates the bits in the BitArray (with wraparound).

**Parameters:**

**int direction**

The direction to rotate.

**int count**

The number of bits to rotate.

**Prototype:**

**void Shift(int direction, int count, int where=0);**

**Remarks:**

This method is available in release 2.0 and later only.

Shifts the bits in the BitArray (without wraparound).

**Parameters:**

**int direction**

One of the following values:

**LEFT\_BITSHIFT**

**RIGHT\_BITSHIFT**

**int count**

The number of bits to shift.

**int where=0**

This indicates where the shift will begin. For example, if you have a

**BitArray** containing: **10101010**

and you **Shift(LEFT\_BITSHIFT, 1, 4)** you'll get: **10100100**

All the bits from 4 to 8 are shifted one bit left, with zeroes shifted in from the right. The first bit affected is the **where** bit. If you leave off the **where** parameter you'd get the usual: **01010100**

The **RIGHT\_BITSHIFT** starts at that bit; it is unaffected because the operation proceeds to the right: **10101010**.

**Shift(RIGHT\_BITSHIFT, 1, 4)** results in: **10101101**.

**Prototype:**

**void EnumSet(BitArrayCallback &cb);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used to enumerate all the elements that have a "1" value, and call the callback **proc()** with the index of the element.

**Parameters:****BitArrayCallback &cb**

The callback object whose **proc()** method is called.

**Prototype:**

```
void DeleteSet(BitArray & dset, int mult=1);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method allows you to delete a selection of elements from this BitArray.

This is useful, for instance, if you're deleting a set of vertices from a mesh and wish to keep the vertSel and vertHide arrays up to date.

**Parameters:****BitArray & dset**

This is a bit array which represents which elements should be deleted.

Typically (if mult==1) dset will have the same size as (this).

**int mult=1**

This is a multiplier which indicates how many elements in (\*this) are deleted for each entry in dset. For instance, when deleting faces in a mesh, you also need to delete the corresponding edge selection data. Since edgeSel[f\*3], edgeSel[f\*3+1], and edgeSel[f\*3+2] correspond to face f, you'd use mult=3:

```
faceSel.DeleteSet (fdel);
```

```
edgeSel.DeleteSet (fdel, 3);
```

**Prototype:**

```
IOResult Save(ISave* isave);
```

**Remarks:**

Saves the BitArray to the 3ds max file.

**Prototype:**

```
IOResult Load(ILoad* iload);
```

**Remarks:**

Loads the BitArray from the 3ds max file.

**Operators:**

**Prototype:**

**int operator[](int i) const;**

**Remarks:**

Gets the i-th bit.

**Parameters:**

**int i**

The index of the bit.

**Prototype:**

**BOOL operator==(const BitArray& b);**

**Remarks:**

This operator is available in release 3.0 and later only.

Comparison operator.

**Parameters:**

**const BitArray& b**

The BitArray to compare with this one.

**Return Value:**

TRUE if the BitArrays are 'equal' (same size and same bits set); otherwise

FALSE.

**Assignment operators: These require arrays of the same size!**

**Prototype:**

**BitArray& operator=(const BitArray& b)**

**Remarks:**

Assignment operator.

**Prototype:**

**BitArray& operator&=(const BitArray& b)**

**Remarks:**

AND= this BitArray with the specified BitArray.

**Prototype:**

**BitArray& operator|=(const BitArray& b)**

**Remarks:**

OR= this BitArray with the specified BitArray.

**Prototype:**

**BitArray& operator^=(const BitArray& b)**

**Remarks:**

XOR= this BitArray with the specified BitArray.

**Binary operators: These require arrays of the same size!**

**Prototype:**

**BitArray operator&(const BitArray&) const**

**Remarks:**

AND two BitArrays

**Prototype:**

**BitArray operator|(const BitArray&) const**

**Remarks:**

OR two BitArrays

**Prototype:**

**BitArray operator^(const BitArray&) const**

**Remarks:**

XOR two BitArrays

**Unary operators**

**Prototype:**

**BitArray operator~()**

**Remarks:**

Unary NOT function

## Class Class\_ID

See Also: [Dll Functions and Class Descriptors](#), [List of Class\\_IDs](#), [Generate a Class\\_ID](#).

class Class\_ID

### **Description:**

This class represents the unique class ID for a 3ds max plug-in. A plug-ins Class\_ID must be **unique**. A program is provided with the SDK to generate these ClassIDs. It is VERY important you use this program to create the ClassIDs for your plug-ins. To generate a random Class\_ID and optionally copy it to the clipboard, click [Generate a Class\\_ID](#). A Class\_ID consists of two unsigned 32-bit quantities. The constructor assigns a value to each of these, for example **Class\_ID(0xCAD834E2, 0x27E47C5A)**.

All the methods of this class are implemented by the system.

### **Important Notes:**

Make sure you use the program provided to create your ClassIDs. This will greatly reduce the likelihood of conflicts between plug-ins.

If you use one of the 3ds max source code examples to create your plug-in, you **MUST** change the existing Class\_ID. If you don't, you'll get a conflict. If two ClassIDs conflict, the system will only load the first one it finds. The system will post a message when it attempts to load the second one noting that there is a Class\_ID conflict.

The sample code plug-ins used in 3ds max use 0 as the second 32-bit quantity of the Class\_ID. Only the built-in classes (those that ship with 3ds max) should have the second 32 bits equal to 0. All plug-in developers should use both 32 bit quantities.

### **Methods:**

#### **Prototype:**

**Class\_ID(ulong aa, ulong bb)**

#### **Remarks:**

Constructor.

This is the standard constructor to be used by 3ds max plug-ins. Each of the 32-bit quantities may be assigned separately.

#### **Parameters:**

**ulong aa**

Assigned to the first 32-bit quantity.

**ulong bb**

Assigned to the second 32-bit quantity.

#### **Prototype:**

**Class\_ID()**

#### **Remarks:**

Constructor.

Assigns a value of **0xFFFFFFFF** to each 32-bit quantity.

#### **Prototype:**

**Class\_ID(const Class\_ID& cid)**

**Remarks:**

Constructor.

Creates a new class ID with the same values as the argument.

**Parameters:**

**const Class\_ID& cid**

A reference to the Class ID to copy.

**Prototype:**

**ULONG PartA()**

**Remarks:**

Returns the first unsigned 32-bit quantity.

**Prototype:**

**ULONG PartB()**

**Remarks:**

Returns the second unsigned 32-bit quantity.

**Operators:**

**Prototype:**

**int operator==(const Class\_ID& cid) const**

**Remarks:**

Checks for equality between two Class IDs.

**Prototype:**

**int operator!=(const Class\_ID& cid) const**

**Remarks:**

Check for Inequality between two Class IDs.

**Prototype:**

**Class\_ID& operator=(const Class\_ID& cid)**

**Remarks:**

Assignment operator. Note: In release 3.0 and later this method checks for self-assignment.

**Prototype:**

**bool operator<(const Class\_ID& rhs) const;**

**Remarks:**

This operator is available in release 4.0 and later only.

Less than operator. This returns true if the specified Class\_ID's two parts are numerically less than this Class\_ID's; false if not.

## Class ClassDesc

See Also: [Class ClassDesc2](#), [Class IParamBlock2](#), [Class ParamBlockDesc2](#), [Class ILoad](#), [Class ISave](#), [Class Interface](#), [Class Class\\_ID](#).

class ClassDesc

### Description:

Class descriptors provide the system with information about the plug-in classes in the DLL. The developer creates a class descriptor by deriving a class from **ClassDesc** and implementing several of its methods.

In release 3.0 and later there are new methods which are supplied and implemented by ClassDesc2. These methods relate to the parameter block2 system.

In release 4.0 plug-ins wishing to use the new Function Publishing system must use **ClassDesc2** rather than this class for their class descriptors. See [Function Publishing System](#).

### Methods Groups:

The hyperlinks below take you to the start of groups of related methods within the class:

[Creation Related Methods](#)

[ClassID / SuperClassID / ClassName, Category Methods](#)

[Class Parameter Related Methods](#)

[Action Table Related Methods](#)

[Manipulator Related Methods](#)

[ParamBlock2 Related Methods](#)

[Function Publishing Related Methods](#)

[Schematic View Related Methods](#)

[Generic Expansion Function](#)

### Methods:

#### Prototype:

```
virtual int IsPublic()=0;
```

#### Remarks:

Implemented by the Plug-In.

Controls if the plug-in shows up in lists from the user to choose from.

### **Return Value:**

If the plug-in can be picked and assigned by the user, as is usually the case, return TRUE. Certain plug-ins may be used privately by other plug-ins implemented in the same DLL and should not appear in lists for user to choose from. These plug-ins would return FALSE.

## **Creation Related Methods**

### **Prototype:**

**virtual void \*Create(BOOL loading=FALSE)=0;**

### **Remarks:**

Implemented by the Plug-In.

3ds max calls this method when it needs a pointer to a new instance of the plug-in class. For example, if 3ds max is loading a file from disk containing a previously used plug-in (procedural object, modifier, controller, etc...), it will call the plug-in's **Create()** method. The plug-in responds by allocating a new instance of its plug-in class. See the Advanced Topic section on [Memory Allocation](#) for more details.

### **Parameters:**

**BOOL loading=FALSE**

This parameter is a flag indicating if the class being created is going to be loaded from a disk file. If the flag is TRUE, the plug-in may not have to perform any initialization of the object because the loading process will take care of it. See the Advanced Topics section on [Loading and Saving](#) for more information.

Note: If this parameter is TRUE, developers must initialize their references to NULL. Otherwise 3ds max may crash.

3ds max provides a default plug-in object creation process. Many plug-ins fit this form. When the system is about to create an instance of the plug-in object it calls a method **BaseObject::GetCreateMouseCallback()**. This method returns a callback object whose **proc()** method handles the mouse input during its creation phase. Most of the work is then handled by the system. The procedural sphere is an example of this type of plug-in. Certain plug-ins may have special creation needs however. The target camera is an example of such a plug-in. Because it needs to create two nodes in the scene (the camera and the

target) it requires a custom creation process. To support these plug-ins the following two methods are provided. They allow the plug-in to manage the creation process themselves. See the Advanced Topics section on [Object Creation Methods](#) for more details.

**Prototype:**

**virtual int BeginCreate(Interface \*i)**

**Remarks:**

Implemented by the Plug-In.

The custom creation process of the plug-in object is handled by this method. For example, a plug-in can create a custom command mode and push it on the command stack to handle the creation process.

Important Note: A plug-in that doesn't want to participate in the standard object creation mechanism using CreateMouseCallBack **must** push a CommandMode on the stack in this method and remove it in **EndCreate()**. This is true even if the plug-in doesn't do anything inside the mode. A mode has to be pushed on the stack and then later popped off otherwise a crash will occur (if the default implementation of this method is not used). For more details on object creation see the Advanced Topics section [Object Creation Methods](#).

**Parameters:**

**Interface \*i**

An interface pointer the plug-in may use to call functions in 3ds max.

**Return Value:**

To use the default creation process (the system implementation for this method) return 0; Return nonzero if the plug-in implements this method.

**Default Implementation:**

**{ return 0; }**

**Prototype:**

**virtual int EndCreate(Interface \*i)**

**Remarks:**

Implemented by the Plug-In.

The termination of the custom creation process is managed by the implementation of this method. For example, the plug-in could remove a custom command mode from the command stack. See the Advanced Topics section on [Object Creation Methods](#) for more details.

**Parameters:**

**Interface \*i**

An interface pointer the plug-in may use to call functions in 3ds max.

**Return Value:**

To use the system implementation for this method return 0; Return nonzero if the plug-in implements this method.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual BOOL OkToCreate(Interface *i)
```

**Remarks:**

Implemented by the Plug-In.

This method is used to enable or disable the button that allows the plug-ins class to be created. For example, at certain times it is not appropriate for the Boolean object to be created. When there is not an object of the appropriate type selected the Boolean object cannot be created. At these times the button should be disabled (the button will appear as grayed out in the Create branch of the command panel). The button should be enabled if there is an object of the appropriate type selected. This method allows a plug-in to control the state of the button.

**Parameters:**

**Interface \*i**

An interface pointer the plug-in may use to call functions in 3ds max.

**Return Value:**

TRUE to enable the class creation button; FALSE to disable it.

**Default Implementation:**

```
{ return TRUE; }
```

**Sample Code:**

The following code from `\MAXSDK\SAMPLES\OBJECTS\BOOLOBJ.CPP` demonstrates an implementation of this method. If there is not a node selected, it is not OK to use the command so the button should appear disabled. To disable the button **OkToCreate()** returns FALSE. If the object that is selected is not of the appropriate type it the button is disabled as well.

```
BOOL BoolObjClassDesc::OkToCreate(Interface *i)  
{  
  if (i->GetSelNodeCount()!=1) return FALSE;  
  ObjectState os = i->GetSelNode(0)->GetObjectRef()->Eval(i-  
>GetTime());  
  if (os.obj->SuperClassID()!=GEOMOBJECT_CLASS_ID) {  
    return FALSE;  
  }  
  return TRUE;  
}
```

## **ClassName, ClassID, SuperClass ID, Category Methods**

### **Prototype:**

```
virtual const TCHAR* ClassName()=0;
```

### **Remarks:**

Implemented by the Plug-In.

This method returns the name of the class. This name appears in the button for the plug-in in the 3ds max user interface.

### **Return Value:**

The name of the class.

### **Prototype:**

```
virtual SClass_ID SuperClassID()=0;
```

### **Remarks:**

Implemented by the Plug-In.

This method returns a system defined constant describing the class this plug-in class was derived from. For example, the Bend modifier returns **OSM\_CLASS\_ID**. This super class ID is used by all object space modifiers.

See [List of SuperClassIDs](#).

**Return Value:**

The SuperClassID of the plug-in.

**Prototype:**

**virtual Class\_ID ClassID()=0;**

**Remarks:**

Implemented by the Plug-In.

This method must return the **unique** ID for the object. If two ClassIDs conflict, the system will only load the first one it finds. The ClassID consists of two unsigned 32-bit quantities. The constructor assigns a value to each of these, for example **Class\_ID(0xA1C8E1D1, 0xE7AA2BE5)**. A developer should use the random Class\_ID generator to avoid conflicts ([Generate a random Class\\_ID](#)). See [Class Class\\_ID](#) for more information.

**Return Value:**

The unique ClassID of the plug-in.

**Prototype:**

**virtual Class\_ID SubClassID();**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

This method can be used for further categorizing plugin's. If a plugin has sub-plugins (like light > shadows, particles > operators), this method can be used to differentiate them. sub-plugins can be derived off reference target but return a particular class ID published by the parent plugins SDK headers. Then parent plugin can get a list of all reference targets whose SubClassID matches the published SubClassID

**Default Implementation:**

**{ return Class\_ID(); }**

**Prototype:**

**virtual const TCHAR\* Category()=0;**

## Remarks:

Implemented by the Plug-In.

This methods returns a string describing the category a plug-in fits into. The category is usually selected in the drop down list in the create, or utility branch of the command panel. In the create branch, if this is set to be an existing category (i.e. "Standard Primitives", "Splines", ...) then the plug-in will appear in that category. If the category doesn't yet exists then it is created. If the plug-in does not need to appear in the list, it may simply return a null string as in `_T("")`.

In the modify branch, the category determines which group it appears in the Configure Button Sets / Modifiers list. These are the categories such as "MAX STANDARD", "MAX EDIT", and "MAX SURFACE".

This method is also used to distinguish between the various types of texture maps so they can be separated in the Material/Map Browser. The appropriate string should be returned by this method of the Texmap. For example:

```
const TCHAR* Category() { return TEXMAP_CAT_3D; }
```

The options for texture maps are:

```
TCHAR TEXMAP_CAT_2D[]; - 2D maps.
```

```
TCHAR TEXMAP_CAT_3D[]; - 3D maps.
```

```
TCHAR TEXMAP_CAT_COMP[]; - Composite.
```

```
TCHAR TEXMAP_CAT_COLMOD[]; - Color modifier.
```

```
TCHAR TEXMAP_CAT_ENV[]; - Environment.
```

## Class Parameter Related Methods

The following three methods deal with default settings for plug-in classes. Most plug-in do not need to be concerned with these methods.

In the 3ds max user interface, from the Files / Preferences... menu on the Animation page there is an option for Controller Defaults. There are buttons for 'Set Defaults...' and 'Restore to Factory Settings...'. When the user presses the 'Set Defaults...' button the user is presented with a list of plug-ins that have responded TRUE to the **HasClassParams()** method. These plug-ins have default parameters that the user can edit. These are the defaults used when a new instance of the plug-in class is created. When the user picks an item from the list, its **EditClassParams()** method is called to allow the plug-in to put up a modal dialog to let the user edit the default parameters. If the user presses the 'Reset to

Factory Defaults...' button, the **ResetClassParams()** method is called. The plug-in can then be reset to use any default settings that it has.

**Prototype:**

**virtual BOOL HasClassParams()**

**Remarks:**

Implemented by the Plug-In.

If a plug-in class has default parameters that it needs to allow the user to edit, TRUE should be returned and **EditClassParams()** and **ResetClassParams()** should be implemented. Otherwise return FALSE (the default). See the description above.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

**virtual void EditClassParams(HWND hParent);**

**Remarks:**

Implemented by the Plug-In.

If the user picks the class from the list this method is called. The plug-in should put up a modal dialog that allows the user to edit the plug-ins default parameters. The plug-in should not return until the user has finished editing the parameters. See the description above.

**Parameters:**

**HWND hParent**

The parent window handle.

**Prototype:**

**virtual void ResetClassParams(BOOL fileReset=FALSE);**

**Remarks:**

Implemented by the Plug-In.

When the user executes File / Reset or presses the 'Reset to Factory Settings...' button in the File / Preferences... / Animation tab / Controller Defaults section

this method is called. The plug-in can respond by resetting itself to use its default values. See the description above.

**Parameters:**

**BOOL fileReset=FALSE**

When TRUE, the user has performed a File / Reset operation. When FALSE, the user is in the Preferences... dialog doing a reset controller defaults operation.

## Action Table Related Methods

These two functions return keyboard action tables that plug-ins can use

**Prototype:**

**virtual int NumActionTables();**

**Remarks:**

This method is available in release 4.0 and later only.

This method is called at Dll-load time to get the number of action tables from a plug-in. Note: If more than one class uses the table only one of the classes should export the table, but they can all use them. See [Class ActionTable](#).

**Default Implementation:**

**{ return 0; }**

**Prototype:**

**virtual ActionTable\* GetActionTable(int i);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the 'i-th' action table. See [Class ActionTable](#).

**Parameters:**

**int i**

The zero based index of the table to return.

**Default Implementation:**

**{ return NULL; }**

## Manipulator Related Methods

The following methods support manipulators. These methods are called on the class descriptors of manipulators when manipulate mode is entered, or when selection changes while in manipulate mode.

### Prototype:

**virtual BOOL UseSelectionBrackets();**

### Remarks:

This method is available in release 4.0 and later only.

This method allows an object to choose whether or not it will display selection brackets in shaded viewports. The method will return FALSE if no selection brackets are displayed or TRUE if it does display selection brackets.

### Default Implementation:

**{ return TRUE; }**

### Prototype:

**virtual BOOL IsManipulator();**

### Remarks:

This method is available in release 4.0 and later only.

This method is what is used by the system to determine whether the ClassDesc describes a manipulator. Returns TRUE if the class implements a manipulator object; otherwise FALSE.

### Default Implementation:

**{ return FALSE; }**

### Prototype:

**virtual BOOL CanManipulate(ReferenceTarget\* hTarget);**

### Remarks:

This method is available in release 4.0 and later only.

The method returns true if the class is a manipulator and it manipulates the given base object, modifier or controller. When starting "Manipulate" mode, this is called on selected nodes for the base object, all modifiers, the TM controller and the position, rotation and scale controllers, if the TM controller

is a PRSController.

**Parameters:**

**ReferenceTarget\* hTarget**

A pointer to a reference target.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual BOOL CanManipulateNode(INode* pNode);
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns TRUE if the manipulator applies to the given node; otherwise FALSE.

This is a general case if **CanManipulateClassID()** isn't sufficient.

**Parameters:**

**INode\* pNode**

The node to check.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void* CreateObjectManipulator(Object* pObject);
```

**Remarks:**

This method is available in release 4.0 and later only.

When a manipulator returns TRUE to **CanManipulateClassID()**, the system calls this method to create an instance and return a pointer to it.

**Parameters:**

**Object\* pObject**

Points to the Object that the manipulator said it could manipulate.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual void* CreateTMControlManipulator(Control* pControl);
```

**Remarks:**

This method is available in release 4.0 and later only.

When a manipulator returns TRUE to **CanManipulateClassID()**, the system calls this method to create an instance and return a pointer to it.

**Parameters:**

**Control\* pControl**

Points to the Controller that the manipulator said it could manipulate.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual void* CreateModifierManipulator(Modifier* pModifier);
```

**Remarks:**

This method is available in release 4.0 and later only.

When a manipulator returns TRUE to **CanManipulateClassID()**, the system calls this method to create an instance and return a pointer to it.

**Parameters:**

**Modifier\* pModifier**

Points to the Modifier that the manipulator said it could manipulate.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual void* CreateManipulator(ReferenceTarget* hTarget,  
INode* pNode);
```

**Remarks:**

This method is available in release 4.0 and later only.

When a manipulator returns TRUE to **CanManipulateNode(INode\*)**, the system calls this version of **CreateManipulator()** to create an instance.

**Parameters:**

**ReferenceTarget\* hTarget**

Points to the reference target.

**INode\* pNode**

Points to the node that the manipulator said it could manipulate.

**Return Value:**

```
{ return NULL; }
```

## **Load / Save Related Methods**

The following three methods may be used to save data associated with a class in a 3ds max file. If you want to save data associated with the class have **NeedsToSave()** return TRUE and implement the **Save()** and **Load()** methods.

**Prototype:**

```
virtual BOOL NeedsToSave();
```

**Remarks:**

Implemented by the Plug-In.

Returns TRUE if there is data associated with the class that needs to be saved in the 3ds max file. If this is so, implement the **Save()** and **Load()** methods below. If there is no class data to save return FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual IOResult Save(ISave *is);
```

**Remarks:**

Implemented by the Plug-In.

If **NeedsToSave()** returns TRUE then this method should be implemented to save the data associated with the class.

**Parameters:**

**ISave \*is**

A pointer that may be used to call methods to save data to disk.

**Return Value:**

**IO\_OK** if the save was successful; otherwise **IO\_ERROR**.

**Prototype:**

**virtual IOResult Load(ILoad \*il);**

**Remarks:**

Implemented by the Plug-In.

If **NeedsToSave()** returns TRUE then this method should be implemented to load the data associated with the class.

**Parameters:**

**ILoad \*il**

A pointer that may be used to load data from a file.

**Return Value:**

**IO\_OK** if the load was successful; otherwise **IO\_ERROR**.

## **Parameter Map 2 Related Methods**

**Prototype:**

**virtual DWORD InitialRollupPageState();**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns a DWORD which is used to initialize the rollup state in both the create branch and the modify branch. The semantics are different, however for these two cases. Whenever the rollups are created in the create branch, their state will be that specified by this method. In the modify branch, the first time an object of this type is modified the state will be that of this method, but after that it will remain what it was last set to.

**Return Value:**

The bits of this DWORD set indicate the corresponding rollup page is closed. The zero bit corresponds to the plug-ins first rollup, the first bit is the second rollup, etc. The value **0x7fffffff** is returned by the default implementation so the command panel can detect this method is not being overridden, and just

leave the rollups as is.

**Default Implementation:**

```
{ return 0x7fffffff; }
```

**Prototype:**

```
virtual const TCHAR* InternalName();
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

Returns a string which provides a fixed, machine parsable internal name for the plug-in. This name is used by MAXScript.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual HINSTANCE HInstance();
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

Returns the DLL instance handle of the plug-in. This is used so that string resources can be loaded by the ParamBlock2 system.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual int NumParamBlockDescs();
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Returns the number of ParamBlockDesc2s used by the plug-in.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual ParamBlockDesc2* GetParamBlockDesc(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Returns a pointer to the 'i-th' parameter block 2 descriptor.

**Parameters:**

**int i**

The zero based index of the descriptor to return.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual ParamBlockDesc2* GetParamBlockDescByID(BlockID id);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Returns a pointer to the specified parameter block 2 descriptor.

**Parameters:**

**BlockID id**

The ID of the parameter block.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void AddParamBlockDesc(ParamBlockDesc2* pbd);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Adds the specified parameter block 2 descriptor to the list of those maintained by the class.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block 2 descriptor to add.

**Default Implementation:**

{ }

**Prototype:**

**virtual void BeginEditParams(IObjParam \*ip, ReferenceMaker\* obj, ULONG flags, Animatable \*prev);**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

This method is called to handle the beginning of the automatic command panel user interface management provided by the param map 2 system. This method is called by the plug-in from its **Animatable::BeginEditParams()** method. The parameters passed to that method are simply passed along to this method.

**Parameters:**

**IObjParam \*ip**

The interface pointer passed to the plug-in.

**ReferenceMaker\* obj**

Points to the plug-in class calling this method.

**ULONG flags**

The flags passed along to the plug-in in **Animatable::BeginEditParams()**.

**Animatable \*prev**

The pointer passed to the plug-in in **Animatable::BeginEditParams()**.

**Default Implementation:**

{ }

**Prototype:**

**virtual void EndEditParams(IObjParam \*ip, ReferenceMaker\* obj, ULONG flags, Animatable \*prev);**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

This method is called to handle the ending of the automatic command panel user interface management provided by the param map 2 system. This method is called by the plug-in from its **Animatable::EndEditParams()** method. The parameters passed to that method are simply passed along to this method.

**Parameters:**

**IObjParam \*ip**

The interface pointer passed to the plug-in.

**ReferenceMaker\* obj**

Points to the plug-in class calling this method.

**ULONG flags**

The flags passed along to the plug-in in **Animatable::EndEditParams()**.

**Animatable \*prev**

The pointer passed to the plug-in in **Animatable::EndEditParams()**.

**Default Implementation:**

{ }

**Prototype:**

**virtual void InvalidateUI(ParamBlockDesc2\* pbd);**

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Invalidates the user interface for the rollup or dialog managed by the specified descriptor. This will cause the controls in that rollup to be redrawn.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block 2 descriptor whose corresponding UI is

invalidated.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void MakeAutoParamBlocks(ReferenceMaker* owner);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

This method creates the automatic parameter blocks for the specified plug-in.

These are the ones with the **ParamBlockDesc2.flags**

**P\_AUTO\_CONSTRUCT** bit set.

**Parameters:**

**ReferenceMaker\* owner**

Points to the owner of the parameter block.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int NumParamMaps();
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Returns the number of parameter map2s used by the plug-in.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual IParamMap2* GetParamMap(int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Returns a pointer to the 'i-th' parameter map2.

**Parameters:**

**int i**

The zero based index of the parameter map2 to return.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual IParamMap2* GetParamMap(ParamBlockDesc2* pbd);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Returns a pointer to the parameter map2 whose descriptor is passed.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block2 descriptor.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void SetUserDlgProc(ParamBlockDesc2* pbd,  
ParamMap2UserDlgProc* proc=NULL);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Sets the parameter map 2 user dialog proc for the specified descriptor.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block 2 descriptor.

**ParamMap2UserDlgProc\* proc=NULL**

This object manages user interface control that require special processing. See [Class ParamMap2UserDlgProc](#).

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual ParamMap2UserDlgProc*  
GetUserDlgProc(ParamBlockDesc2* pbd);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the System.

Returns a pointer to the parameter map 2 user dialog proc (if any) for the specified descriptor.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block 2 descriptor.

**Return Value:**

See [Class ParamMap2UserDlgProc](#).

**Default Implementation:**

```
{ return NULL; }
```

## Function Publishing Related Methods

**Prototype:**

```
virtual int NumInterfaces();
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Returns the number of function publishing interfaces maintained by the class descriptor.

**Prototype:**

**virtual FPInterface\* GetInterface(int i);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Returns a pointer to the 'i-th' function publishing interface.

**Parameters:**

**int i**

The zero based index of the interface to return.

**Prototype:**

**virtual FPInterface\* GetInterface(Interface\_ID id);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the function publishing interface whose ID is specified.

**Parameters:**

**Interface\_ID id**

The interface ID.

**Prototype:**

**virtual FPInterface\* GetInterface(TCHAR\* name);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the function publishing interface whose name is specified.

**Parameters:**

**TCHAR\* name**

The name of the interface.

**Prototype:**

**virtual void AddInterface(FPInterface\* fpi);**

**Remarks:**

This method is available in release 4.0 and later only.

Adds the specified interface to the list maintained by this class descriptor.

**Parameters:**

**FPInterface\* fpi**

Points to the interface to add.

**Prototype:**

**virtual void ClearInterfaces();**

**Remarks:**

This method is available in release 4.0 and later only.

Deletes all the interfaces maintained by the class descriptor.

## **Schematic View Related Methods**

**Prototype:**

**virtual bool DrawRepresentation(COLORREF bkColor, HDC  
hDC, Rect &rect);**

**Remarks:**

This method is available in release 3.0 and later only.

This method allows this plug-in class to provide a custom image for display in Schematic View.

**Parameters:**

**COLORREF bkColor**

The background color. See [COLORREF-DWORD format](#).

**HDC hDC**

The handle for the device context.

**Rect &rect**

The rectangle to draw in.

**Return Value:**

TRUE if this class can draw an image to represent itself graphically; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

## Generic Expansion Function

### Prototype:

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0);
```

### Remarks:

This method is available in release 3.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

### Parameters:

**int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

### Return Value:

An integer return value (defined uniquely for each **cmd**).

### Default Implementation:

```
{ return 0; }
```

## Class GammaMgr

See Also: [Class Color](#), [COLORREF](#).

class GammaMgr

### Description:

The gamma manager class. Methods of this class are used to gamma correct and de-gamma colors in various formats. Various settings from the 3ds max user interface are also accessible via data members of this class (for example the display, and file gamma settings). These settings may be read but should not be set by a plug-in developer. All methods of this class are implemented by the system.

There is a global instance of this class (defined in `\MAXSDK\INCLUDE\GAMMA.H`):

**GammaMgr gammaMgr;**

Note the following #defines. These are used to reduce the size of the gamma tables for correcting 16 bit values.

**#define RCBITS 13**

**#define RCOLN (1<<RCBITS)**

This class provides a set of commonly used gamma tables. This class does not provide tables for all types of conversion however. For example if you have a different gamma setting that you are using, or if you are going in a different conversion direction than the tables provided here you may use the classes [GamConvert16](#) and [GamConvert8](#) to build gamma tables.

### Data Members:

public:

**BOOL enable;**

Indicates if gamma correction is enabled or disabled.

**BOOL dithTrue;**

Indicates if output dithering is to be used for true color images.

**BOOL dithPaletted;**

Indicates if output dithering is to be used for paletted images.

**float dispGamma;**

The display gamma setting.

**float fileInGamma;**

The file input gamma setting.

**float fileOutGamma;**

The file output gamma setting.

**UBYTE disp\_gamtab[256];**

Display gamma table for drawing color swatches (8->8)

**UBYTE disp\_gamtabw[RCOLN];**

Display gamma table (RCBITS->8).

**UBYTE file\_in\_gamtab[256];**

File input gamma table (8->8).

**UWORD file\_in\_degamtab[256];**

For de-gamifying bitmaps on input. (8->16)

**UWORD file\_out\_gamtab[RCOLN];**

Gamma correct for file output, before dither (RCBITS->16).

**Methods:**

**Prototype:**

**GammaMgr();**

**Remarks:**

Constructor.

**Prototype:**

**inline COLORREF DisplayGammaCorrect(COLORREF col);**

**Remarks:**

Gamma corrects the specified color using the display gamma setting.

**Parameters:**

**COLORREF col**

The color to gamma correct.

**Return Value:**

The gamma corrected color.

**Prototype:**

## **Color DisplayGammaCorrect(Color c);**

### **Remarks:**

Gamma corrects the specified color using the display gamma setting.

### **Parameters:**

#### **Color c**

The color to gamma correct.

### **Return Value:**

The gamma corrected color.

### **Prototype:**

**void Enable(BOOL onOff);**

### **Remarks:**

Sets the gamma correction enabled setting.

### **Parameters:**

#### **BOOL onOff**

TRUE to enable; FALSE to disable.

### **Prototype:**

**BOOL IsEnabled();**

### **Remarks:**

Returns the gamma correction enabled setting; TRUE if enabled; FALSE if disabled.

### **Prototype:**

**void SetDisplayGamma(float gam);**

### **Remarks:**

Sets the display gamma setting.

### **Parameters:**

#### **float gam**

The value to set.

**Prototype:**

**float GetDisplayGamma();**

**Remarks:**

Returns the display gamma setting.

**Prototype:**

**void SetFileInGamma(float gam);**

**Remarks:**

Sets the file input gamma setting.

**Parameters:**

**float gam**

The value to set.

**Prototype:**

**float GetFileInGamma();**

**Remarks:**

Returns the file input gamma setting.

**Prototype:**

**void SetFileOutGamma(float gam);**

**Remarks:**

Sets the file output gamma setting.

**Parameters:**

**float gam**

The value to set.

**Prototype:**

**float GetFileOutGamma();**

**Remarks:**

Returns the file output gamma setting.

**The following functions are not part of class GammaMgr but are available for use.**

**Prototype:**

**inline COLORREF gammaCorrect(DWORD c);**

**Remarks:**

Returns a gamma corrected version of the specified color using the display gamma setting.

**Parameters:**

**DWORD c**

The color to gamma correct.

**Prototype:**

**inline UBYTE gammaCorrect(UBYTE b);**

**Remarks:**

Returns a gamma corrected version of the specified color using the display gamma setting.

**Parameters:**

**UBYTE b**

The color to gamma correct.

**Prototype:**

**void BuildGammaTab8(UBYTE gamtab[256], float gamma, int onoff=TRUE);**

**Remarks:**

Builds the gamma table that maps 8->8.

**Parameters:**

**UBYTE gamtab[256]**

The table to build.

**float gamma**

The gamma setting.

**int onoff=TRUE**

TRUE to enable; FALSE to disable.

**Prototype:**

**void BuildGammaTab8(UWORD gamtab[256], float gamma, int onoff=TRUE);**

**Remarks:**

Builds a gamma table that maps 8->16.

**Parameters:**

**UBYTE gamtab[256]**

The table to build.

**float gamma**

The gamma setting.

**int onoff=TRUE**

TRUE to enable; FALSE to disable.

**Prototype:**

**void BuildGammaTab(UBYTE gamtab[RCOLN], float gamma, int onoff=TRUE);**

**Remarks:**

Build a gamma table that maps RCBITS->8.

**Parameters:**

**UBYTE gamtab[RCOLN]**

The table to build.

**float gamma**

The gamma setting.

**int onoff=TRUE**

TRUE to enable; FALSE to disable.

**Prototype:**

**void BuildGammaTab(UWORD gamtab[RCOLN], float gamma, int onoff=TRUE);**

**Remarks:**

Build a gamma table that maps RCBITS->16.

**Parameters:**

**UWORD gamtab[RCOLN]**

The table to build.

**float gamma**

The gamma setting.

**int onoff=TRUE**

TRUE to enable; FALSE to disable.

**Prototype:**

**float gammaCorrect(float v, float gamma);**

**Remarks:**

Gamma corrects the value passed using the specified gamma setting.

**Parameters:**

**float v**

The value to gamma correct.

**float gamma**

The gamma setting.

**Return Value:**

The gamma corrected value.

**Prototype:**

**float deGammaCorrect(float v, float gamma);**

**Remarks:**

De-gamma corrects the value passed using the specified gamma setting.

**Parameters:**

**float v**

The value to de-gamma correct.

**float gamma**

The gamma setting.

**Prototype:**

**UBYTE gammaCorrect(UBYTE v, float gamma);**

**Remarks:**

Gamma corrects the value passed using the specified gamma setting.

**Parameters:**

**UBYTE v**

The value to gamma correct.

**float gamma**

The gamma setting.

**Return Value:**

The gamma corrected value.

**Prototype:**

**UBYTE deGammaCorrect(UBYTE v, float gamma);**

**Remarks:**

De-gamma corrects the value passed using the specified gamma setting.

**Parameters:**

**UBYTE v**

The value to de-gamma correct.

**float gamma**

The gamma setting.

**Prototype:**

**UWORD gammaCorrect(UWORD c, float gamma);**

**Remarks:**

Gamma corrects the value passed using the specified gamma setting.

**Parameters:**

**UWORD c**

The value to gamma correct.

**float gamma**

The gamma setting.

**Return Value:**

The gamma corrected value.

**Prototype:**

**UWORD deGammaCorrect(UWORD c, float gamma);**

**Remarks:**

De-gamma corrects the value passed using the specified gamma setting.

**Parameters:**

**UWORD c**

The value to de-gamma correct.

**float gamma**

The gamma setting.

## Class Quantizer

See Also: [BMM\\_Color\\_64](#), [BMM\\_Color\\_48](#), [BMM\\_Color\\_24](#).

class Quantizer

### Description:

Color quantizer, for doing true-color to paletted conversion. All methods of this class are implemented by the system. Create a Quantizer object by calling:

```
Quantizer *BMMNewQuantizer();
```

Be sure to call **Quantizer::DeleteThis()** when done.

### Methods:

#### Prototype:

```
virtual int Partition(BMM_Color_48 *pal, int palsize,  
BMM_Color_64 *forceCol)=0;
```

#### Remarks:

This method uses the histogram and computes the palette.

#### Parameters:

**BMM\_Color\_48 \*pal**

Storage for the palette to compute.

**int palsize**

The size of the palette.

**BMM\_Color\_64 \*forceCol**

If there is a color that you want to make sure is available in the palette, you may pass it here. This is used for the background color of an image for example. If this is not NULL the quantizer will make up a palette and make sure that this color is in it. For backgrounds, this looks much nicer because the background won't be dithered when shown using the palette.

#### Return Value:

Nonzero if the palette was computed; otherwise zero.

#### Prototype:

```
virtual void DeleteThis()=0;
```

**Remarks:**

This method is called to delete the Quantizer when you are done with it.

**Prototype:**

```
virtual int AllocHistogram(void)=0;
```

**Remarks:**

This method allocates the histogram used in doing the conversion.

**Return Value:**

Nonzero if the histogram was allocated; otherwise zero.

**Prototype:**

```
virtual void AddToHistogram(BMM_Color_64 *image, int  
npix)=0;
```

**Remarks:**

Adds the specified colors to the histogram so they are taken into account in the palette computations.

**Parameters:**

```
BMM_Color_64 *image
```

The pixels to include.

```
int npix
```

The number of pixels above.

**Prototype:**

```
virtual void AddToHistogram(BMM_Color_48 *image, int  
npix)=0;
```

**Remarks:**

Adds the specified colors to the histogram so they are taken into account in the palette computations.

**Parameters:**

```
BMM_Color_48 *image
```

The pixels to include.

```
int npix
```

The number of pixels above.

**Prototype:**

```
virtual void AddToHistogram(BMM_Color_24 *image, int  
npix)=0;
```

**Remarks:**

Adds the specified colors to the histogram so they are taken into account in the palette computations.

**Parameters:**

**BMM\_Color\_24 \*image**

The pixels to include.

**int npix**

The number of pixels above.

# Class ColorPacker

See Also: [Palettes](#), [Class Quantizer](#).

class ColorPacker

## Description:

Methods of this class are used for packing colors into a 256 color paletted representation. Create an instance of this class using

**BMMNewColorPacker()** described below. All methods of this class are implemented by the system.

## Prototype:

```
ColorPacker *BMMNewColorPacker(int w, BMM_Color_48 *pal,  
int npal, BYTE* remap=NULL);
```

## Remarks:

This is called to create an instance of a ColorPacker. When done, be sure to call **ColorPacker::DeleteThis()**.

## Parameters:

**int w**

The width of bitmap to be packed.

**BMM\_Color\_48 \*pal**

The palette to use.

**int npal**

The number of entries in the palette.

**BYTE\* remap=NULL**

This is a 256 byte table that maps the numbers into another number. This is used so the palette may be rearranged in Windows order. To make palettes for Windows, the best thing to do is to put the colors in so that colors that Windows uses are either left alone, or they are occupied by the colors that are least important in the image. This is because Windows will come along and alter these colors. This is the first 10 colors and the last 10 colors.

The quantizer creates a palette from 0-239, where 0 is the most used color and 239 is the least used. The color packer uses 0-239 as well, and it operates most efficiently when the colors are sorted as the quantizer orders them.

What can be done is to use **FixPaletteForWindows()** to rearrange the colors for the Windows palette. This creates the **remap** table passed to this method. Then as a last step the remap table is used to reorganize the palette. Below is the documentation for the global function **FixPaletteForWindows()**. Sample code that uses these APIs is in **\MAXSDK\SAMPLES\IO\FLIC\FLIC.CPP**.

**Prototype:**

```
void FixPaletteForWindows(BMM_Color_48 *pal,  
BMM_Color_48 *newpal,int ncols, BYTE *remap=NULL);
```

**Remarks:**

Rearranges the palette **pal** (which has colors 0..**ncols**-1 occupied, in descending order of frequency), into **newpal** so that the colors 10-245 are populated first, then 0-9, then 246-255. Sets the optional array **remap** to map the old palette index values to the new ones.

**Parameters:**

**BMM\_Color\_48 \*pal**

The palette to rearrange.

**BMM\_Color\_48 \*newpal**

The rearranged palette.

**int ncols**

The number of colors in the palette.

**BYTE \*remap=NULL**

An array that maps the old palette index values to the new ones.

**Methods:**

**Prototype:**

```
virtual void DeleteThis()=0;
```

**Remarks:**

This method is called to delete the ColorPacker when you are done with it.

**Prototype:**

```
virtual void EnableDither(BOOL onoff)=0;
```

**Remarks:**

This method is used to enable dithering of the packed pixels. It defaults to the 3ds max default.

**Parameters:****BOOL onoff**

TRUE to enable dithering; FALSE to disable.

**Prototype:**

**virtual void PropagateErrorBetweenLines(BOOL onoff)=0;**

**Remarks:**

This method controls the propagation of error between lines. For static images this is best left to default to on. For animated images, it is better to set this to off and not propagate the error between lines. This defaults to ON.

**Parameters:****BOOL onoff**

TRUE to enable error propagation between lines; FALSE to disable.

**Prototype:**

**virtual void PackLine(BMM\_Color\_64\* in, BYTE \*out, int w)=0;**

**Remarks:**

Packs the specified line of pixels into the 256 color representation.

**Parameters:****BMM\_Color\_64\* in**

The line of pixels to pack.

**BYTE \*out**

The result, the output pixels.

**int w**

The number of pixels in the line.

**Prototype:**

**virtual void PackLine(BMM\_Color\_48\* in, BYTE \*out, int w)=0;**

**Remarks:**

Packs the specified line of pixels into the 256 color representation.

**Parameters:**

**BMM\_Color\_48\* in**

The line of pixels to pack.

**BYTE \*out**

The result, the output pixels.

**int w**

The number of pixels in the line.

## Class CStr

See Also: [Class WStr](#), [Character Strings](#).

class CStr

### Description:

A simple character string class. This is the standard character string class used in MAX. Methods and operators are provided for calculating lengths, concatenation, substring operations, character searching, case conversion, comparison, and formatted writing.

This class automatically allocates the proper amount of space for the string. This is very handy in the case of internationalization / localization. For example, if you code something like:

```
TSTR myString = GetString(IDS_STRING_ID);
```

then **myString**'s constructor will allocate enough space to store the resource string no matter how long it is. This is much better than doing the following:

```
TCHAR myString[64];  
_tcscpy(myString, GetString(IDS_STRING_ID));
```

because the resource string may turn out to be much longer than 64 bytes once it's translated to German or French (or whatever).

As another example, if you have the following code:

```
TSTR str1 = _T("This is string1.");  
TSTR str2 = _T("This is string2.");
```

Then

```
TSTR concatStr = str1 + str2;
```

will again yield a (concatenated) string will enough space to hold the concatenated contents of **str1** and **str2**, automatically.

All methods are implemented by the system.

### Methods:

### Prototype:

```
CStr();
```

### Remarks:

Constructor. The string is set to NULL.

**Prototype:**

**CStr(const char \*cs);**

**Remarks:**

Constructor. The string is initialized to the string passed.

**Prototype:**

**CStr(const wchar\_t \*wcstring);**

**Remarks:**

Constructor. The string is initialized to the string passed.

**Prototype:**

**CStr(const CStr& ws);**

**Remarks:**

Constructor. The string is initialized to the string passed.

**Prototype:**

**~CStr()**

**Remarks:**

Destructor. The string is deleted.

**Prototype:**

**char \*data();**

**Remarks:**

Returns a pointer to the string. If the string is NULL, 0 is returned.

**Prototype:**

**const char \*data() const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the string. If the string is NULL, 0 is returned.

**Prototype:**

**operator char \*();**

**Remarks:**

Returns a pointer to the string. If the string is NULL, 0 is returned.

**Prototype:**

**void Resize(int nchars);**

**Remarks:**

Reallocates the string to contain **nchars** characters. If the string is enlarged it is padded with blanks.

**Parameters:**

**int nchars**

Specifies the new number of characters for the string.

**Prototype:**

**int Length() const;**

**Remarks:**

Returns the number of characters in the string.

**Prototype:**

**int length() const;**

**Remarks:**

Returns the number of characters in the string.

**Prototype:**

**BOOL isNull();**

**Remarks:**

Returns TRUE if the string length is 0; otherwise FALSE.

**Prototype:**

**CStr & operator=(const CStr& cs);**

**Remarks:**

Assignment operator.

**Prototype:**

**CStr & operator=(const wchar\_t \*wcstring);**

**Remarks:**

Assignment operator.

**Prototype:**

**CStr & operator=(const char \*cs);**

**Remarks:**

Assignment operator. In release 3.0 and later this method check for self-assignment.

**Prototype:**

**CStr operator+(const CStr& cs) const;**

**Remarks:**

Concatenation operator. Returns a new string that is this string with string **cs** appended.

**Prototype:**

**CStr& operator+=(const CStr& cs);**

**Remarks:**

Concatenation. Returns this string with **cs** appended.

**Prototype:**

**CStr& Append(const CStr& cs);**

**Remarks:**

Concatenation. Returns this string with **cs** appended.

**Prototype:**

**CStr& append(const CStr& cs);**

**Remarks:**

Concatenation. Returns this string with **cs** appended to the end.

**Prototype:**

**CStr& remove(int pos);**

**Remarks:**

Returns this string with all characters from **pos** to the end removed.

**Parameters:**

**int pos**

Specifies the last position in the string.

**Prototype:**

**CStr& remove(int pos, int N);**

**Remarks:**

Returns this string with N characters removed from **pos** to the end.

**Parameters:**

**int pos**

Specifies the position to begin removing characters.

**int N**

Specifies the number of characters to remove.

**Prototype:**

**CStr Substr(int start, int nchars) const;**

**Remarks:**

Returns a substring of this string, beginning at position **start**, of length **nchars**.

**Prototype:**

**char& operator[(int i)]**

**Remarks:**

Returns a substring of this string beginning at position **i**.

**Prototype:**

**const char& operator[](int i) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a substring of this string beginning at position **i**.

**Prototype:**

**int first(char c) const;**

**Remarks:**

Returns the index of the first occurrence of character **c** in this string. Returns -1 if not found.

**Prototype:**

**int last(char c) const;**

**Remarks:**

Returns the index of the last occurrence of character **c** in this string. Returns -1 if not found.

**Prototype:**

**int operator==(const CStr &cs) const;**

**Remarks:**

Equality operator.

**Return Value:**

Nonzero if the strings are equal; otherwise 0.

**Prototype:**

**int operator<(const CStr &cs) const;**

**Remarks:**

Returns nonzero if this string is less than **cs**; otherwise 0.

**Prototype:**

**int operator<=(const CStr &ws) const;**

**Remarks:**

Returns nonzero if this string is less than or equal to **ws**; otherwise 0.

**Prototype:**

**int operator>(const CStr &ws) const;**

**Remarks:**

Returns nonzero if this string is greater than **ws**; otherwise 0.

**Prototype:**

**int operator>=(const CStr &ws) const;**

**Remarks:**

Returns nonzero if this string is greater than or equal to **ws**; otherwise 0.

**Prototype:**

**void toUpper();**

**Remarks:**

Converts all character of this string to uppercase.

**Prototype:**

**void toLower();**

**Remarks:**

Converts all character of this string to lowercase.

**Prototype:**

**int printf(const char \*format, ...);**

**Remarks:**

Formatted output to this string. The internal buffer size for the output string is 512 bytes.

**Return Value:**

The number of character output or EOF on error.

**Sample Code:**

**TSTR buf;**

**buf.printf(\_T("Rendering In Progress: Frame %d"), curFrame);**

## Class WStr

See Also: [Class CStr](#), [Character Strings](#).

class WStr

### Description:

A wide character string class. This class uses 16 bits to hold each character. Methods and operators are provided for calculating lengths, concatenation, substring operations, character searching, case conversion, comparison, and formatted writing. All methods are implemented by the system.

OLE file IO requires the wide characters of WStr.

### Methods:

#### Prototype:

**WStr();**

#### Remarks:

Constructor. The string is set to NULL.

#### Prototype:

**WStr(const char \*cs);**

#### Remarks:

Constructor. The string is initialized to the string passed.

#### Prototype:

**WStr(const wchar\_t \*wcstring);**

#### Remarks:

Constructor. The string is initialized to the string passed.

#### Prototype:

**WStr(const WStr& ws);**

#### Remarks:

Constructor. The string is initialized to the string passed.

**Prototype:**

`~WStr()`

**Remarks:**

Destructor. The string is deleted.

**Prototype:**

`wchar_t *data();`

**Remarks:**

Returns a pointer to the string. If the string is NULL, 0 is returned.

**Prototype:**

`const wchar_t *data() const;`

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the string. If the string is NULL, 0 is returned.

**Prototype:**

`const wchar_t *data() const;`

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the string. If the string is NULL, 0 is returned.

**Prototype:**

`operator wchar_t *();`

**Remarks:**

Returns a pointer to the string. If the string is NULL, 0 is returned.

**Prototype:**

`void Resize(int nchars);`

**Remarks:**

Reallocates the string to contain **nchars** characters. If the string is enlarged it

is padded with blanks.

**Parameters:**

**int nchars**

Specifies the new number of characters for the string.

**Prototype:**

**int Length() const;**

**Remarks:**

Returns the number of characters in the string.

**Prototype:**

**int length() const;**

**Remarks:**

Returns the number of characters in the string.

**Prototype:**

**BOOL isNull();**

**Remarks:**

Returns TRUE if the string length is 0; otherwise FALSE.

**Prototype:**

**WStr & operator=(const WStr& ws);**

**Remarks:**

Assignment operator. In release 3.0 and later this operator checks for self-assignment.

**Prototype:**

**WStr & operator=(const wchar\_t \*wctr);**

**Remarks:**

Assignment operator.

**Prototype:**

**WStr & operator=(const char \*cstr);**

**Remarks:**

Assignment operator.

**Prototype:**

**WStr operator+(const WStr& ws) const;**

**Remarks:**

Concatenation operator. Returns a new string that is this string with string **ws** appended.

**Prototype:**

**WStr & operator+=(const WStr& ws);**

**Remarks:**

Concatenation. Returns this string with **ws** appended.

**Prototype:**

**WStr& Append(const WStr& ws)**

**Remarks:**

Concatenation. Returns this string with **ws** appended.

**Prototype:**

**WStr& append(const WStr& ws);**

**Remarks:**

Concatenation. Returns this string with **ws** appended.

**Prototype:**

**WStr& remove(int pos);**

**Remarks:**

Returns this string with N characters removed from **pos** to the end.

**Parameters:**

**int pos**

Specifies the position to begin removing characters.

**int N**

Specifies the number of characters to remove.

**Prototype:**

**WStr Substr(int start, int nchars) const;**

**Remarks:**

Returns a substring of this string, beginning at position **start**, of length **nchars**.

**Prototype:**

**wchar\_t& operator[](int i)**

**Remarks:**

Returns a substring of this string beginning at position **i**.

**Prototype:**

**int first(wchar\_t c) const;**

**Remarks:**

Returns the index of the first occurrence of character **c** in this string. Returns -1 if not found.

**Prototype:**

**int last(wchar\_t c) const;**

**Remarks:**

Returns the index of the last occurrence of character **c** in this string. Returns -1 if not found.

**Prototype:**

**int operator==(const WStr &ws) const;**

**Remarks:**

Equality operator.

**Return Value:**

Nonzero if the strings are equal; otherwise 0.

**Prototype:**

```
int operator<(const WStr &ws) const;
```

**Remarks:**

Returns nonzero if this string is less than **ws**; otherwise 0.

**Prototype:**

```
int operator<=(const WStr &ws) const;
```

**Remarks:**

Returns nonzero if this string is less than or equal to **ws**; otherwise 0.

**Prototype:**

```
int operator>(const WStr &ws) const;
```

**Remarks:**

Returns nonzero if this string is greater than **ws**; otherwise 0.

**Prototype:**

```
int operator>=(const WStr &ws) const;
```

**Remarks:**

Returns nonzero if this string is greater than or equal to **ws**; otherwise 0.

**Prototype:**

```
void toUpper();
```

**Remarks:**

Converts all character of this string to uppercase.

**Prototype:**

```
void toLower();
```

**Remarks:**

Converts all character of this string to lowercase.

**Prototype:**

```
int printf(const wchar_t *format, ...);
```

**Remarks:**

Formatted output to this string.

**Return Value:**

The number of character output or EOF on error.

**Sample Code:**

```
WStr buf;  
buf.printf(_T("Rendering In Progress: Frame %d"),curFrame);
```

## Class NameTab

See Also: [Class Tab](#).

```
class NameTab : public Tab<TCHAR *>
```

### Description:

This class is used to store a table of names. For example, this class is used by lights for their "Inclusion" and "Exclusion" lists. This class maintains an 'include' flag that specifies whether the list of names is things to be included, or things to be excluded. There is no reason the NameTab class can't be used for other things where inclusion/exclusion is not relevant: in that case one can just ignore the 'include' flag. All methods are implemented by the system.

### Methods:

#### Prototype:

**NameTab()**

#### Remarks:

Constructor. The 'include' flag is set to FALSE.

#### Prototype:

**void SetFlag(ULONG f, BOOL b=1)**

#### Remarks:

Sets the specified flag to the specified value.

#### Parameters:

**ULONG f**

The flag(s) to set. One or more of the following values:

**NT\_INCLUDE**

This bit is used to indicate "Include" mode.

**NT\_AFFECT\_ILLUM**

This bit is used to indicate the "Illumination" check box in the exclusion list dialog.

**NT\_AFFECT\_SHADOWCAST**

This bit is used to indicate the "Shadow Casting" check box in the exclusion list dialog.

**BOOL b=1**

The value to set.

**Prototype:**

**BOOL TestFlag(ULONG f)**

**Remarks:**

Returns TRUE if the specified flag(s) are set; otherwise FALSE.

**Parameters:**

**ULONG f**

The flag(s) to set. One or more of the following values:

**NT\_INCLUDE**

This bit is used to indicate "Include" mode.

**NT\_AFFECT\_ILLUM**

This bit is used to indicate the "Illumination" check box in the exclusion list dialog.

**NT\_AFFECT\_SHADOWCAST**

This bit is used to indicate the "Shadow Casting" check box in the exclusion list dialog.

**Prototype:**

**int AddName(TCHAR \*n);**

**Remarks:**

Appends the specified name to the end of the list.

**Parameters:**

**TCHAR \*n**

The name to add.

**Return Value:**

Returns the number of items in the list prior to appending.

**Prototype:**

**void SetName(int i, TCHAR \*n);**

**Remarks:**

Stores the specified name at the specified position in the list.

**Parameters:**

**int i**

The position in the list for the name.

**TCHAR \*n**

The name to store. If the name is NULL, the 'i-th' entry is set to NULL.

**Prototype:**

**void SetSize(int num);**

**Remarks:**

Sets the size of the list. If the new size is smaller than the current size entries are deleted.

**Parameters:**

**int num**

Specifies the size of the list.

**Prototype:**

**void RemoveName(int i);**

**Remarks:**

Removes the 'i-th' name from the list.

**Parameters:**

**int i**

Specifies the index of the name to remove.

**Prototype:**

**int FindName(TCHAR\* n);**

**Remarks:**

Returns the index of the name passed; otherwise returns -1.

**Parameters:**

**TCHAR\* n**

The name to find.

**Prototype:**

**IOResult Load(ILoad \*iLoad);**

**Remarks:**

Loads this NameTab from disk.

**Parameters:**

**ILoad \*iLoad**

This class provides methods to load data from disk.

**Return Value:**

See Also: [List of IO Results](#).

**Prototype:**

**IOResult Save(ISave \*iSave);**

**Remarks:**

Saves this NameTab to disk.

**Parameters:**

**ISave \*iSave**

This class provides methods to save data to disk.

**Return Value:**

See Also: [List of IO Results](#).

**Operators:**

**Prototype:**

**NameTab& operator=(const NameTab& n);**

**Remarks:**

Assignment operator. The specified **NameTab** is copied to this **NameTab**.

## Template Class Tab

See Also: [Class BitArray](#).

```
template <class T> class Tab
```

### Description:

This is a generic table class. This is a type-safe variable length array which also supports list-like operations of insertion, appending and deleting. Two instance variables are maintained: **nalloc** is the number elements allocated in the array; **count** is the number actual used. (**count** ≤ **nalloc**). Allocation is performed automatically when Insert or Append operations are performed. It can also be done manually by calling **Resize()** or **Shrink()**.

Note: Delete does not resize the storage: to do this call **Shrink()**. If you are going to do a sequence of Appends, its more efficient to first call **Resize()** to make room for them. Beware of using the **Addr()** function: it returns a pointer which may be invalid after subsequent **Insert()**, **Append()**, **Delete()**, **Resize()**, or **Shrink()** operations.

Also note: In 3ds max 1.x, the method **SetCount(n)** will set the count to **n**, but will not assure that only **n** items are allocated. To do that you should call **Resize(n)**. This sets the number allocated. It will also make sure that **count** ≤ **numAlloc**. To make sure that exactly **n** are allocated **and** that **count** = **n**, call both **Resize(n)** and **SetCount(n)**. In 3ds max 2.x and later using **SetCount()** will also effectively call **Resize()**.

The implementation minimizes the storage of empty Tables: they are represented by a single NULL pointer. Also, the major part of the code is generic, shared by different Tabs for different types of elements.

Tabs may be used on the stack, i.e. they may be declared as a local variable of a function or method. You can set the number of elements in the table, work with them, and then when the function returns, the destructor of the Tab is called, and the memory will be deallocated.

Tabs are only appropriate for use with classes that don't allocate memory. For example, Tab<float> is fine while Tab<TSTR> is problematic (TSTR is the class used for strings in 3ds max). In this case, the TSTR class itself allocates memory for the string. It relies on its constructor or destructor to allocate and free the memory. The problem is the Tab class will not call the constructors and

destructors for all the items in the table, nor will it call the copy operator. As an example of this, when you assign a string to another string, the TSTR class does not just copy the pointer to the string buffer (which would result in two items pointing to the same block of memory). Rather it will allocate new memory and copy the contents of the source buffer. In this way you have two individual pointers pointing at two individual buffers. When each of the TSTR destructors is called it will free each piece of memory. So, the problem with using a Tab<TSTR> is that when you assign a Tab to another Tab, the Tab copy constructor will copy all the elements in the table, but it will not call the copy operator on the individual elements. Thus, if you had a Tab<TSTR> and you assigned it to another Tab<TSTR>, you'd have two TSTRs pointing to the same memory. Then when the second one gets deleted it will be trying to double free that memory.

So again, you should only put things in a Tab that don't allocate and deallocate memory in their destructors. Thus, this class should not be used with classes that implement an assignment operator and or destructor because neither are guaranteed to be called. The way around this is to use a table of pointers to the items. For example, instead of Tab<TSTR> use Tab <TSTR \*>. As another example, Tab<int> is OK, while Tab<BitArray> would be no good. In the BitArray case one should use class pointers, i.e. Tab<BitArray \*>.

All methods of this class are implemented by the system except the compare function used in sorting (see **Sort()**).

**Methods:**

**Prototype:**

**Tab(const Tab& tb)**

**Remarks:**

Copy constructor.

**Parameters:**

**const Tab& tb**

The table to copy.

**Prototype:**

**~Tab()**

**Remarks:**

Destructor. The memory associated with the table is freed.

**Prototype:**

**void Init();**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Provides a way of initializing a **Tab**<> instance outside of its constructor. Can be used to init **Tab**<> instances in malloc'd memory.

**Prototype:**

**int Count() const**

**Remarks:**

Returns the number of entries being used.

**Prototype:**

**void ZeroCount()**

**Remarks:**

Set the number of elements in the array that are actually used to zero.

**Prototype:**

**void SetCount(int n)**

**Remarks:**

Set the number of elements in the array that are actually used to **n**.

**Parameters:**

**int n**

The number of elements in the array that are actually used.

**Prototype:**

**T\* Addr(const int i) const**

**Remarks:**

Returns the address of the i-th element. Beware of using this method as it returns a pointer which may be invalid after subsequent Insert, Append, Delete, Resize, or Shrink operations.

**Parameters:**

**const int i**

Specifies the element to return the address of.

**Return Value:**

Pointer to the i-th element.

**Prototype:**

**int Insert(int at, int num, T \*el)**

**Remarks:**

Insert "num" elements at position "at".

**Parameters:**

**int at**

Array index where to insert the elements.

**int num**

Number of elements to insert.

**T \*el**

Array of elements to insert.

**Return Value:**

Returns **at**.

**Prototype:**

**int Append(int num, T \*el, int allocExtra=0)**

**Remarks:**

Append "num" elements at the end of the array. If you need to enlarge the array, allocate "allocExtra" extra slots

**Parameters:**

**int num**

The number of elements to append to the end of the array.

**T \*el**

The elements to append.

**int allocExtra=0**

If you need to enlarge the array, allocate "allocExtra" extra slots.

**Return Value:**

Returns the number of elements in use (prior to appending).

**Prototype:**

**int Delete(int start,int num)**

**Remarks:**

List-type delete of "num" elements starting with "start"

**Parameters:**

**int start**

The start position for element deletion.

**int num**

The number of elements to delete.

**Return Value:**

Returns the number of items left in the table.

**Prototype:**

**int Resize(int num)**

**Remarks:**

Changes the number of allocated items to "num".

**Parameters:**

**int num**

The new size of the array.

**Return Value:**

Nonzero if the array was resized; otherwise 0.

**Prototype:**

**void Shrink()**

**Remarks:**

Reallocate so there is no wasted space (nalloc = count).

**Prototype:**

```
void Sort(CompareFnc cmp)
```

**Remarks:**

Sorts the array using the compare function.

**Parameters:****CompareFnc cmp**

Type of function to pass to **Sort()**. Note: **Sort()** just uses the C library `qsort` function. The developer must implement the **CompareFnc** function.

```
typedef int( __cdecl *CompareFnc) (const void *elem1, const void *elem2);
```

The return value of **CompareFnc** is show below in the relationship of **elem1** to **elem2**:

< 0

if **elem1** less than **elem2**

0

if **elem1** identical to **elem2**

> 0

if **elem1** greater than **elem2**

**Sample Code:**

```
static int CompTable( const void *elem1, const void *elem2 ) {  
    TCHAR *a = (TCHAR *)elem1;  
    TCHAR *b = (TCHAR *)elem2;  
    return(_tcscmp(a,b));  
}
```

**Operators:****Prototype:**

```
Tab& operator=(const Tab& tb)
```

**Remarks:**

Assignment operator. The **Tab** maintains a pointer to the table buffer. This assignment operator copies the table data from the item you are copying from into the memory buffer of the item you are copying to.

**Parameters:**

**const Tab& tb**

The table to assign.

**Prototype:**

**T& operator[](const int i) const**

**Remarks:**

Accesses the i-th entry.

**Parameters:**

**const int i**

The index of the element to access.

**Return Value:**

The i-th element.

# Class ColorPicker

See Also: [Class HSVCallback](#), [COLORREF - DWORD](#), [Class IPoint2](#).

```
class ColorPicker : public InterfaceServer
```

## Description:

This class allows a plug-in to create a **modeless** color picker dialog box.

Developers may also create a **modal** version of this dialog box. The function available below is defined for this purpose. Note that this is not a class method but a global function.

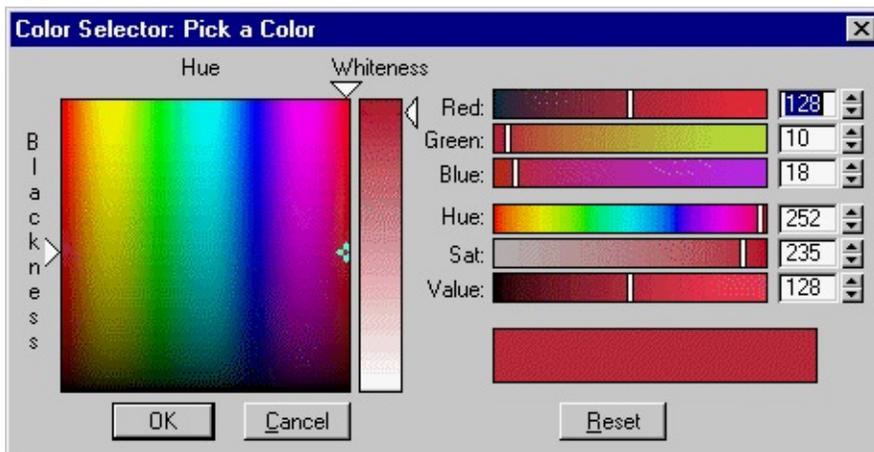
To use these APIs you need to **#include "hsv.h"**.

## Prototype:

```
int HSVDlg_Do(HWND hwndOwner, DWORD *lpc, IPoint2 *spos, HSVCallback *callBack, TCHAR *name);
```

## Remarks:

This method puts up the modal HSV color picker dialog box. This dialog appears below:



## Parameters:

**HWND hwndOwner**

Owner window handle.

**DWORD \*lpc**

Pointer to color to be edited. See [COLORREF](#).

**IPoint2 \*spos**

The starting position of the upper left corner of the dialog window. This is set to the ending position when the user is done. You may pass NULL to get the default location.

**HSVCallback \*callBack**

Callback object whose procs are called when the color changes. See [Class HSVCallback](#).

**TCHAR \*name**

The name of color being edited to appear in the title bar.

**Return Value:**

Returns TRUE if the user exists the dialog with OK, otherwise returns FALSE.

The following functions are also available

**Prototype:**

```
ColorPicker *CreateColorPicker(HWND hwndOwner, DWORD  
initColor, IPoint2* spos, HSVCallback *pcallback, TCHAR *name,  
int objClr=0);
```

**Remarks:**

Call this function to bring up the modeless color picker.

**Parameters:**

**HWND hwndOwner**

The owner window handle.

**DWORD initColor**

The initial color for the color picker.

**IPoint2\* spos**

The initial screen position of the upper left corner. NULL may be passed for the default location.

**HSVCallback \*pcallback**

The callback object to respond to color change events.

**TCHAR \*name**

The title string in the dialog.

**int objClr=0**

A BOOLEAN used to indicate that the ColorPicker is being used to set the object color from the control panel. In all other cases, the default value of 0 should be used.

**Return Value:**

A pointer to a ColorPicker object.

**Prototype:**

```
void SetCPInitPos(IPoint2 &pos);
```

**Remarks:**

Establishes the color picker initial screen position.

**Parameters:**

**IPoint2 &pos**

The upper left corner screen coordinate for the color picker.

**Prototype:**

**IPoint2 GetCPInitPos();**

**Remarks:**

Retrieves the color picker initial screen position.

**Return Value:**

The screen coordinates of the color picker. This is the coordinate of the upper left corner.

**Methods:**

**Prototype:**

**ColorPicker();**

**Remarks:**

Constructor.

**Prototype:**

**virtual ~ColorPicker();**

**Remarks:**

Destructor.

**Prototype:**

**virtual void ModifyColor(DWORD color)=0;**

**Remarks:**

This method changes the current color in the color picker, but does not change the "reset" color.

**Parameters:**

**DWORD color**

The current color.

**Prototype:**

**virtual void SetNewColor(DWORD color, TCHAR \*name)=0;**

**Remarks:**

Sets a new color as current in the dialog.

**Parameters:**

**DWORD color**

The color to set.

**TCHAR \*name**

A new name to display in the title bar.

**Prototype:**

**virtual DWORD GetColor()=0;**

**Remarks:**

Returns the current color.

**Prototype:**

**virtual IPoint2 GetPosition()=0;**

**Remarks:**

Returns the screen position of the upper left corner of the dialog as a IPoint2.

**Prototype:**

**virtual void Destroy()=0;**

**Remarks:**

This is called when the parent is going away.

**Prototype:**

**virtual void InstallNewCB(DWORD col, HSVCallback \*pcb,  
TCHAR \*name)=0;**

**Remarks:**

This method is used to add a different callback, set a new initial color and update the title string.

**Parameters:**

**DWORD col**

The new initial color.

**HSVCallback \*pcb**

The new callback.

**TCHAR \*name**

The new title string.

**Prototype:**

**virtual void RefreshUI();**

**Remarks:**

This method is available in release 4.0 and later only.

This method is called when the display gamma changes.

# **COLORREF - DWORD Color Format**

### **From the Windows API:**

The COLORREF value is a 32-bit value used to specify an RGB color.

#### **COLORREF RGB(**

**BYTE bRed, // red component of color**

**BYTE bGreen, // green component of color**

**BYTE bBlue // blue component of color**

**);**

When specifying an explicit RGB color, the COLORREF value has the following hexadecimal form:

**0x00bbggrr**

The low-order byte contains a value for the relative intensity of red; the second byte contains a value for green; and the third byte contains a value for blue. The high-order byte must be zero. The maximum value for a single byte is 0xFF.

Also see the following macros in the Windows API:

**GetBValue, GetGValue, GetRValue, RGB.**

These are used to create COLORREF values and break them down into component values. These can be used as follows:

**COLORREF fillcolor = RGB(48, 96, 192);**

**int r = (int) GetRValue(fillcolor);**

**int g = (int) GetGValue(fillcolor);**

**int b = (int) GetBValue(fillcolor);**

## Class MNMesh

See Also: [Class FlagUser](#), [Class MNFace](#), [Class MNVert](#), [Class MNEdge](#), [Class MNMap](#), [Class PerData](#), [Class MNMeshBorder](#), [Class Mesh](#), [Class BitArray](#), [Template Class Tab](#), [MNMesh Note on Edge Characteristics](#), [Class MNMeshBorder](#), [MNMesh Notes on Debugging](#).

```
class MNMesh : public FlagUser, public BaseInterfaceServer
```

### Description:

This class is available in release 2.0 and later only.

The MNMesh class is provided for temporary use by plug-ins, to help with complex topology-based modifications to Meshes. It has certain capabilities, such as the ability to recognize faces with more than 3 sides, that are useful in some applications. It is not suitable for use as a pipeline object. All methods of this class are implemented by the system.

MNMesh has a winged-edge structure and a highly interlinked topology. This is very useful for some modifiers, but requires far more memory & processing time than an equivalent normal mesh. It's convenient for the programmer, but may be sluggish for the user.

Although some methods can be a bit slow, none of them are non-linear in the amount of time they take. Even the tessellation algorithms, which can as much as quadruple the size of the meshes, operate locally, vertex by vertex, so they are order-of-n algorithms.

Note: You must **#include "MNMATH.H"** to use this class as it's not included by default by **MAX.H**.

### Methods Groups:

The hyperlinks below take you to the start of groups of related methods within the class:

[Initialization & Cleanup](#)

[Access to mesh components](#)

[Adding new components](#)

[Removing & deleting components](#)

[Internal computation](#)

[Component Targeting & Flag methods](#)

[Component information methods](#)

[Smoothing-group & Material methods](#)

[Face-joining methods](#)

[Splitting methods](#)

[Border methods](#)

[Tessellation & related methods](#)

[Welding methods](#)

[Boolean operations](#)

[Debugging methods](#)

[Operators](#)

## Data Members:

public:

**MNVert \*v;**

This data member is available in release 3.0 and later only.

The array of MNMesh verticies.

**MNEdge \*e;**

This data member is available in release 3.0 and later only.

The array of MNMesh edges.

**MNFace \*f;**

This data member is available in release 3.0 and later only.

The array of MNMesh faces.

**int numv**

This data member is available in release 3.0 and later only.

The number of MNVerts in **v**.

**int nume**

This data member is available in release 3.0 and later only.

The number of MNEdges in **e**.

**int numf**

This data member is available in release 3.0 and later only.

The number of MNFaces in **f**.

**int numm;**

This data member is available in release 3.0 and later only.

The number of MNMaps in **m**.

**PerData \*vd;**

This data member is available in release 3.0 and later only.

The array of PerData objects which maintain and provide access to the floating point vertex data. There is one of these for each supported channel. PerData objects are used to store such information as vertex weighting and weighted selections.

### **BitArray vdSupport;**

This data member is available in release 3.0 and later only.

This bit array indicates if a particular vertex data channel is supported in this MNMesh. If the bit is set the channel is supported.

### **PerData \*ed;**

This data member is available in release 3.0 and later only.

The array of PerData objects which maintain and provide access to the floating point edge data. There is one of these for each supported channel. One PerData object store edge weighting for NURMS-type MeshSmooths.

### **BitArray edSupport;**

This data member is available in release 3.0 and later only.

This bit array indicates if a particular edge data channel is supported in this MNMesh. If the bit is set the channel is supported.

### **DWORD selLevel;**

This data member is available in release 3.0 and later only. Note however that, since MNMeshes are not yet used as pipeline objects, the subobject selection methods are not yet used in 3ds max. Third party developers are welcome to use these methods themselves, but they should be aware that this is a work-in-progress.

The current MNMesh selection level. One of the following values:

**MNM\_SL\_OBJECT** - object level selection.

**MNM\_SL\_VERTEX** - select vertices

**MNM\_SL\_EDGE** - select edges

**MNM\_SL\_FACE** - select polygon faces

### **DWORD dispFlags;**

This data member is available in release 3.0 and later only. Note however that, since MNMeshes are not yet used as pipeline objects, the display methods and flags are not yet used in 3ds max. Third party developers are welcome to use the display methods themselves, but they should be aware that this is a work-

in-progress.

The display flags. One or more of the following values:

**MNDISP\_VERTTICKS** - Display vertex tick marks.

**MNDISP\_SELVERTS** - Display selected verticies.

**MNDISP\_SELFACES** - Display selected faces.

**MNDISP\_SELEDGES** - Display selected edges.

**MNDISP\_NORMALS** - Display face normals.

**MNDISP\_SMOOTH\_SUBSEL** - Analogous to the Mesh display flag MESH\_SMOOTH\_SUBSEL, this indicates whether we should display smooth faces with selection-color outlines (TRUE) or transparent shaded faces (FALSE).

**MNDISP\_BEEN\_DISP** - Set when the MNMesh has been displayed (at the end of the render method.)

**MNDISP\_DIAGONALS** – Set when diagonals should be displayed.

**Tab<int> \*vedg;**

This data member is available in release 3.0 and later only.

This is an array of int Tabs that records which edges use each vertex. For instance, if edges 3, 5, and 6 have an endpoint at vertex 0, vedg[0] would include 3, 5, and 6, though not in any particular order. This replaces the 2.5 MNVert::edg data member. Note that this information is only valid if the MN\_MESH\_FILLED\_IN flag is set.

**Tab<int> \*vfac;**

This data member is available in release 3.0 and later only.

This is an array of int Tabs that records which faces touch each vertex. For instance, if faces 0, 1, 3, 8, and 9 use vertex 0 as one of their corners, vfac[0] would include 0, 1, 3, 8, and 9, though not in any particular order. This replaces the 2.5 MNVert::fac data member. Note that this information is only valid if the MN\_MESH\_FILLED\_IN flag is set.

**Flags:**

For more information on flags, see [Class FlagUser](#).

**MN\_MESH\_NONTRI**

If set, this mesh's faces are not all triangles -- at least one face has more than 3 sides.

## **MN\_MESH\_FILLED\_IN**

If set, all topological links, such as the list of edges, are complete.

## **MN\_MESH\_RATSNEST**

Some regular Meshes have more than two faces referencing the same edge, or more than one referencing it in the same direction. These are termed "rats' nest meshes". Since our edge structure only permits 1 or 2 faces (one in each direction), these meshes are unacceptable. Upon conversion, certain vertices & edges are replicated (increasing the vertex count) to separate these into regular, non-rats'-nest parts. If this happens, this flag is set to let you know this change has occurred. In particular, converting this MNMesh back into a regular Mesh will produce a Mesh with more vertices than you started with.

## **MN\_MESH\_NO\_BAD\_VERTS**

This indicates that the mesh has had its vertices checked and "bad" ones eliminated by `EliminateBadVerts` .

## **MN\_MESH\_VERTS\_ORDERED**

This indicates that the mesh has had its vertices ordered by the method `OrderVerts`.

## **MN\_MESH\_HAS\_VOLUME**

This flag is available in release 2.5 or later only.

This mesh has at least one connected component which contains volume, i.e. represents a solid object with no gaps or holes. The flag is set by the

**MNMesh::FindOpenRegions()** method.

## **MN\_USER(1<<16)**

Flag bits at or above MN\_USER are reserved in all MNMesh components for the plug-in developer, if needed. Since FlagUser-derived classes have 32 flag bits, this allows for up to 16 user-defined flags.

## **Methods:**

## **Initialization & Cleanup**

### **Prototype:**

```
MNMesh();
```

### **Remarks:**

Constructor. Initializes the MNMesh with no components and the default

flags.

**Prototype:**

**MNMesh(Mesh &from);**

**Remarks:**

Constructor. Initializes the MNMesh with the mesh "from", and fills in the topology completely (using FillInMesh).

**Prototype:**

**MNMesh(const MNMesh &from);**

**Remarks:**

This method is available in release 3.0 and later only.  
Copy constructor.

**Prototype:**

**~MNMesh();**

**Remarks:**

Destructor. Frees all allocated memory (using Clear).

**Prototype:**

**MNMap \*M(int mp) const;**

**Remarks:**

This method is available in release 4.0 and later only.  
Access to the MNMaps in each MNMesh is made private. Instead of using the MNMap \*m data member, you must now use this accessor method. This accessor now accepts a value in the range **-NUM\_HIDDENMAPS** to **MNum()**.

**Parameters:**

**int mp**

The map channel

**Prototype:**

**void DefaultFlags();**

**Remarks:**

Clears all flags.

**Prototype:**

**void Init();**

**Remarks:**

This method is available in release 3.0 and later only.

The data members are initialized as follows:

**nv\_alloc = ne\_alloc = nf\_alloc = nm\_alloc = 0;**

**numv = nume = numf = numm = 0;**

**v = NULL;**

**e = NULL;**

**f = NULL;**

**m = NULL;**

**vd = NULL;**

**ed = NULL;**

**vedg = NULL;**

**vfac = NULL;**

**bdgBox.Init();**

**fnorm = NULL;**

**dispFlags = 0;**

**rVerts = NULL;**

**cacheGW = NULL;**

**normalsBuilt = 0;**

**norScale = 0.0f;**

**selLevel = MNM\_SL\_OBJECT;**

**Prototype:**

**void VAlloc(int num, bool keep=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.  
Allocates and inits the specified number of MNVerts.

**Parameters:****int num**

The number of verts to allocate.

**bool keep=TRUE**

If TRUE any previous verts are kept; otherwise they are discarded.

**Prototype:**

```
void VShrink(int num=-1);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Shrinks the nv\_alloc size of the MNVert array to the specified size.

**Parameters:****int num=-1**

The new size of the array. The default -1 means to shrink array allocation to numv.

**Prototype:**

```
void freeVEdge();
```

**Remarks:**

This method is available in release 3.0 and later only.  
Deallocates any MNEdges in the **vedg** table and sets the **vedg** pointer to NULL.

**Prototype:**

```
void VEdgeAlloc();
```

**Remarks:**

This method is available in release 3.0 and later only.  
Allocates the **vedg** array.

**Prototype:**

**void freeVFace();**

**Remarks:**

This method is available in release 3.0 and later only.  
Deallocates the **vfac** array.

**Prototype:**

**void VFaceAlloc();**

**Remarks:**

This method is available in release 3.0 and later only.  
Allocates the **vfac** array.

**Prototype:**

**void EAlloc(int num, bool keep=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.  
Allocates the MNEdge data array **e** with the specified size.

**Parameters:**

**int num**

The number of MNEdges to allocate.

**bool keep=TRUE**

If TRUE any previously allocated edges are kept; otherwise they are discarded.

**Prototype:**

**void EShrink(int num=-1);**

**Remarks:**

This method is available in release 3.0 and later only.  
Reduces the ne\_alloc size of the MNEdge data array **e** to the specified number of elements.

**Parameters:**

**int num=-1**

The new size for the array. The value -1 means to use the current number of edges, nume.

**Prototype:**

**void FAlloc(int num, bool keep=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

Allocates the MNFace array **f** with the specified size.

**Parameters:**

**int num**

The number of MNFaces to allocate.

**bool keep=TRUE**

If TRUE any previously allocated faces are kept; otherwise they are discarded.

**Prototype:**

**void FShrink(int num=-1);**

**Remarks:**

This method is available in release 3.0 and later only.

Reduces the nf\_alloc size of the MNFace data array **f** to the specified number of elements.

**Parameters:**

**int num=-1**

The new size for the array. The value -1 means to use the current number of faces, numf.

**Prototype:**

**void MAlloc(int num, bool keep=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

Allocates the MNMap array **m** with the specified size.

**Parameters:**

**int num**

The number of MNMap elements to allocate.

**bool keep=TRUE**

If TRUE any previously allocated MNMaps are kept; otherwise they are discarded.

**Prototype:**

**void MShrink(int num=-1);**

**Remarks:**

This method is available in release 3.0 and later only.

Reduces the nm\_alloc size of the MNMap data array **m** to the specified number of elements.

**Parameters:**

**int num=-1**

The new size for the array. The value -1 means to use the current number of maps, numm.

**Prototype:**

**void PrepForPipeline ();**

**Remarks:**

This method is available in release 4.0 and later only.

This method reconciles flags with arrays, checks and modifies data, ensures that the caches are consistent, and prepares the MNMesh pipeline. For instance, if the **MN\_MESH\_FILLED\_IN** flag is absent but there is still an edge array, this will free the edge array. This method is important to call if a **MNMesh** has been subjected to topology changing operations and should be called at the end of any operation on an MNMesh.

**Prototype:**

**BaseInterface\* GetInterface(Interface\_ID id);**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns a pointer to the base interface of the associated and

provided Interface ID.

**Parameters:**

**Interface\_ID id**

The interface ID.

## **I/O with regular meshes**

**Prototype:**

**void SetFromTri(const Mesh & from);**

**Remarks:**

Clears out all current information, and copies in new faces & vertices from "from".

**Prototype:**

**void AddTri(const Mesh & from);**

**Remarks:**

Adds vertices and faces in "from" to current MNMesh.

**Prototype:**

**void OutToTri(Mesh & tmesh);**

**Remarks:**

Outputs current MNMesh into the mesh given. Note that even if the MNMesh was originally taken from this Mesh, the internal processing may have changed PART\_TOPO, PART\_GEOM, PART\_SELECT, PART\_MAPPING, or PART\_VERTCOLOR.

## **Access to components**

**Prototype:**

**int VNum();**

**Remarks:**

Returns the number of vertices.

**Prototype:**

**MNVert \*V(int i);**

**Remarks:**

Returns a pointer to the i'th MNVert.

**Prototype:**

**Point3 &P(int i);**

**Remarks:**

Returns the point in the i'th MNVert. P(i) is the same as V(i)->p.

**Prototype:**

**int ENum();**

**Remarks:**

Returns the number of edges.

**Prototype:**

**MNEdge \*E(int i);**

**Remarks:**

Returns a pointer to the i'th MNEdge.

**Prototype:**

**int FNum();**

**Remarks:**

Returns the number of faces.

**Prototype:**

**MNFace \*F(int i);**

**Remarks:**

Returns a pointer to the i'th MNFace.

**Prototype:**

**void SetMapNum(int mpnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Allocates and initializes the specified number of MNMap elements in the **m** array. Initializing sets the number of verts and faces in the map to zero and sets the **MN\_DEAD** flag.

**Parameters:**

**int mpnum**

The number of MNMaps to allocate and initialize.

**Prototype:**

**void InitMap(int mp);**

**Remarks:**

This method is available in release 3.0 and later only.

Allocates and initializes basic planar map, or a white map for the vertex color channel.

**Parameters:**

**int mp**

The map channel to initialize:

0: Vertex Color channel.

1: Default mapping channel.

2 through MAX\_MESHMAPS-1: The new mapping channels available in release 3.0.

**Prototype:**

**void ClearMap(int mp);**

**Remarks:**

This method is available in release 3.0 and later only.

Clears and frees the specified map channel, setting the **MN\_DEAD** flag.

**Parameters:**

**int mp**

The map channel to clear.

0: Vertex Color channel.  
1: Default mapping channel.  
2 through MAX\_MESHMAPS-1: The new mapping channels available in release 3.0.

**Prototype:**

**UVVert MV(int mp, int i) const;**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the specified UVVert from the specified mapping channel.

**Parameters:**

**int mp**

The map channel.

0: Vertex Color channel.

1: Default mapping channel.

2 through MAX\_MESHMAPS-1: The new mapping channels available in release 3.0.

**int i**

The zero based index of the UVVert to return.

**Prototype:**

**MNMapFace \*MF(int mp, int i) const;**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns a pointer to the specified MNMapFace from the specified mapping channel.

**Parameters:**

**int mp**

The map channel:

0: Vertex Color channel.

1: Default mapping channel.

2 through MAX\_MESHMAPS-1: The new mapping channels available in

release 3.0.

**int i**

The zero based index of the MNMapFace to return.

**Prototype:**

**int TriNum() const;**

**Remarks:**

Returns the total number of triangles; this is a sum of the number of triangles in each face that does not have the MN\_DEAD flag set.

**Prototype:**

**void setNumVData(int ct, BOOL keep=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the specified number of vertex data elements.

**Parameters:**

**int ct**

The number of vertex data elements to set.

**BOOL keep=FALSE**

If TRUE any previously allocated elements are kept; otherwise they are discarded.

**Prototype:**

**int VNum() const;**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the number of vertex data channels maintained by this MNMesh.

**Prototype:**

**BOOL vDataSupport(int vdChan) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the specified channel of vertex data is available for this MNMesh; otherwise FALSE.

**Parameters:**

**int vdChan**

The vertex data channel. See [List of Vertex Data Index Options](#).

**Prototype:**

**void setVDataSupport(int vdChan, BOOL support=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets if the specified channel of vertex data is supported by this MNMesh.

**Parameters:**

**int vdChan**

The vertex data channel. See [List of Vertex Data Index Options](#).

**BOOL support=TRUE**

TRUE to indicate the channel is supported; FALSE to indicate it's not. If TRUE is specified then elements are allocated (if needed). If FALSE is specified the data for the channel is freed.

**Prototype:**

**void \*vertexData(int vdChan) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the vertex data for the specified channel or NULL if the channel is not supported.

**Parameters:**

**int vdChan**

The vertex data channel. See [List of Vertex Data Index Options](#).

**Prototype:**

**float \*vertexFloat(int vdChan) const;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the floating point vertex data for the specified channel of this mesh or NULL if the channel is not supported.

**Parameters:**

**int vdChan**

The vertex data channel. See [List of Vertex Data Index Options](#).

**Prototype:**

**void freeVData(int vdChan);**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes (deallocates) the vertex data for the specified channel.

**Parameters:**

**int vdChan**

The vertex data channel. See [List of Vertex Data Index Options](#).

**Prototype:**

**void freeAllVData();**

**Remarks:**

This method is available in release 3.0 and later only.

Deallocates the vertex data from all channels and sets the number of supported channels to 0.

**Prototype:**

**float \*getVertexWeights();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the floating point vertex weight data.

**Prototype:**

**void SupportVertexWeights();**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the channel support for the vertex weights channel  
(VDATA\_WEIGHT).

**Prototype:**

**void freeVertexWeights();**

**Remarks:**

This method is available in release 3.0 and later only.  
Frees (deallocates) the vertex weight channel data.

**Prototype:**

**float \*getVSelectionWeights();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns a pointer to the floating point vertex selection weight data.

**Prototype:**

**void SupportVSelectionWeights();**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the channel support for the vertex weights channel  
(VDATA\_SELECT).

**Prototype:**

**void freeVSelectionWeights();**

**Remarks:**

This method is available in release 3.0 and later only.  
Frees (deallocates) the vertex selection weight channel data.

**Prototype:**

**void setNumEData(int ct, BOOL keep=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified number of edge data elements.

**Parameters:****int ct**

The number of edge data elements to set.

**BOOL keep=FALSE**

If TRUE any previously allocated elements are kept; otherwise they are discarded.

**Prototype:**

```
int EDNum() const;
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of edge data channels maintained by this MNMesh.

**Prototype:**

```
BOOL eDataSupport(int edChan) const;
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the specified channel of edge data is available for this MNMesh; otherwise FALSE.

**Parameters:****int edChan**

The edge data channel. See [List of Edge Data Index Options](#).

**Prototype:**

```
void setEDataSupport(int edChan, BOOL support=TRUE);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets if the specified channel of edge data is supported by this MNMesh.

**Parameters:**

## **int edChan**

The edge data channel. See [List of Edge Data Index Options](#).

## **BOOL support=TRUE**

TRUE to indicate the channel is supported; FALSE to indicate it's not.

### **Prototype:**

```
void *edgeData(int edChan) const;
```

### **Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the edge data for the specified channel or NULL if the channel is not supported.

### **Parameters:**

## **int edChan**

The edge data channel. See [List of Edge Data Index Options](#).

### **Prototype:**

```
float *edgeFloat(int edChan) const;
```

### **Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the floating point edge data for the specified channel of this MNMesh or NULL if the channel is not supported.

### **Parameters:**

## **int edChan**

The edge data channel. See [List of Edge Data Index Options](#).

### **Prototype:**

```
void freeEData(int edChan);
```

### **Remarks:**

This method is available in release 3.0 and later only.

Deletes (deallocates) the edge data for the specified channel.

### **Parameters:**

## **int edChan**

The edge data channel. See [List of Edge Data Index Options](#).

**Prototype:**

**void freeAllEData();**

**Remarks:**

This method is available in release 3.0 and later only.

Deallocates the edge data from all channels and sets the number of supported channels to 0.

**Prototype:**

**float \*getEdgeKnots();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the floating point edge knot data.

**Prototype:**

**void SupportEdgeKnots();**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the channel support for the edge knot channel (**EDATA\_KNOT**).

**Prototype:**

**void freeEdgeKnots();**

**Remarks:**

This method is available in release 3.0 and later only.

Frees (deallocates) the edge knot channel data.

**Prototype:**

**void VClear(int vv);**

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old MNVert::Clear method.

Clears and frees the flags and face/edge lists for the specified vertex.

**Parameters:**

**int vv**

The vertex to clear.

**Prototype:**

**void VInit(int vv);**

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old MNVert::Init method.

Initializes the specified MNVert, clearing its flags and emptying its face & edge lists (if vfac & edg are allocated).

**Parameters:**

**int vv**

The vertex to initialize.

**Prototype:**

**int VFaceIndex(int vv, int ff, int ee=-1);**

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old MNVert::FaceIndex method.

Returns the index of face ff in the vfac[vv] table.

NOTE that if this face cannot be found, or if it cannot be found accompanied by edge ee>-1, it will cause an assertion failure.

**Parameters:**

**int vv**

The vertex to check the face list of.

**int ff**

The face to look for.

**int ee=-1**

In cases where the same face touches this vertex more than once, and is therefore represented twice in the vfac table, an optional edge parameter is

used to specify which instance of the face you want the index of. However, if the parent MNMesh doesn't have its vertices ordered (as indicated by the MN\_MESH\_VERTS\_ORDERED flag), the extra edge parameter is meaningless and should not be used.

**Prototype:**

**int VEdgeIndex(int vv, int ee);**

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old MNVert::EdgeIndex method.

Finds the position of edge ee in this MNVert's vedg table. Unlike VFaceIndex (and MNFace's VertIndex and EdgeIndex), each vertex can only touch an edge once, so there's no need for an additional parameter. (There's no such thing as an edge with both ends at the same vertex.)

**Parameters:**

**int vv**

The vertex whose edge list should be searched.

**int ee**

The edge to find.

**Return Value:**

Returns -1 if edge ee is not in the edge table.

**Prototype:**

**void VDeleteEdge(int vv, int ee);**

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old MNVert::DeleteEdge method.

Finds edge ee in the vedg[vv] table and removes it.

**Parameters:**

**int vv**

The vertex from whose edge list the edge should be deleted.

**int ee**

The edge to delete from the vertex's edge list.

**Return Value:**

Returns -1 if edge **ee** is not in the edge table.

**Prototype:**

```
void VDeleteFace(int vv, int ff);
```

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old `MNVert::DeleteFace` method.

Finds face **ff** in the **vfac[vv]** table and removes it. NOTE that this method causes an assertion failure if face **ff** is not in the **vfac** table. If **ff** occurs more than once, which is possible on some valid NONTRI meshes, only the first **ff** is removed.

**Parameters:**

**int vv**

The vertex from whose face list the face should be deleted.

**int ff**

The face to delete from the vertex's face list.

**Prototype:**

```
void VReplaceEdge(int vv, int oe, int ne);
```

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old `MNVert::ReplaceEdge` method.

Finds edge **oe** in the **vedg[vv]** table and replaces it with **ne**. NOTE that this method causes an assertion failure if edge **oe** is not in the **vedg** table.

**Parameters:**

**int vv**

The vertex in whose edge list the edge should be replaced.

**int oe**

The edge to replace.

**int ne**

The replacement edge.

**Prototype:**

**void VReplaceFace(int vv, int of, int nf);**

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old MNVert::ReplaceFace method.

Finds face **of** in the **vfac[vv]** table and replaces it with **nf**. NOTE that this method causes an assertion failure if face **of** is not in the **vfac** table. If **of** occurs more than once, which is possible on some valid NONTRI meshes, only the first of is replaced.

**Parameters:**

**int vv**

The vertex in whose face list the face should be replaced.

**int of**

The face to replace.

**int nf**

The replacement.

**Prototype:**

**void CopyVert(int nv, int ov);**

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old MNVert::operator= by allowing the developer to copy face and edge adjacency information as well as vertex location and flags.

Copies the MNVert data from ov to nv. The face and edge data is copied too if appropriate (ie if MN\_MESH\_FILLED\_IN is set and vfac and vedg are allocated).

**Parameters:**

**int nv**

The destination index.

**int ov**

The source index.

**Prototype:**

```
void MNVDebugPrint(int vv);
```

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old MNVert::MNDebugPrint method.

Uses DebugPrint to print out vertex information to the Debug Results window in DevStudio. The information consists of the position, edge list, and face list. It is generally a good idea to put in a DebugPrint immediately before this with the index of the vertex, so you know which vertex is being printed out:

```
DebugPrint ("Vertex %d: ", vid);  
MNVDebugPrint(vid);
```

**Parameters:**

**int vv**

The zero based index of the MNVert to debug print.

**Prototype:**

```
BOOL SubObjectHitTest(GraphicsWindow *gw, Material *ma,  
HitRegion *hr, DWORD flags, SubObjHitList& hitList, int  
numMat=1 );
```

**Remarks:**

This method is available in release 4.0 and later only.

This method may be called to perform sub-object hit testing on this mesh.

**Parameters:**

**GraphicsWindow \*gw**

The graphics window associated with the viewport the mesh is being hit tested in.

**Material \*ma**

The list of materials for the mesh. See [Class Material](#)

**HitRegion \*hr**

This describes the properties of a region used for the hit testing. See [Class HitRegion](#).

**DWORD flags**

Flags for sub object hit testing. One or more of the following values:

## **SUBHIT\_MNUSECURRENTSEL**

When this bit is set, the sel only and unsel only tests will use the current level (edge or face) selection when doing a vertex level hit test.) This is like the Mesh hit-testing flag SUBHIT\_USEFACESEL.

## **SUBHIT\_MNVERTS**

## **SUBHIT\_MNFACES**

## **SUBHIT\_MNEDGES**

## **SUBHIT\_MNTYPEMASK**

**(SUBHIT\_MNVERTS|SUBHIT\_MNFACES|SUBHIT\_MNEDGE**

## **SubObjHitList& hitList**

The results are stored here. See [Class SubObjHitList](#).

## **int numMat=1**

The number of materials for the mesh.

## **Return Value:**

TRUE if the item was hit; otherwise FALSE.

## **Prototype:**

**void UpdateDisplayVertexColors ();**

## **Remarks:**

This method is available in release 4.0 and later only.

This method is used to manage the display of vertex colors from any channel (or those that are passed in by some calling routine). For instance, to set the mesh to display the Illumination channel as the current vertex colors, you would call **SetDisplayVertexColors (MAP\_SHADING)**. (Normally, it shows the standard vertex color channel, channel 0.)

**UpdateDisplayVertexColors()** is used to refresh the vertex color pointers just before the MNMesh displays itself.

## **Prototype:**

**void SetDisplayVertexColors (int chan);**

## **Remarks:**

This method is available in release 4.0 and later only.

This method is used to manage the display of vertex colors from any channel

(or those that are passed in by some calling routine). For instance, to set the mesh to display the Illumination channel as the current vertex colors, you would call **SetDisplayVertexColors (MAP\_SHADING)**. (Normally, it shows the standard vertex color channel, channel 0.)

**UpdateDisplayVertexColors()** is used to refresh the vertex color pointers just before the MNMesh displays itself.

**Parameters:**

**int chan**

The channel you wish to use.

**Prototype:**

**void SetDisplayVertexColors (UVVert \*mv, MNMapFace \*mf);**

**Remarks:**

This method is available in release 4.0 and later only.

If you have cached your own vertex color information that isn't in any of the map channels provided, you can use this method to set the internal pointers to use your data.

**Parameters:**

**UVVert \*mv**

The array of UV vertices.

**MNMapFace \*mf**

The map face data.

**Prototype:**

**void FillInFaceEdges ();**

**Remarks:**

This method is available in release 4.0 and later only.

Fills in the faces' edge arrays based on the edge list.

**Prototype:**

**void FillInVertEdgesFaces ();**

**Remarks:**

This method is available in release 4.0 and later only.  
Fills in the vertex edge and face lists based on the edge list.

**Prototype:**

**Box3 getBoundingBox (Matrix3 \*tm=NULL, bool targonly=FALSE);**

**Remarks:**

This method is available in release 4.0 and later only.  
Retrieves a bounding box for the MNMesh.

**Parameters:**

**Matrix3 \*tm**

Like the corresponding method in class Mesh, this method takes an optional transform, so the user can get a bounding box in any desired space (with a slower calculation, as all the points must be transformed).

**bool targonly**

If set, only vertices with the **MN\_TARG** flag set are used to compute the bounding box.

**Prototype:**

**void ComputeNormal (int ff, Point3 & N, Point3 \*ctr=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.  
Computes a "balanced" normal, in that the normal takes the contribution of all vertices equally. (This is significant in the case of nonplanar polygons.)

**Parameters:**

**int ff**

The face you want the normal of.

**Point3 & N**

The place to store the computed normal.

**Point3 \*ctr**

If not NULL, it points to a place where ComputeNormal should put the face center.

**Prototype:**

**void GetVertexSpace (int vrt, Matrix3 & tm);**

**Remarks:**

This method is available in release 4.0 and later only.

Chooses a suitable "local space", and sets tm's rotation to match that space.

Does not set tm's translation component. (This is easily done by setting tm.SetRow (3, v[vrt].p).)

The purpose of this method is to support a consistent definition of "local space" around a given vertex. As usual, Z comes from the local normal; the X direction is chosen to be a particular edge, but then this direction is modified to balance the contribution of the other edges. (Thus it points roughly in the direction of one edge, but moving other edges' far endpoints will rotate which way is considered "X" just as much as moving that edge will.)

**Parameters:**

**int vrt**

The vertex index.

**Matrix3 & tm**

The transformation matrix.

**Prototype:**

**float EdgeAngle (int ed);**

**Remarks:**

This method is available in release 4.0 and later only.

Computes the angle at the given edge. (In other words, the angle between the planes of the faces on either side.) Note that in the case of nonplanar polygons, these planes are the "average" planes for the polygon, not the plane of the triangle nearest the edge.

**Parameters:**

**int ed**

The edge index.

**Return Value:**

The angle in radians.

**Prototype:****void FlipNormal(int faceIndex);****Remarks:**

This method is available in release 4.0 and later only.

Flips the normal of the specified face, as well as the corresponding map faces in active maps. Note that doing this on an isolated face with neighbors will cause an illegal condition. Use the FlipElementNormals method to safely and completely flip the normals of entire elements to avoid this problem.

This method uses the MNFace and MNMapFace methods "Flip", which changes the order of the vertices from (0,1,2...,deg-1) to (0, deg-1, deg-2,...1) and rearrange the edge and diagonal information accordingly.

**Parameters:****int faceIndex**

The face index.

**Prototype:****void AutoSmooth (float angle, BOOL useSel, BOOL preventIndirectSmoothing);****Remarks:**

This method is available in release 4.0 and later only.

Applies new smoothing groups to the whole mesh or to selected faces based on angles between faces.

**Parameters:****float angle**

The threshold angle in radians. Edges with angles above this amount will not be smoothed across.

**BOOL useSel**

Indicates if the auto-smoothing should be done only on selected faces.

**BOOL preventIndirectSmoothing**

Sometimes even though two neighboring faces are more than "angle" apart, there may be a path from one to the other via other faces, crossing only edges that are less than "angle" apart, so they'll wind up sharing the same smoothing group anyway. To prevent this sort of "indirect" smoothing, set this value to

TRUE.

**Prototype:**

**void RestrictPolySize (int maxdeg);**

**Remarks:**

This method is available in release 4.0 and later only.

Subdivides polygons as needed so that no polygon has degree larger than maxdeg. (For instance, if maxdeg was 4, an octagon would have edges added until it was composed of 3 quads, or some combination of quads and tris.)

**Parameters:**

**int maxdeg**

Maximum degrees.

**Prototype:**

**void MakePlanar (float planarThresh);**

**Remarks:**

This method is available in release 4.0 and later only.

Makes all faces planar, within the angle threshold given, by subdividing them as necessary. (See MakeFacePlanar.)

**Parameters:**

**float planarThresh**

The planar angle threshold.

**Prototype:**

**void MakeFacePlanar (int ff, float planarThresh);**

**Remarks:**

This method is available in release 4.0 and later only.

Makes the specified face planar by subdividing if necessary. planarThresh represents an angle in radians. If the angle across any of the face's diagonals is larger than this amount, the face is divided on that diagonal.

**Parameters:**

**int ff**

The face of the index to make planar

**float planarThresh**

The planar angle threshold.

**Prototype:**

**UVVert ExtrapolateMapValue (int face, int edge, Point3 & pt, int mp);**

**Remarks:**

This method is available in release 4.0 and later only.

Given a point near a given edge of a given face, but near to it, this method will extrapolate what that point's mapping coordinates would be in the mapping scheme used by the face.

**Parameters:**

**int face**

The face we want to base mapping information on.

**int edge**

The index of the edge we're closest to on the face. (This should be the face-based index, in the range (0, deg-1), not the index of the edge in the MNMesh.)

**Point3 & pt**

The object-space coordinates of the point.

**int mp**

The map channel we're analyzing.

**Prototype:**

**void EliminateDoubledMappingVerts();**

**Remarks:**

This method is available in release 4.0 and later only.

"Doubled" mapping vertices are individual map vertices that are used to correspond to different regular vertices. For instance, a box could have a single (1,1,0) map vertex that it uses in the upper-right corner of all quads. This design is harmful to some of our algorithms, such as the various Tessellators. So this method is available to fix such vertices. It clones map

vertices as needed to ensure that a given map vertex is only used by one regular vertex. Linear-time algorithm.

**Prototype:**

**BOOL CheckForDoubledMappingVerts();**

**Remarks:**

This method is available in release 4.0 and later only.

This method is a debugging tool. All doubled mapping verts are DebugPrinted. Return value is TRUE if there is at least one doubled mapping vertex (on at least one map channel). FALSE if the mesh is clean. Note that this method is not significantly faster than EliminateDoubledMappingVerts, so you should not use it to determine if EliminateDoubledMappingVerts should be called.

**Prototype:**

**void EliminateIsoMapVerts();**

**Remarks:**

This method is available in release 4.0 and later only.

Deletes isolated mapping vertices - i.e., mapping vertices that aren't used by any mapping faces.

**Prototype:**

**void EliminateIsoMapVerts(int mp);**

**Remarks:**

This method is available in release 4.0 and later only.

Deletes isolated mapping vertices - i.e., mapping vertices that aren't used by any mapping faces. If the "int mp" parameter is given, the algorithm operates only on that map channel. If not, the algorithm operates on all map channels.

**Parameters:**

**int mp**

The map channel to operate on.

**Prototype:**

**int SplitEdge (int ee, float prop, Tab<int> \*newTVerts);**

**Remarks:**

This method is available in release 4.0 and later only.

This new SplitEdge variant allows you to recover information about the new map vertices created at the point where the edge is split. Everything else is the same as the existing SplitEdge (int ee, float prop=.5f) method.

**Parameters:**

**Tab<int> \*newTVerts**

Pointer to a table in which the new map vertices can be stored. This table is set to size (NUM\_HIDDENMAPS + numm)\*2. The two entries for each map channel are used to store the two new map vertices at this edge split: newTVerts[(NUM\_HIDDENMAPS+mp)\*2+0] is the map vertex for map channel mp on the "f1" side of the edge, and newTVerts[mp\*2+1] is the map vertex for the "f2" side if f2>=0 (or otherwise left uninitialized). (These values are often, but not always, the same. They are different if the map has a seam along this edge.)

**Prototype:**

**void FacePointBary (int ff, Point3 & p, Tab<float> & bary);**

**Remarks:**

This method is available in release 4.0 and later only.

Finds "Generalized Barycentric Coordinates" for the point given. Generalized barycentric coordinates are not uniquely determined for polygons of degree greater than 3, but this algorithm should find a reasonable balance, where for instance a point in the center of a polygon would have a significant contribution from all vertices.

Generalized barycentric coordinates are a set of floats, one per vertex in the polygon, such that the sum of all the floats is 1, and the sum of all the floats times the corresponding vertices comes out to the point given.

**Parameters:**

**int ff**

The face we're finding barycentric coordinates on.

**Point3 & p**

The point we're trying to find barycentric coordinates for. If this point is not in

the plane of the polygon, the coordinates produced should represent its projection into the polygon's plane. Points outside the boundary of the polygon should be acceptable; some of the barycentric coordinates will be negative in this case.

**Tab<float> & bary**

The table to put the results in. This table is set to size f[ff].deg.

**Prototype:**

```
void CloneVerts (DWORD cloneFlag = MN_SEL, bool  
clear_orig=TRUE);
```

**Remarks:**

This method is available in release 4.0 and later only.

Clones flagged vertices, creating new vertices that aren't used by any faces.

**Parameters:**

**DWORD cloneFlag = MN\_SEL**

Indicates which vertices should be cloned.

**bool clear\_orig = TRUE**

If true, the original vertices should have the cloneFlag cleared. (The clones will always have this flag set.)

**Prototype:**

```
void CloneFaces (DWORD cloneFlag = MN_SEL, bool  
clear_orig=TRUE);
```

**Remarks:**

This method is available in release 4.0 and later only.

Clones the flagged faces, as well as all the vertices and edges that are used by the faces.

**Parameters:**

**DWORD cloneFlag = MN\_SEL**

Indicates which faces should be cloned.

**bool clear\_orig = TRUE**

If true, the original faces should have the cloneFlag cleared. (The clones will always have this flag set.)

**Prototype:****int DivideFace (int ff, Tab<float> & bary);****Remarks:**

This method is available in release 4.0 and later only.

Divides a face by creating a point in the face's interior and creating an edge from that point to each of the face's vertices. An n-gon becomes n triangles by this method.

**Parameters:****int ff**

The face to divide.

**Tab<float> & bary**

The generalized barycentric coordinates of the point that should be created. (See FacePointBary for details on barycentric coordinates.)

**Return Value:**

The index of the newly created vertex, or -1 if there's an error.

**Prototype:****int CreateFace (int degg, int \*vv);****Remarks:**

This method is available in release 4.0 and later only.

Creates a new face, using the vertices given.

Note that this method, unlike the similar NewFace method, maintains all topological links. If there's an edge between vv[0] and vv[1] which is already used by some other face (on the other side), that edge is modified to use this face on its "f2" side. If there's an edge between two sequential vertices with faces on both sides, the creation fails, because it would create an illegal condition.

This method also creates map faces in any active map channels to maintain the validity of the mesh. These map faces use the map vertices that other faces use for the vertices passed in, or if there are no corresponding map vertices yet, use newly created map vertices with the value (1,1,1).

**Parameters:****int degg**

The degree of the new face.

**int \*vv**

A pointer to an array of degg vertices for the new face.

**Return Value:**

The index of the new face, or -1 if the method was unable to create that face.

**Prototype:**

**bool MakeFlaggedPlanar (int selLev, DWORD flag=MN\_SEL, Point3 \*delta=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

Moves the flagged components into their "average plane". (Plane computed using average component positions and normals.)

**Parameters:**

**int selLev**

Selection level, one of MNM\_SL\_VERTEX, MNM\_SL\_EDGE, MNM\_SL\_FACE, or MNM\_SL\_OBJECT.

**DWORD flag=MN\_SEL**

The flag that indicates which components to align. Ignored if msl is set to MNM\_SL\_OBJECT.

**Point3 \*delta=NULL**

If non-NULL, this is presumed to point to an array of size equal to MNMesh::numv, and instead of actually moving the vertices, the algorithm stores offsets in this array such that  $v[i].p + delta[i]$  is in the plane.

**Return Value:**

Indicates whether anything was moved. (Or if delta is non-NULL, if any deltas are nonzero.)

**Prototype:**

**bool MoveVertsToPlane (Point3 & norm, float offset, DWORD flag=MN\_SEL, Point3 \*delta=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

Projects the flagged vertices into the specified plane.

**Parameters:**

**Point3 & norm, float offset**

The definition of the plane: DotProd (norm, x) - offset = 0.

**DWORD flag=MN\_SEL**

The flag that indicates which vertices to move.

**Point3 \*delta=NULL**

If non-NULL, this is presumed to point to an array of size equal to MNMesh::numv, and instead of actually moving the vertices, the algorithm stores offsets in this array such that  $v[i].p + delta[i]$  is in the plane.

**Return Value:**

Indicates whether anything was moved. (Or if delta is non-NULL, if any deltas are nonzero.)

**Prototype:**

**bool SplitFlaggedVertices (DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

Splits the flagged vertices into a clone for each face using the vertex. For example, if used on the front top left corner of a standard 3ds max box, it splits the vertex into 3, one for the front face, one for the top face, and one for the left face.

**Parameters:**

**DWORD flag=MN\_SEL**

The flag that indicates which vertices to split.

**Return Value:**

TRUE if anything happened, FALSE if none of the vertices were split. (Note that this method will return FALSE if no vertices are flagged, but also if there are flagged vertices but they're all on 1 or 0 faces already.)

**Prototype:**

**bool SplitFlaggedEdges (DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

"Splits" edges by breaking vertices on two or more flagged edges into as many copies as needed. In this way, any path of flagged edges becomes an two open seams.

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which edges should be split. (Left at the default, selected edges are split.)

**Return Value:**

True if any topological changes happened, false if nothing happened.

**Prototype:**

**bool DetachFaces (DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

Detaches specified faces to a separate element, cloning vertices and edges as necessary on the boundary between flagged and unflagged faces.

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which edges should be split. (Left at the default, selected edges are split.)

**Return Value:**

True if any faces were detached, false if nothing happened. (Note that the algorithm will return false if, for instance, all flagged faces form elements which are already distinct from those formed from non-flagged faces.)

**Prototype:**

**bool DetachElementToObject (MNMESH & nmesh, DWORD fflags=MN\_SEL, bool delDetached=true);**

**Remarks:**

This method is available in release 4.0 and later only.

Detaches specified faces to a new MNMesh.

**Parameters:****MNMesh & mesh**

An empty new mesh into which the detached faces (and accompanying vertices and edges) should be put.

**DWORD fflags=MN\_SEL**

Indicates which faces should be detached.

**Return Value:**

True if any faces were detached, false if nothing happened.

**Prototype:**

**bool ExtrudeFaceClusters (MNFaceClusters & fclust);**

**Remarks:**

This method is available in release 4.0 and later only.

Performs the topological component of an extrusion on all face clusters. Each cluster is "extruded", which means that vertices and edges on the boundary of the cluster are cloned and new faces and edges are created to connect the clones to the originals. (Note that nothing is moved in this process - movement is handled separately. See GetExtrudeDirection for details.)

**Parameters:****MNFaceClusters & fclust**

A list of face clusters. See the constructors in class MNFaceClusters and the face cluster related methods of class MNTempData for information on how to set up these clusters based on edge angles and face flags.

**Return Value:**

True if any faces were extruded, false if nothing happened.

**Prototype:**

**bool ExtrudeFaceCluster (MNFaceClusters & fclust, int cl);**

**Remarks:**

This method is available in release 4.0 and later only.

Performs the topological component of an extrusion on the specified face cluster. This means that vertices and edges on the boundary of the cluster are cloned and new faces and edges are created to connect the clones to the

originals. (Note that nothing is moved in this process - movement is handled separately. See GetExtrudeDirection for details.)

**Parameters:**

**MNFaceClusters & fclust**

A list of face clusters. See the constructors in class MNFaceClusters and the face cluster related methods of class MNTempData for information on how to set up these clusters based on edge angles and face flags.

**int cl**

The cluster we wish to extrude.

**Return Value:**

True if any faces were extruded, false if nothing happened.

**Prototype:**

**bool ExtrudeFaces (DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

Performs the topological component of an extrusion on each flagged face individually. (This differs from ExtrudeFaceClusters in that each face is treated like its own cluster.) Extrusion means that the vertices and edges used by each flagged face are cloned, and new faces and edges are created on the "sides" to connect the clones to their originals. (Note that nothing is moved in this process - movement is handled separately. See GetExtrudeDirection for details.)

**Parameters:**

**DWORD flag=MN\_SEL**

The flag that identifies the faces we wish to extrude.

**Return Value:**

True if any faces were extruded, false if nothing happened.

**Prototype:**

**void GetExtrudeDirection (MNChamferData \*mcd,  
MNFaceClusters \*fclust=NULL, Point3 \*clustNormals=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

Finds the direction vectors for the geometric component of an extrusion after the topological component has been completed. (See methods `ExtrudeFaceClusters`, `ExtrudeFaceCluster`, and `ExtrudeFaces` for details on the topological component.)

**Parameters:**

**MNChamferData \*mcd**

The data structure in which the extrusion directions are stored. (Note that there is no map support for this operation, as there is no well-defined way to modify mapping values during an extrusion drag.)

**MNFaceClusters \*fclust = NULL**

The face clusters.

**Point3 \*clustNormals = NULL**

The cluster normals.

This information is only needed if we're extruding by cluster normals. If we're extruding clusters by local normals or just extruding faces separately, these parameters can be left at NULL. See [class MNTempData](#) for handy methods to obtain cluster normals.

**Prototype:**

**bool SetVertColor (UVVert clr, int mp, DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

Sets vertex colors for the specified vertices. This is done by finding all map vertices in the specified vertex color channel that correspond to each flagged vertex and setting them to the color given.

**Parameters:**

**UVVert clr**

The color to set the vertices to.

**int mp**

The map channel - use 0 for the standard vertex color channel, `MAP_SHADING` for the vertex illumination channel, or `MAP_ALPHA` for the alpha channel. (Note that alpha color values should always be shades of grey - `clr.r` should equal `clr.g` and `clr.b`.)

**DWORD flag=MN\_SEL**

Indicates which vertices to modify the colors of.

**Return Value:**

Returns true if any vertex colors were modified.

**Prototype:**

**bool SetFaceColor (UVVert clr, int mp, DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

Sets vertex colors for the specified faces. This is done by finding all map vertices used by flagged faces and setting them to the color given. In cases where a map vertex is used by both a flagged and an unflagged face, the map vertex is split so that the unflagged faces' colors are unaffected by this change.

**Parameters:**

**UVVert clr**

The color to set the faces to.

**int mp**

The map channel - use 0 for the standard vertex color channel, MAP\_SHADING for the vertex illumination channel, or MAP\_ALPHA for the alpha channel. (Note that alpha color values should always be shades of grey - clr.r should equal clr.g and clr.b.)

**DWORD flag=MN\_SEL**

Indicates which faces to modify the colors of.

**Return Value:**

Returns true if any vertex colors were modified.

**Prototype:**

**bool ChamferVertices (DWORD flag=MN\_SEL, MNChamferData \*mcd=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

Performs the topological component of a vertex chamfer on the flagged vertices, and provides the data needed to do the geometric component. That is

to say, this method clones the flagged vertices and creates the new edges and faces needed in a chamfer operation. It also determines the direction each vertex and mapping vertex will go as the user drags out the chamfer - but it doesn't actually move any vertices.

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which vertices to chamfer.

**MNChamferData \*mcd=NULL**

If non-NULL, this points to a data structure which should be filled with information needed to perform the geometric component of the chamfer, such as vertex directions and limits. See [class MNChamferData](#) for additional information.

**Return Value:**

True if any vertices were chamfered, false otherwise.

**Prototype:**

**bool ChamferEdges (DWORD flag=MN\_SEL, MNChamferData \*mcd=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

Performs the topological component of an edge chamfer on the flagged edges, and provides the data needed to do the geometric component. That is to say, this method clones the flagged edges and creates the new vertices and faces needed in the edge chamfer operation. It also determines the direction each vertex and mapping vertex will go as the user drags out the chamfer - but it doesn't actually move anything.

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which edges to chamfer.

**MNChamferData \*mcd=NULL**

If non-NULL, this points to a data structure which should be filled with information needed to perform the geometric component of the chamfer, such as vertex directions and limits. See [class MNChamferData](#) for additional information.

**Return Value:**

True if any edges were chamfered, false otherwise.

**Prototype:**

**bool FlipElementNormals (DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

Flips the normals of the specified elements in the mesh.

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which elements should be flipped, in the following way: any element that has at least one flagged face is completely flipped. Elements without any flagged faces are not.

**Return Value:**

Returns true if anything was flipped, false otherwise.

**Prototype:**

**void SmoothByCreases (DWORD creaseFlag);**

**Remarks:**

This method is available in release 4.0 and later only.

This is an auto-smooth algorithm that allows the developer to specify exactly which edges should be creases and which should be smoothed across. All face smoothing groups are rewritten by this algorithm. This algorithm is used, for example, in MeshSmooth, NURMS style, when the user turns on "smooth result" and applies crease values to some edges.

**Parameters:**

**DWORD creaseFlag**

Indicates which edges should be treated as creases. Edges that have the flag (or flags) set should be creases. Those that don't should not be creases.

**Prototype:**

**int CutFace (int f1, Point3 & p1, Point3 & p2, Point3 & Z, bool split);**

**Remarks:**

This method is available in release 4.0 and later only.

Implements the Editable Poly Cut algorithm from the face level - cuts from a point on one face to a point on another face.

**Parameters:****int f1**

The starting face of the Cut.

**Point3 & p1**

The starting point of the Cut, which should lie on face f1.

**Point3 & p2**

The end point of the Cut.

**Point3 & Z**

The view direction. All Cut algorithms require a view direction to establish the plane that the cut occurs in. This plane is defined by this Z vector and by the vector p2-p1. All new vertices created by the cut are in this plane.

**bool split**

If true, the faces on the top and bottom of the cut should have an open seam between them. All the edges and vertices along the cut (except for the first and last vertex) are split into two parts, one copy for the top and one for the bottom.

**Return Value:**

The last vertex created by the cut, or -1 if the cut was unable to finish.

**Prototype:**

```
int CutEdge (int e1, float prop1, int e2, float prop2, Point3 & Z,  
bool split);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implements the Editable Poly Cut algorithm from the edge level - cuts from a point on one edge to a point on another edge.

**Parameters:****int e1**

The starting edge of the Cut.

**float prop1**

The proportion along edge e1 where the cut should begin. That is, the first point should be located at  $(1-prop1)*v[e[e1].v1].p + prop1*v[e[e1].v2].p$ .

**int e2**

The ending edge of the Cut.

**float prop2**

The proportion along edge e2 where the cut should end. That is, the last point should be located at  $(1-prop2)*v[e[e2].v1].p + prop2*v[e[e2].v2].p$ .

**Point3 & Z**

The view direction. All Cut algorithms require a view direction to establish the plane that the cut occurs in. This plane is defined by this Z vector and by the vector between the start and end points. All new vertices created by the cut are in this plane.

**bool split**

If true, the faces on the top and bottom of the cut should have an open seam between them. All the edges and vertices along the cut (except for the first and last vertex) are split into two parts, one copy for the top and one for the bottom.

**Return Value:**

The last vertex created by the cut, or -1 if the cut was unable to finish.

**Prototype:**

**int Cut (int startv, Point3 & end, Point3 & Z, bool split);**

**Remarks:**

This method is available in release 4.0 and later only.

Implements the Editable Poly Cut algorithm from the vertex level - cuts from one vertex to another.

**Parameters:****int startv**

The starting vertex of the Cut.

**Point3 & end**

The location of the end vertex of the Cut.

**Point3 & Z**

The view direction. All Cut algorithms require a view direction to establish the plane that the cut occurs in. This plane is defined by this Z vector and by the vector between the start and end points. All new vertices created by the cut are in this plane.

**bool split**

If true, the faces on the top and bottom of the cut should have an open seam between them. All the edges and vertices along the cut (except for the first and last vertex) are split into two parts, one copy for the top and one for the bottom.

**Return Value:**

The last vertex created by the cut, or -1 if the cut was unable to finish.

**Prototype:**

**bool WeldBorderVerts (int v1, int v2, Point3 \*destination);**

**Remarks:**

This method is available in release 4.0 and later only.

Welds the specified border vertices together.

**Parameters:**

**int v1, int v2**

The vertices to be welded. They must be border vertices, in the sense that each of them must be used by open edges (those on only one face).

**Point3 \*destination**

If non-NULL, this indicates where the joined vertex should be located. (If NULL, it's put at the average location,  $(v[v1].p + v[v2].p)/2$ .)

**Return Value:**

True if anything was welded, false if the operation could not proceed.

**Prototype:**

**bool WeldBorderEdges (int e1, int e2);**

**Remarks:**

This method is available in release 4.0 and later only.

Welds the specified border edges together.

**Parameters:**

**int e1, int e2**

The edges to be welded. They must be border edges, in the sense that each of them must be open (used by only one face). The result is located where edge e2 was. e[e1].v1 is joined to e[e2].v2, and e[e1].v2 is joined to e[e2].v1.

**Return Value:**

True if anything was welded, false if the operation could not proceed.

**Prototype:**

**bool WeldBorderVerts (float thresh, DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

Welds the specified border vertices together.

**Parameters:**

**float thresh**

The welding threshold. Vertices further apart than this distance (in object space) will not be welded.

**DWORD flag=MN\_SEL**

Indicates which vertices should be welded. (Non-border vertices, those in the "interior" of the surface, are ignored even if flagged.)

**Return Value:**

True if anything was welded, false otherwise.

## **Adding new components**

NOTE: all face creation methods clear the MN\_MESH\_FILLED\_IN, MN\_MESH\_NO\_BAD\_VERTS, and MN\_MESH\_VERTS\_ORDERED flags. If your calling routine takes care of the work of updating or creating all relevant MNVertices and MNEdges, you may be able to set these flags again -- but be careful! For some convenient face & edge subdivision methods that preserve a complete topology, check out the [Splitting](#) methods.

**Prototype:**

**int NewTri(int a, int b, int c, DWORD smG=0, MtlID mt=0);**

**Remarks:**

Creates a new tri-face. Note that no mapping coords or vertex colors can be specified.

**Parameters:**

**int a,b,c**

The indices of the vertices that form this triangle.

**DWORD smG=0**

The smoothing group(s) assigned to the new face.

**MtlID mt=0**

The material ID assigned to the new face.

**Return Value:**

Returns the index of the new face created.

**Prototype:**

**int NewTri(int \*vv, int \*tt, int \*cc, DWORD smG=0, MtlID mt=0);**

**Remarks:**

Creates a new face of degree 3. Edge selection and visibility flags are set to the default: all visible and not selected.

**Parameters:**

**int \*vv**

The indices of the vertices that form this face. (There must be 3 of these.)

**int \*tt**

If this is not NULL, it points to the indices of 3 mapping coordinates for this face.

**int \*cc**

If this is not NULL, it points to the indices of 3 vertex colors for this face.

**DWORD smG**

The smoothing group(s) assigned to this face.

**MtlID mt**

The material ID assigned to this face

**Return Value:**

The index of the face created.

**Prototype:**

```
int NewQuad(int a, int b, int c, int d, DWORD smG=0, MtlID  
mt=0);
```

**Remarks:**

This method will create a new quad.

Previous to 4.0 this method used two tri-faces that shared and invisible edge.

**Parameters:**

**int a, b, c, d**

The indices of the vertices that form this quad.

**DWORD smG=0**

The smoothing group(s) assigned to the new faces.

**MtlID mt=0**

The material ID assigned to the new faces.

**Return Value:**

Returns the index of the quad face created.

**Prototype:**

```
int NewQuad(int *vv, int *tt, int *cc, DWORD smG=0, MtlID  
mt=0);
```

**Remarks:**

This method will create a new quad.

Previous to 4.0 this method used two tri-faces that shared and invisible edge.

**Parameters:**

**int \*vv**

The indices of the vertices that form this face. (There must be 4 of these.)

**int \*tt**

If this is not NULL, it points to the indices of 4 mapping coordinates for this face.

**int \*cc**

If this is not NULL, it points to the indices of 4 vertex colors for this face.

**DWORD smG**

The smoothing group(s) assigned to this face.

**MtlID mt**

The material ID assigned to this face

**Return Value:**

Returns the index of the quad face created.

**Prototype:**

```
int NewFace(MNFace *ff, int degg=0, int *vv=NULL, bool  
*vis=NULL, bool *sel=NULL);
```

**Remarks:**

This method creates a (single) new face with the characteristics given. The default triangulation for a face of this degree is used; if the face is not convex, this triangulation may be inappropriate. If this is the case, call

**RetriangulateFace()** on this face after it's created.

**Parameters:****MNFace \*ff**

A current face from which smoothing groups, material ID, and flags should be copied. If this is NULL, these values are left at their default values.

**int degg**

The degree of the face to be created.

**int \*vv**

The indices of the vertices that form the new face. (There must be degg of these.)

**bool \*vis**

This is an array of visibility bits for the edges of the new face. If this is NULL, the default of all edges being visible is used. If this is not NULL, there must be degg values.

**bool \*sel**

This is an array of selection bits for the edges of the new face. If this is NULL, the default of all edges not being selected is used. If this is not NULL, there must be degg values.

**Prototype:**

```
int AppendNewFaces(int nfname);
```

**Remarks:**

This method is available in release 3.0 and later only.

Appends the specified number of MNFaces to **f**.

**Parameters:**

**int nfnnum**

The number of MNFaces to append.

**Return Value:**

The index of the first appended face (ie the old numf).

**Prototype:**

**void setNumFaces(int nfnnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified number of MNFaces allocated in **f**.

**Parameters:**

**int nfnnum**

The number of MNFaces to set.

**Prototype:**

**int RegisterEdge(int v1, int v2, int f, int fpos);**

**Remarks:**

Edge creation tool. If there is no edge currently joining vertices v1 and v2, it creates such an edge. If there is an edge starting at v2 and ending on v1, it registers face f as being on the "other side" of the edge.

**Parameters:**

**int v1, v2**

The start & end vertices of the edge you wish to register.

**int f**

The face on the "left" side of this edge, if you're looking from v1 towards v2 with the surface normal above.

**int fpos**

The index of this edge in face f. This is used to extract visibility and selection

information from the face.

**Return Value:**

The index of the new edge, or -1 if the edge already exists in the specified direction.

**Prototype:**

**int SimpleNewEdge(int v1, int v2);**

**Remarks:**

Edge creation tool. Simply makes a new edge from v1 to v2, without worrying about whether such an edge may already exist, or what faces may be on either side. Since edges are required to have at least one valid face on them, using this method obligates the developer to assign f1 on the new edge themselves.

**Return Value:**

The index of the new edge.

**Prototype:**

**int NewEdge(int v1, int v2, int f, int fpos);**

**Remarks:**

Edge creation tool. Requires the developer to previously ascertain that there is no edge from v1 to v2 or from v2 to v1.

**Parameters:**

**int v1, v2**

The start & end vertices of the edge you wish to create.

**int f**

The face on the "left" side of this edge, if you're looking from v1 towards v2 with the surface normal above.

**int fpos**

The index of this edge in face f. This is used to extract visibility and selection information from the face.

**Return Value:**

The index of the new edge.

**Prototype:**

**int AppendNewEdges(int nenum);**

**Remarks:**

This method is available in release 3.0 and later only.  
Appends the specified number of edges.

**Parameters:**

**int nenum**

The number of edges to append.

**Prototype:**

**void setNumEdges(int nenum);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the number of MNEdges allocated in **e**.

**Parameters:**

**int nenum**

The number of MNEdges to set.

**Prototype:**

**int NewVert(Point3 &p);**

**Remarks:**

This method is available in release 3.0 and later only.  
Creates a new vertex (increasing numv) and sets it to the specified point.

**Parameters:**

**Point3 &p**

The point to which the new vert should be initialized.

**Prototype:**

**int NewVert(Point3 &p, int vid);**

**Remarks:**

This method is available in release 3.0 and later only. It replaces the old  
MNMesh::NewVert (Point3 &p, MNVert \*mv=NULL) method.

Creates a new vertex (increasing numv) and initializes it to the point p and the flags and other characteristics of vertex vid.

**Parameters:**

**Point3 &p**

The point to which the new vert should be initialized.

**int vid**

The index of the existing MNVert from which flags and PerData info should be copied. (Only the MN\_SEL and MN\_TARG flags are copied.)

**Prototype:**

**int NewVert(int vid);**

**Remarks:**

This method is available in release 3.0 and later only.

Creates a new vertex and initializes it to location, flags and other characteristics of vertex vid.

**Parameters:**

**int vid**

The index of the existing MNVert from which location, flags and PerData info should be copied. (Only the MN\_SEL and MN\_TARG flags are copied.)

**Prototype:**

**int NewVert(int v1, int v2, float prop);**

**Remarks:**

This method is available in release 3.0 and later only.

Creates a new vertex which is a linear combination of two existing vertices. The new vertex has the MN\_SEL and MN\_TARG flags of whichever vertex it's closest to. The location and PerData info is interpolated.

**Parameters:**

**int v1**

The first vertex to combine.

**int v2**

The second vertex to combine.

**float prop**

The proportion along the segment from v1 to v2 where the new vertex should be located.

**Prototype:**

```
int AppendNewVerts(int nvnum);
```

**Remarks:**

This method is available in release 3.0 and later only.

Appends the specified number of MNVerts.

**Parameters:**

**int nvnum**

The number of MNVerts to append.

**Return Value:**

The index of the first appended vertex (ie the old numv).

**Prototype:**

```
void setNumVerts(int nvnum);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the number of verts, allocating if needed.

**Parameters:**

**int nvnum**

The desired number of vertices in the mesh.

## **Removing & Deleting Components**

**Prototype:**

```
void CollapseDeadVerts();
```

**Remarks:**

Removes all MNVerts with the MN\_DEAD flag from the list of vertices. Also, it re-indexes all the faces' and edges' vertex references to maintain mesh integrity.

**Prototype:****void CollapseDeadEdges();****Remarks:**

Removes all MNEdges with the MN\_DEAD flag from the list of edges. Also, re-indexes all the faces' and vertices' edge references to maintain mesh integrity.

**Prototype:****void CollapseDeadFaces();****Remarks:**

Removes all MNFaces with the MN\_DEAD flag from the list of faces. Also, re-indexes all the edges' and vertices' face references to maintain mesh integrity.

**Prototype:****void CollapseDeadStructs();****Remarks:**

Performs all 5 of the above collapse functions, safely removing all unused components from this mesh.

**Prototype:****void Clear();****Remarks:**

Reinitializes all verts, faces, and edges, freeing the data members of these components, but not freeing the vertex, edge, and face arrays themselves. This option is suitable if you need to clear a MNMesh you will be reusing. numv, etc, are set to 0.

**Prototype:****void ClearAndFree();****Remarks:**

This method is available in release 3.0 and later only.

Deletes everything and frees all relevant memory. Leaves you with an empty

MNMesh with the default flags.

**Prototype:**

**void freeVerts();**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes the MNVert array and frees any corresponding vertex data.

**Prototype:**

**void freeEdges();**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes the MNEdge array and frees and corresponding edge data.

**Prototype:**

**void freeFaces();**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes the MNFace array.

**Prototype:**

**void freeMap(int mp);**

**Remarks:**

This method is available in release 3.0 and later only.

Deletes the MNMap on the specified map channel.

**Parameters:**

**int mp**

The map channel.

0: Vertex Color channel.

1: Default mapping channel.

2 through MAX\_MESHMAPS-1: The new mapping channels available in release 3.0.

**Prototype:****void freeMaps();****Remarks:**

This method is available in release 3.0 and later only.

Deletes and frees all the MNMaps.

**Prototype:****void DeleteFlaggedFaces(DWORD deathflags, DWORD  
nvCopyFlags=0x0);****Remarks:**

This method is available in release 2.5 and later only.

Deletes faces with any of the death flags set, as well as delete vertices and edges surrounded by faces with death flags, and correct the mesh components remaining.

**Parameters:****DWORD deathflags**

The collection of flags marking the faces you wish to kill.

**DWORD nvCopyFlags=0x0**

If the NO\_BAD\_VERTS flag is set on this mesh, DeleteFlaggedFaces will preserve this property. This may involve duplicating some vertices. (See EliminateBadVerts for more information.) If you have vertex flags that you want preserved in this duplication, indicate them in nvCopyFlags. MN\_SEL and MN\_TARG are always copied, but all other flags are cleared on the new vertex.

**Prototype:****void SetDispFlag(DWORD f);****Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the display flags.

**Parameters:****DWORD f**

The following flags are supported;

### **MNDISP\_VERTTICKS**

Displays vertices with tick-marks (plus signs).

### **MNDISP\_SELVERTS**

Displays selected vertices in red (and soft-selected vertices in soft selection colors).

### **MNDISP\_SELFACES**

Displays selected faces.

### **MNDISP\_SELEDGES**

Displays selected edges.

### **MNDISP\_NORMALS**

Displays face normals on selected faces.

### **MNDISP\_DIAGONALS**

Displays diagonals (using the same drawing style as regular Meshes use for hidden edges).

#### **Prototype:**

**DWORD GetDispFlag(DWORD f);**

#### **Remarks:**

This method is available in release 4.0 and later only.

This method returns the **displFlags & f**.

#### **Parameters:**

**DWORD f**

The following flags are supported;

### **MNDISP\_VERTTICKS**

Displays vertices with tick-marks (plus signs).

### **MNDISP\_SELVERTS**

Displays selected vertices in red (and soft-selected vertices in soft selection colors).

### **MNDISP\_SELFACES**

Displays selected faces.

### **MNDISP\_SELEDGES**

Displays selected edges.

### **MNDISP\_NORMALS**

Displays face normals on selected faces.

### **MNDISP\_DIAGONALS**

Displays diagonals (using the same drawing style as regular Meshes use for hidden edges).

#### **Prototype:**

**void ClearDispFlag(DWORD f);**

#### **Remarks:**

This method is available in release 4.0 and later only.

This method allows you to clear the specified display flags.

#### **Parameters:**

##### **DWORD f**

The following flags are supported;

##### **MNDISP\_VERTTICKS**

Displays vertices with tick-marks (plus signs).

##### **MNDISP\_SELVERTS**

Displays selected vertices in red (and soft-selected vertices in soft selection colors).

##### **MNDISP\_SELFACES**

Displays selected faces.

##### **MNDISP\_SELEDGES**

Displays selected edges.

##### **MNDISP\_NORMALS**

Displays face normals on selected faces.

##### **MNDISP\_DIAGONALS**

Displays diagonals (using the same drawing style as regular Meshes use for hidden edges).

### **Internal computation**

#### **Prototype:**

**void FillInMesh();**

#### **Remarks:**

If this mesh does not have the `MN_MESH_FILLED_IN` flag, this method completely recomputes all combinatorial information. It re-creates all `MNEdges` and `MNFace::edg`, `MNVert::edg`, and `MNVert::fac` lists based on the information in the `MNFace::vtx` lists.

Since this routine completely reconstructs the combinatorics, it clears the `MN_MESH_VERTS_ORDERED` flag.

**Prototype:**

**bool EliminateBadVerts (DWORD flag=0);**

**Remarks:**

A "bad" vertex in this context is one which is shared between two distinct boundaries for this mesh. As an example, imagine a union between two circles, converted to meshes, that touch at a single vertex. This causes a vertex to exist which is on two faces, but four edges. Since most vertices are on equal numbers of faces and edges (if they're not on a boundary) or on one more edge than face (if they are on a boundary), these types of vertices can mess up some forms of processing. This method eliminates such vertices by duplicating them, assigning one vertex to each boundary. A `MNMesh` that has gone through this method will have the `MN_MESH_NO_BAD_VERTS` flag set until a method (such as `NewFace`) is called that could conceivably create bad vertices.

`EliminateBadVerts` requires a filled in mesh, and will call `FillInMesh` if the `MN_MESH_FILLED_IN` flag is not set.

**Parameters:**

**DWORD flag:**

This parameter is available in release 4.0 and later only.

If nonzero, it indicates that only flagged vertices should be split up if "bad".

**Return Value:**

False if nothing changed, true if at least one bad vertex was found and split.

**Prototype:**

**void OrderVert (int vid);**

**Remarks:**

This method is available in release 4.0 and later only.

This routine organizes the face and edge lists of the specified vertex such that going counterclockwise around the vertex (with the surface normal pointing towards you), you'll encounter `edg[0]`, `fac[0]`, `edg[1]`, `fac[1]`, etc, ending in either the last face if the vertex is not on a boundary or the last edge if the vertex is on a boundary. (If the vertex is on a boundary, the first & last edges are part of that boundary.)

NOTE: `OrderVert` requires a filled-in mesh with no "bad" vertices. Failing to adequately prepare your mesh may result in a crash. (See the methods `FillInMesh` and `EliminateBadVerts` for details.)

**Prototype:**

**`void OrderVerts();`**

**Remarks:**

This routine organizes the face and edge lists of each vertex such that going counterclockwise around the vertex (with the surface normal pointing up), you'll encounter `edg[0]`, `fac[0]`, `edg[1]`, `fac[1]`, etc, ending in either the last face if the vertex is not on a boundary or the last edge if the vertex is on a boundary. (If the vertex is on a boundary, the first & last edges are part of that boundary.)

`OrderVerts` requires a filled-in mesh with no bad vertices, so it will call `FillInMesh` and/or `EliminateBadVerts` as needed.

**Prototype:**

**`void Triangulate();`**

**Remarks:**

Converts a `MN_MESH_NONTRI` mesh, with polygon faces and possibly hidden vertices, into a completely triangulated mesh, wherein all faces have degree 3. This routine is called from `OutToTri` if the `MN_MESH_NONTRI` flag is set.

**Prototype:**

**`void TriangulateFace(int ff);`**

**Remarks:**

Triangulates the specified face, splitting it into as many faces as are needed ( $\text{deg}-2+\text{hdeg}*2$ ) to represent all the triangles.

**Parameters:**

**int ff**

Specifies the face to triangulate.

**Prototype:**

**void InvalidateTopoCache();**

**Remarks:**

Clears out topology-dependent cache information. Note that this method clears topology-dependent flags such as **MN\_MESH\_FILLED\_IN**, and thus invalidates the edge list. If you have taken pains to preserve the integrity of your edge list, you should set the **MN\_MESH\_FILLED\_IN** flag immediately after calling **InvalidateTopoCache()**.

**Prototype:**

**void Transform(Matrix3 &xfm);**

**Remarks:**

Transforms all vertices & hidden vertices by xfm.

**Prototype:**

**bool IsClosed();**

**Remarks:**

Figures out if this mesh is completely closed. Meshes with the **MN\_MESH\_RATSNEST** flags are automatically considered open. Otherwise, each edge is checked to see if it has a face on both sides. If so, the mesh is closed. Otherwise, it's open.

**Prototype:**

**void BBox(Box3 & bbox, bool targonly=FALSE);**

**Remarks:**

Calculates the bounding box of the vertices & hidden vertices of this mesh.

**Parameters:****Box3 & bbox**

The computed bounding box is placed here.

**bool targonly**

If this is TRUE, only targeted vertices are used to compute the bounding box. Hidden vertices, which are also normally used, will be ignored in this case, since they cannot be targeted.

**Prototype:**

```
void checkNormals (BOOL illum);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method can be used to build the normals and allocate RVert space only if necessary. This is a very inexpensive call if the normals are already calculated. When illum is FALSE, only the RVerts allocation is checked (since normals aren't needed for non-illum rendering). When illum is TRUE, normals will also be built, if they aren't already. So, to make sure normals are built, call this with illum=TRUE.

**Parameters:****BOOL illum**

If TRUE normals are built. If FALSE only the RVert array is allocated.

**Prototype:**

```
void buildNormals ();
```

**Remarks:**

This method is available in release 4.0 and later only.

This method resolves the normals on the RVertex array. If the MNMesh already has normals prescribed on each vertex, the normal is just moved to the RVertex array. See [Class RVertex](#) and [Class RNormal](#).

If you are creating or modifying a MNMesh, after you are done specifying all the vertices and faces, this method should be called. This allocates the RVertex and RNormal database for the MNMesh. This will allow you to query the MNMesh and ask about normals on the vertices.

This method also builds the face normals for the mesh if needed.

**Prototype:**

**void buildRenderNormals();**

**Remarks:**

This method is available in release 4.0 and later only.

This method is similar to **buildNormals()** above, but ignores the material index (mtlIndex). In other words, the difference between this and **buildNormals()** is that it doesn't look at the mtlIndex of the faces: normals of faces with the same smoothing group are averaged regardless.

**Prototype:**

**void UpdateBackfacing (GraphicsWindow \*gw, bool force);**

**Remarks:**

This method is available in release 4.0 and later only.

Updates the **MN\_BACKFACING** flag in all components based on the specified view.

**Parameters:**

**GraphicsWindow \*gw**

A pointer to the current graphics window.

**bool force**

If the gw points to the same GraphicsWindow that it did last time this method was called, it doesn't necessarily need to update the flags. If force is true, it will update them anyway. (Useful if you think the GraphicsWindow perspective may have changed.)

**Prototype:**

**void render(GraphicsWindow \*gw, Material \*ma, RECT \*rp, int compFlags, int numMat=1, InterfaceServer \*pi=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

Causes the MNMesh to display itself in the indicated GraphicsWindow.

**Parameters:****GraphicsWindow \*gw**

Points to the graphics window to render to.

**Material \*ma**

The list of materials to use to render the mesh. See [Class Material](#).

**RECT \*rp**

Specifies the rectangular region to render. If the mesh should be rendered to the entire viewport pass NULL.

**int compFlags**

One or more of the following flags:

**COMP\_TRANSFORM**

Forces recalculation of the model to screen transformation; otherwise attempt to use the cache.

**COMP\_IGN\_RECT**

Forces all polygons to be rendered; otherwise only those intersecting the box will be rendered.

**COMP\_LIGHTING**

Forces re-lighting of all vertices (as when a light moves); otherwise only re-light moved vertices

**COMP\_ALL**

All of the above flags.

**COMP\_OBJSELECTED**

If this bit is set then the node being displayed by this mesh is selected. Certain display flags only activate when this bit is set.

**int numMat=1**

The number of materials for the MNMesh.

**Prototype:**

```
void renderFace (GraphicsWindow *gw, int ff);
```

**Remarks:**

This method is available in release 4.0 and later only.

Displays the indicated face in the GraphicsWindow. This method is usually called only by **MNMesh::render()**.

**Parameters:**

**GraphicsWindow \*gw**

The GraphicsWindow in which to display this face.

**int ff**

The face to display.

**Prototype:**

**void render3DFace (GraphicsWindow \*gw, int ff);**

**Remarks:**

This method is available in release 4.0 and later only.

Displays the indicated face in the GraphicsWindow using hardware acceleration for texture and lighting if available. This method is usually called only by **MNMesh::render()**.

**Parameters:**

**GraphicsWindow \*gw**

The GraphicsWindow in which to display this face.

**int ff**

The face to display.

**Prototype:**

**void render3DDiagonals (GraphicsWindow \*gw, DWORD compFlags);**

**Remarks:**

This method is available in release 4.0 and later only.

Displays all face diagonals in the GraphicsWindow using hardware acceleration for texture and lighting if available. See the render method for a description of the parameters. This method is usually called only by **MNMesh::render()**.

**Prototype:**

**void renderDiagonals (GraphicsWindow \*gw, DWORD compFlags);**

**Remarks:**

This method is available in release 4.0 and later only.

Displays all face diagonals in the GraphicsWindow. See the render method for a description of the parameters. This method is usually called only by **MNMesh::render()**.

**Prototype:**

```
void renderDiagonal (GraphicsWindow *gw, int ff, bool  
useSegments=false, bool *lastColorSubSel=NULL);
```

**Remarks:**

This method is available in release 4.0 and later only.

Displays diagonals for the specified face in the GraphicsWindow. This method is usually called only by **MNMesh::render()**.

**Parameters:**

**GraphicsWindow \*gw**

The GraphicsWindow in which to display these diagonals.

**int ff**

The face to display the diagonals of.

**bool useSegments = false**

Indicates if we are in segment-drawing mode. See [class GraphicsWindow](#), method **segment()** for details.

**bool \*lastColorSubSel=NULL**

If non-NULL, it points to a bool variable which should be true if the last color set was the subobject selection color (**GetSubSelColor()**), and false if the color is set to the selected object color (**GetSelColor()**). This saves processing time that would be needed to switch between the two colors.

**Prototype:**

```
void render3DEdges (GraphicsWindow *gw, DWORD  
compFlags);
```

**Remarks:**

This method is available in release 4.0 and later only.

Displays all edges in the GraphicsWindow using hardware acceleration for

texture and lighting if available. See the render method for a description of the parameters. This method is usually called only by **MNMesh::render()**.

**Prototype:**

**void renderEdges (GraphicsWindow \*gw, DWORD compFlags);**

**Remarks:**

This method is available in release 4.0 and later only.

Displays all edges in the GraphicsWindow. See the render method for a description of the parameters. This method is usually called only by **MNMesh::render()**.

**Prototype:**

**void renderEdge (GraphicsWindow \*gw, int ee, bool useSegments=false, bool \*lastColorSubSel=NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

Displays an edge in a GraphicsWindow. This method is usually called only by **MNMesh::render()**.

**Parameters:**

**GraphicsWindow \*gw**

The GraphicsWindow in which to display this edge.

**int ee**

The edge to display.

**bool useSegments = false**

Indicates if we are in segment-drawing mode. See [class GraphicsWindow](#), method **segment()** for details.

**bool \*lastColorSubSel=NULL**

If non-NULL, it points to a bool variable which should be true if the last color set was the subobject selection color (**GetSubSelColor()**), and false if the color is set to the selected object color (**GetSelColor()**). This saves processing time that would be needed to switch between the two colors.

**Prototype:**

**BOOL select (GraphicsWindow \*gw, Material \*ma, HitRegion \*hr, int abortOnHit=FALSE, int numMat=1);**

**Remarks:**

This method is available in release 4.0 and later only.

Checks the given HitRegion to see if it intersects this Mesh object.

**Parameters:**

**GraphicsWindow \*gw**

Points to the graphics window to check.

**Material \*ma**

The list of materials for the mesh.

**HitRegion \*hr**

This describes the properties of a region used for the hit testing. See [Class HitRegion](#).

**int abortOnHit = FALSE**

If nonzero, the hit testing is complete after any hit. Note that although there is only one object to hit, setting this to TRUE prevents the algorithm from finding the closest hit on the MNMesh.

**int numMat=1**

The number of materials for the mesh.

**Return Value:**

TRUE if the item was hit; otherwise FALSE.

## **Component Targeting & Flag methods**

**Prototype:**

**void ClearVFlags(DWORD fl);**

**Remarks:**

Clears all specified flag bits in all MNVerts.

**Prototype:**

**void ClearEFlags(DWORD fl);**

**Remarks:**

Clears all specified flag bits in all MNEdges.

**Prototype:**

**void ClearFFlags(DWORD fl);**

**Remarks:**

Clears all specified flag bits in all MNFaces.

**Prototype:**

**void PaintFaceFlag(int ff, DWORD fl, DWORD fenceflags=0x0);**

**Remarks:**

This method is available in release 2.5 and later only.

Recursively sets flag on given face and all faces connected to it.

**Parameters:**

**int ff**

The face to begin painting the face flag on.

**DWORD fl**

The flag to set on these faces.

**DWORD fenceflags=0x0**

If nonzero, this represents flags of edges that should not be crossed. In this way you can set up a "fence" of edges and set a particular face flag on all the faces within that fence.

**Prototype:**

**void VertexSelect(const BitArray & vsel);**

**Remarks:**

This method is available in release 3.0 and later only.

Selects or deselects the verticies as specified by on bits in the given bit array.

If the bit array size is smaller than the number of verticies then only those verticies in the bit array are modified.

**Parameters:**

**const BitArray & vsel**

Specifies which verticies to select or deselect.

**Prototype:**

**void EdgeSelect(const BitArray & esel);**

**Remarks:**

This method is available in release 3.0 and later only.

Selects or deselects the edges as specified by on bits in the given bit array. If the bit array size is smaller than the number of edges then only those edges in the bit array are modified.

**Parameters:**

**const BitArray & esel**

Specifies which edges to select or deselect.

**Prototype:**

**void FaceSelect(const BitArray & fsel);**

**Remarks:**

This method is available in release 3.0 and later only.

Selects or deselects the faces as specified by on bits in the given bit array. If the bit array size is smaller than the number of faces then only those faces in the bit array are modified.

**Parameters:**

**const BitArray & fsel**

Specifies which faces to select or deselect.

**Prototype:**

**bool getVertexSel(BitArray & vsel);**

**Remarks:**

This method is available in release 3.0 and later only.

Fills the given bit array with the current vertex selection state.

**Parameters:**

**BitArray & vsel**

The results are stored here. Bits which are on indicate selected vertices.

**Prototype:**

**bool getEdgeSel(BitArray & esel);**

**Remarks:**

This method is available in release 3.0 and later only.

Fills the given bit array with the current edge selection state.

**Parameters:**

**BitArray & esel**

The results are stored here. Bit which are on indicate selected edges.

**Prototype:**

**bool getFaceSel(BitArray & fsel);**

**Remarks:**

This method is available in release 3.0 and later only.

Fills the given bit array with the current face selection state.

**Parameters:**

**BitArray & fsel**

The results are stored here. Bit which are on indicate selected faces.

**Prototype:**

**bool getVerticesByFlag(BitArray & vset, DWORD flags, DWORD fmask=0x0);**

**Remarks:**

This method is available in release 3.0 and later only.

Creates a BitArray of all vertices using the specified flags.

**Parameters:**

**BitArray & vset**

The BitArray which is filled in. vset is set to size numv.

**DWORD flags**

The flags to search for.

**DWORD fmask=0x0**

This optional parameter allows the user to look for particular combinations of on and off flags. For instance, if flags is MN\_DEAD and fmask is 0, the method finds vertices with the MN\_DEAD flag set. But if flags is MN\_SEL and fmask is MN\_SEL|MN\_DEAD, it would find vertices that have MN\_SEL set, but don't have MN\_DEAD set.

**Prototype:**

```
bool getEdgesByFlag(BitArray & eset, DWORD flags, DWORD  
fmask=0x0);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Creates a BitArray of all edges using the specified flags.

**Parameters:****BitArray & eset**

The BitArray which is filled in. eset is set to size nume.

**DWORD flags**

The flags to search for.

**DWORD fmask=0x0**

This optional parameter allows the user to look for particular combinations of on and off flags. For instance, if flags is MN\_DEAD and fmask is 0, the method finds edges with the MN\_DEAD flag set. But if flags is MN\_SEL and fmask is MN\_SEL|MN\_DEAD, it would find edges that have MN\_SEL set, but don't have MN\_DEAD set.

**Prototype:**

```
bool getFacesByFlag(BitArray & fset, DWORD flags, DWORD  
fmask=0x0);
```

**Remarks:**

This method is available in release 3.0 and later only.  
Creates a BitArray of all faces using the specified flags.

**Parameters:****BitArray & fset**

The BitArray which is filled in. fset is set to size numf.

**DWORD flags**

The flags to search for.

**DWORD fmask=0x0**

This optional parameter allows the user to look for particular combinations of on and off flags. For instance, if flags is MN\_DEAD and fmask is 0, the method finds faces with the MN\_DEAD flag set. But if flags is MN\_SEL and

fmask is MN\_SEL|MN\_DEAD, it would find faces that have MN\_SEL set, but don't have MN\_DEAD set.

**Prototype:**

**void ElementFromFace(int ff, BitArray & fset);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets bits for all faces in the same "element", or connected component, with face ff. Faces already selected in fset will be considered "walls" for this processing and will not be evaluated. That is, if ff is not selected, but there's a ring of faces around it that is, the algorithm will stop at that ring.

**Parameters:**

**int ff**

The zero based index of the face.

**BitArray & fset**

The bits of the faces in the element are selected in this array.

**Prototype:**

**void BorderFromEdge(int ee, BitArray & eset);**

**Remarks:**

This method is available in release 3.0 and later only.

Takes a one-sided edge and sets the bits representing this edge's "border". (All one-sided edges in MNMeshes can be grouped into chains, end to end, that represent boundaries of the mesh. For instance, in a box with one side deleted, all the one-sided edges are part of the chain that goes around the hole.

**Parameters:**

**int ee**

The edge to start looking from. (Note that edge ee should be one-sided – e[ee].f2 should be -1.

**BitArray & eset**

The bitarray to return the border's edge set in.

**Prototype:**

**void SetEdgeVis(int ee, BOOL vis=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

This not only sets the edge's MN\_EDGE\_INVIS flag, it also sets the visedg state of the faces on either side. This is the proper way to set edge visibility permanently, as the MNEdge flag will be lost if the edge list needs to be reconstructed.

**Parameters:**

**int ee**

The edge whose visibility should be set.

**BOOL vis=TRUE**

The desired visibility – TRUE for visible, FALSE for invisible.

**Prototype:**

**void SetEdgeSel(int ee, BOOL sel=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

This not only sets the edge's MN\_SEL flag, it also sets the edgsel state of the faces on either side. This is the proper way to set edge selection permanently, as the MNEdge flag will be lost if the edge list needs to be reconstructed.

**Parameters:**

**int ee**

The edge whose selection state should be set.

**BOOL sel=TRUE**

The desired selection state.

**Prototype:**

**int TargetVertsBySelection(DWORD selLevel);**

**Remarks:**

Sets vertex MN\_TARG flags based on the existing MN\_SEL flags.

**Parameters:**

**DWORD selLevel**

Specifies which components to check for MN\_SEL flags. If selLevel is MNM\_SL\_OBJECT, all vertices are targeted. If selLevel is MNM\_SL\_VERTEX, the vertices with MN\_SEL flags set also have their MN\_TARG flags set. If selLevel is MNM\_SL\_FACE or MNM\_SL\_EDGE, vertices that touch selected faces or edges, respectively, have their MN\_TARG flags set.

**Return Value:**

The (highly) approximate number of targeted vertices.

**Prototype:**

**int TargetEdgesBySelection(DWORD selLevel);**

**Remarks:**

Sets edge MN\_TARG flags based on the existing MN\_SEL flags.

**Parameters:**

**DWORD selLevel**

Specifies which components to check for MN\_SEL flags. If selLevel is MNM\_SL\_OBJECT, all edges are targeted. If selLevel is MNM\_SL\_EDGE, the edges with MN\_SEL flags set also have their MN\_TARG flags set. If selLevel is MNM\_SL\_FACE or MNM\_SL\_VERTEX, edges that touch selected faces or vertices, respectively, have their MN\_TARG flags set.

**Return Value:**

The (highly) approximate number of targeted edges.

**Prototype:**

**int TargetFacesBySelection(DWORD selLevel);**

**Remarks:**

Sets face MN\_TARG flags based on the existing MN\_SEL flags.

**Parameters:**

**DWORD selLevel**

Specifies which components to check for MN\_SEL flags. If selLevel is MNM\_SL\_OBJECT, all faces are targeted. If selLevel is MNM\_SL\_FACE, the faces with MN\_SEL flags set also have their MN\_TARG flags set. If selLevel is MNM\_SL\_EDGE or MNM\_SL\_VERTEX, faces that touch

selected edges or vertices, respectively, have their MN\_TARG flags set.

**Return Value:**

The (highly) approximate number of targeted faces.

**Prototype:**

```
int PropegateComponentFlags(DWORD slTo, DWORD flTo,  
DWORD slFrom, DWORD flFrom, bool ampersand=FALSE, bool  
set=TRUE);
```

**Remarks:**

This method is available in release 3.0 and later only.

This allows the developer to set flags in one type of component based on what the nearby flags of another type of component are. For instance, you might want to set MN\_TARG on all vertices that are used by faces with the MN\_SEL flag set; that would be PropegateComponentFlags (MESH\_VERTEX, MN\_TARG, MESH\_FACE, MN\_SEL);

Another example:

```
PropegateComponentFlags (MNM_SL_OBJECT, MN_MESH_FILLED_IN,  
MNM_SL_EDGE, MN_DEAD, FALSE, FALSE);
```

This would clear the MN\_MESH\_FILLED\_IN flag from the MNMesh if any of its MNEdges were dead.

**Parameters:**

**DWORD slTo**

The selection level of the components you wish to set. This would be one of MNM\_SL\_OBJECT, MNM\_SL\_VERTEX, MNM\_SL\_EDGE, or MNM\_SL\_FACE.

**DWORD flTo**

The flag to set.

**DWORD slFrom**

The selection level of the components you wish to base the selection upon. This would be one of MNM\_SL\_OBJECT, MNM\_SL\_VERTEX, MNM\_SL\_EDGE, or MNM\_SL\_FACE.

**DWORD flFrom**

The flag to test.

**bool ampersand=FALSE**

When **slFrom** and **slTo** are different, this indicates whether the flags of the nearby components should be "or'd" or "and'd". If it's false, then any flagged components in the "from" level will cause the component in the "to" level to be affected. If true, then all the components in the "from" level that touch a component in the "to" level must be flagged in order for the "to" level component to be affected. (i.e., if from is faces and to is vertices, a vertex would only be modified if all faces that use it have the **flFrom** flag set.)

**bool set=TRUE**

If TRUE (as would be usual) the slTo components have flags flTo set. If FALSE, these flags would be cleared instead.

**Return Value:**

Returns the number of slFrom components that tested positive for the flFrom flags. (If 0, nothing happened.)

**Prototype:****void DetargetVertsBySharpness(float sharpval);****Remarks:**

Clears the MN\_TARG flag on vertices representing relatively flat areas, keeping the flag on vertices at sharper corners. The purpose of this method is to avoid working on smooth areas of the mesh in, for instance, tessellation algorithms which are designed to smooth an area. An example of this is the sharpness option in 3ds max's MeshSmooth modifier, although the parameter there is 1.0 - this value.

Note that this method requires filled in geometry and ordered vertices, and will call FillInMesh and OrderVerts as needed.

**Parameters:****float sharpval**

The threshold for determining whether a vertex is sharp enough to continue being targeted. A value of 0 indicates that all vertices are sharp enough; a value of 1 would de-target all vertices. The actual determination is made by comparing the dot product of any two consecutive face normals is less than  $1 - 2 * \text{sharpval}$ .

The following "fence" methods are useful for algorithms such as SabinDoo wherein you don't want to mix faces with different characteristics.

**Prototype:**

**void FenceMaterials();**

**Remarks:**

Sets the MN\_EDGE\_NOCROSS flag on all edges that lie between faces with different material IDs.

**Prototype:**

**void FenceSmGroups();**

**Remarks:**

Sets the MN\_EDGE\_NOCROSS flag on all edges that lie between faces with exclusive smoothing groups.

**Prototype:**

**void FenceFaceSel();**

**Remarks:**

Sets the MN\_EDGE\_NOCROSS flag on all edges that lie between selected & non-selected faces. (This checks the MN\_SEL, not the MN\_TARG, flags on the faces.)

**Prototype:**

**void FenceOneSidedEdges();**

**Remarks:**

Sets the MN\_EDGE\_NOCROSS flag on all edges that are on the boundary.

**Prototype:**

**void FenceNonPlanarEdges(float thresh=.9999f, bool makevis=FALSE);**

**Remarks:**

This method is available in release 2.5 and later only.

Sets **MN\_EDGE\_NOCROSS** flags on edges between faces that aren't in

the same plane.

**Parameters:**

**float thresh=.9999f**

This is the threshold used to determine if two adjacent faces have the same normals, i.e. lie in the same plane. If the dot product between the normals is less than thresh, they are considered different, otherwise they're considered the same. The threshold angle between faces is the arc cosine of this amount, so for instance to set a threshold angle of .5 degrees, you would call FenceNonPlanarEdges with a thresh of  $\cos(.5 * \text{PI} / 180.)$ . The default value is equivalent to about .81 degrees.

**bool makevis=FALSE**

Indicates whether nonplanar edges should be made visible, i.e. have their MN\_EDGE\_INVIS flag cleared. (This is sometimes done before MakePolyMesh, so it can be used to influence whether nonplanar faces are joined together.)

**Prototype:**

**void SetMapSeamFlags ();**

**Remarks:**

This method is available in release 4.0 and later only.

Sets the MN\_EDGE\_MAP\_SEAM on all edges that are "seams" for any active map(s). A map seam is an edge where the faces on either side use different mapping vertices for at least one end.

**Prototype:**

**void SetMapSeamFlags (int mp);**

**Remarks:**

This method is available in release 4.0 and later only.

Sets the MN\_EDGE\_MAP\_SEAM on all edges that are "seams" for the specified map(s). A map seam is an edge where the faces on either side use different mapping vertices for at least one end.

**Parameters:**

**int mp**

The map to use to set the seams. If left at the default -1, it'll check all active maps.

**Prototype:**

**void SetTVSeamFlags();**

**Remarks:**

Sets the **MN\_EDGE\_TV\_SEAM** flag on all edges whose endpoints on one face have mapping coordinates that are offset from the mapping coordinates of the same endpoints on the other face by the same amount. Operating on a standard cylinder with mapping coordinates assigned, for instance, this will set the **MN\_EDGE\_TV\_SEAM** flag on the column of edges that forms the "wrap" boundary for the mapping coordinates.

**Prototype:**

**void PrepForPipeline ();**

**Remarks:**

This method is available in release 4.0 and later only.

Prepares MNMesh for pipeline. This just does a few basic checks and modifies data caches to be consistent. In particular it throws away the edge list if the **MN\_MESH\_FILLED\_IN** flag is not set, and it frees any data in **MN\_DEAD** map channels. This is a good method to call at the end of any operation on an MNMesh.

## **Component information methods**

In the following face center methods, hidden vertices have no effect.

**Prototype:**

**void ComputeCenters(Point3 \*ctr, bool targonly=FALSE);**

**Remarks:**

Finds the centers of all the faces, using repeated calls to ComputeCenter.

**Parameters:**

**Point3 \*ctr**

An array of at least FNum() points for storing the centers.

**bool targonly**

If this is TRUE, centers will only be computed for targeted faces. (The rest of the ctr array will remain unmodified.)

**Prototype:**

```
void ComputeCenter(int ff, Point3 & ctr);
```

**Remarks:**

Finds the center of face ff by finding the average of all its vertices.

**Prototype:**

```
void ComputeSafeCenters(Point3 *ctr, bool targonly=FALSE, bool  
detarg=FALSE);
```

**Remarks:**

Finds the "safe" centers of all the faces, using repeated calls to ComputeSafeCenter.

**Parameters:****Point3 \*ctr**

An array of at least FNum() points for storing the centers.

**bool targonly**

If this is TRUE, centers will only be computed for targeted faces. (The rest of the ctr array will remain unmodified.)

**bool detarg**

If TRUE, this will remove the MN\_TARG flag from faces where safe centers could not be found.

**Prototype:**

```
bool ComputeSafeCenter(int ff, Point3 & ctr);
```

**Remarks:**

Finds the "safe" center of face ff, if possible. For non-convex faces, the average point found in ComputeCenter is unsuitable for some applications because it can lie outside the face completely, or in a region where it cannot "see" all the faces' vertices. (I.e., line segments from the center to the corner pass outside of the face.)

This routine provides a better solution in some cases by finding the center of the convex hull of the face. The convex hull is defined as the region in a face with a clear line-of-sight to all the corners. Some faces, such as the top face in an extruded letter M, have an empty convex hull, in which case this routine fails and merely provides the regular center given by ComputeCenter.

**Return Value:**

Returns TRUE if a safe center was found, FALSE if no such center could be found.

**Prototype:**

**void RetriangulateFace(int ff);**

**Remarks:**

This method is available in release 2.5 and later only.

Throws out the current triangulation for face ff and computes a new one. Note that hidden vertices that actually fall outside of the region of the face will be thrown out during this routine, since they cannot be incorporated into any triangulation and don't make sense anyway.

**Prototype:**

**void BestConvexDiagonals (int ff, int \*diag=NULL);**

**Remarks:**

This method is available in release 2.5 and later only.

If the given face is convex, this will often produce a better diagonals than RetriangulateFace would. The diagonals are less likely to overuse any single vertex. The face is not checked for convexity, but if it is not convex the diagonals produced will probably be self-overlapping.

**Parameters:**

**int ff**

The face to find a diagonal for.

**int \*diag=NULL**

If NULL, the new diagonals are put into the face's tri array. If this tri is non-NULL, the diagonals are put here instead. Be sure that tri is allocated with space for at least  $(deg-2+hdeg)*3$  elements.

**Prototype:**

```
void BestConvexDiagonals (int deg, int *vv, int *diag);
```

**Remarks:**

This method is available in release 4.0 and later only.

Uses a triangulation scheme optimized for convex polygons to find a set of diagonals for this sequence of vertices, creating a triangulation for the polygon they form.

**Parameters:**

**int deg**

The number of vertices in the sequence.

**int \*vv**

The array of vertices.

**int \*diag**

A pointer to an array of size  $(deg-3)*2$  where the diagonals can be put.

**Prototype:**

```
void FindDiagonals (int ff, int *diag);
```

**Remarks:**

Finds a diagonal of face ff that does not include any hidden vertices. This can be used with the method FindFacePointTri to get a sub-triangle and barycentric coordinates for a hidden vertex, so that vertex can be kept in the face when the non-hidden vertices are moved in some way.

**Parameters:**

**int ff**

The face to get an external diagonal of.

**int \*diag**

An array of at least  $(F(ff)->deg-2)*3$  elements to store the diagonals in.

**Prototype:**

```
void FindDiagonals (int deg, int *vv, int *diag);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method finds diagonals for this sequence of vertices, creating a triangulation for the polygon they form.

**Parameters:**

**int deg**

The number of vertices in the sequence.

**int \*vv**

The array of vertices.

**int \*diag**

A pointer to an array of size  $(deg-3)*2$  where the diagonals can be put.

**Prototype:**

**void FaceBBox(int ff, Box3 & bbox);**

**Remarks:**

Finds the bounding box of all vertices and hidden vertices used by face ff.

**Prototype:**

**int FindEdgeFromVertToVert(int vrt1, int vrt2);**

**Remarks:**

Finds an edge connecting these two vertices, if one exists. This algorithm is relatively fast, since it just checks the edges in  $V(vrt1)$ 's edge list. However, it requires that the MNMesh be filled in, and it WILL call the much slower FillInMesh routine if the MN\_MESH\_FILLED\_IN flag is not set.

**Return Value:**

The index of the desired edge, or -1 if no such edge could be found.

**Prototype:**

**void GetVertexSpace (int vrt, Matrix3 & tm);**

**Remarks:**

This method is available in release 4.0 and later only.

Chooses a suitable "local space", and sets tm's rotation to match that space.

Does not set tm's translation component. This is easily done by setting `tm.SetRow (3, v[vrt].p)`.

**Parameters:****int vrt**

The vertex index.

**Matrix3 & tm**

The transformation matrix.

**Prototype:****Point3 GetVertexNormal(int vrt);****Remarks:**

Returns the surface normal at the vertex **vrt**. It computes the normal by taking the average of the face normals from faces that **V(vrt)** is on, weighted by the angles of the **vrt** corners of each of those faces. That is, if **vrt** is at a very acute angle on one face, but a very obtuse angle on the next, the face with the obtuse **vrt** corner will count much more heavily. The return value has a length of 1.

This normal is not related to the normals used in rendering.

**Prototype:****Point3 GetEdgeNormal(int ed);****Remarks:**

Returns the surface normal at the edge **ed**. This is just the average of the face normals on either side, or, if this is an edge with only one face, it's just that face's normal. The return value has a length of 1.

This normal is not related to the normals used in rendering.

**Prototype:****Point3 GetFaceNormal(int fc, bool nrmlz=FALSE);****Remarks:**

Returns the surface normal of face **fc**. If this face has degree 3, this is the same as the regular 3ds max normal for this face. However, if the face is more complex, the normal may not be the same as the ones for the component triangles.

**Parameters:****int fc**

The index of the face to find the normal for.

**bool nrmlz**

Whether or not to scale the result to length 1. If this is FALSE, the length of the return value corresponds (in planar faces) to the area of the face, times 2.

**Prototype:**

**void FlipNormal(int faceIndex);**

**Remarks:**

This method is available in release 4.0 and later only.

This method flips the normal of the indicated face. This is done by reordering the vertices. The faces for any assigned texture map is handled as well.

**Parameters:**

**int faceIndex**

The index of the face for which to flip the normal.

**Prototype:**

**Point3 GetEdgeNormal(int vrt1, int vrt2);**

**Remarks:**

This merely combines GetEdgeNormal (int ed) with FindEdge (int vrt1, int vrt2). It returns the normal of the edge connecting vrt1 to vrt2, if such an edge can be found. (If there is no such edge, it returns Point3(0.0f, 0.0f, 0.0f).)

**Prototype:**

**int FindFacePointTri(int ff, Point3 & pt, double \*bary, int \*tri);**

**Remarks:**

Finds the sub-triangle and (optionally) barycentric coordinates within that triangle of a point that lies on this face. For faces that are not themselves triangles, this helps in particular to create mapping coordinates or vertex colors for new points on this face. To use this method, you must now call **MNFace::GetTriangles** to get a triangle table, then pass the contents of that table to this method. Note that the optional size of the triangulation array was removed; this is assumed to be (f[ff].deg-2)\*3.

**Parameters:**

**int ff**

The face.

**Point3 & pt**

The point.

**double \*bary**

An array of 3 double-precision values to store the barycentric coordinates in. If this is NULL, barycentric coordinates are not computed.

**int \*tri**

An optional alternative triangulation, such as that produced by FindExternalTriangulation. (If NULL, the face's regular triangulation is used.)

**Return Value:**

3 times the index of the triangle the point is found in, or -1 if this point doesn't seem to lie on this face. That is, if this point is found in the triangle represented by tri[6], tri[7], and tri[8], this routine will return 6.

**Prototype:**

**UVVert FindFacePointMapValue(int ff, Point3 & pt, int mp);**

**Remarks:**

This method is available in release 3.0 and later only. It's the generalized version of the old FindFacePointCV and FindFacePointTV.

Uses FindFacePointTri and the mapping coordinates of the endpoints of the relevant triangle to compute a map vertex corresponding to the point pt. If the point is not on face ff, UVVert (0,0,0) is returned.

This method is useful e.g. for getting map coordinates to match a new vertex when dividing up a face.

**Parameters:**

**int ff**

The face to find map coordinates on.

**Point3 & pt**

The point (lying on the face) to find map coordinates for.

**int mp**

The map channel to get map coordinates in.

**Prototype:****VertColor FindFacePointCV(int ff, Point3 & pt);****Remarks:**

Uses FindFacePointTri and the vertex colors of the endpoints of the relevant triangle to compute a vertex color corresponding to the point pt. If the point is not on face ff, VertColor (0.0f,0.0f,0.0f) (black) is returned.

**Prototype:****UVVert FindFacePointTV(int ff, Point3 & pt);****Remarks:**

Uses FindFacePointTri and the mapping coordinates of the endpoints of the relevant triangle to compute mapping coordinates corresponding to the point pt. If the point is not on face ff, UVVert (0.0f,0.0f,0.0f) is returned.

**Prototype:****int IntersectRay (Ray& ray, float& at, Point3& norm);****Remarks:**

This method is available in release 4.0 and later only.

Provides the intersection point and normal for the ray with this mesh.

**Parameters:****Ray & ray**

The ray we want an intersection point for.

**float & at**

This is filled in with a value giving the intersection point along the ray. (The actual point is computed by  $\text{ray.p} + \text{ray.dir} * \text{at}$ .)

**Point3 & norm**

Filled in with the surface normal at the intersection point.

**Return Value:**

Returns TRUE if an intersection point was found, or FALSE if the ray doesn't intersect this MNMesh.

**Prototype:**

**int IntersectRay (Ray& ray, float& at, Point3& norm, int &fi, Tab<float> & bary);**

**Remarks:**

This method is available in release 4.0 and later only.

Provides the intersection point and normal for the ray with this mesh.

**Parameters:**

**Ray & ray**

The ray we want an intersection point for.

**float & at**

This is filled in with a value giving the intersection point along the ray. (The actual point is computed by  $\text{ray.p} + \text{ray.dir} * \text{at}$ .)

**Point3 & norm**

Filled in with the surface normal at the intersection point.

**int & fi**

Filled in with the face index for the face that was hit by the ray.

**Tab<float> & bary**

Filled in with the "generalized barycentric coordinates" of the intersection point on the face. This is a table of floats of size  $f[\text{fi}].\text{deg}$ , where each float represents the contribution of the corresponding face vertex, and where the floats all sum to 1.

**Return Value:**

Returns TRUE if an intersection point was found, or FALSE if the ray doesn't intersect this MNMesh.

**Prototype:**

**BitArray VertexTempSel (DWORD fmask=MN\_DEAD|MN\_SEL, DWORD fset=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

Gets the current vertex selection, based on the current selection level. That is, if the current selection level is **MNM\_SL\_VERTEX** and the parameters are at their defaults, it'll return the current vertex selection, but if the selection level is **MNM\_SL\_FACE**, for instance, it'll return the vertices used by

currently selected faces.

This method is used, e.g. in `PolyObject::Deform` to determine which vertices to affect by pipeline modifiers.

**Parameters:**

**DWORD fmask=MN\_DEAD|MN\_SEL**

Indicates the flags we're trying to match in the components at the current selection level.

**DWORD fset=MN\_SEL**

Indicates which flags (from fmask) we want to see set. The default values mean "find components with **MN\_SEL** set and **MN\_DEAD** cleared".

**Prototype:**

**void ShallowCopy(MNMesh \*amesh, ULONG\_PTR channels);**

**Remarks:**

This method is available in release 4.0 and later only.

Provides the guts of pipeline shallow copying. Used by

**PolyObject::ShallowCopy()**.

**Prototype:**

**void NewAndCopyChannels(ULONG\_PTR channels);**

**Remarks:**

This method is available in release 4.0 and later only.

Provides the guts of pipeline new & copying. Used by

**PolyObject::NewAndCopyChannels()**.

**Prototype:**

**void FreeChannels (ULONG\_PTR channels, BOOL zeroOthers=1);**

**Remarks:**

This method is available in release 4.0 and later only.

Provides the guts of pipeline channel freeing. Used by

**PolyObject::FreeChannels()**.

## Smoothing-group & Material methods

### Prototype:

```
void Resmooth(bool smooth=TRUE, bool targonly=FALSE,  
DWORD targmask=~0x0);
```

### Remarks:

Applies new smoothing (or removes smoothing groups) from selected areas of the MNMesh. With the default parameters, it smoothes all faces with the same group.

### Parameters:

#### **bool smooth**

If TRUE, Resmooth will generate a new smoothing group (using GetNewSmGroup) to apply to the relevant faces. All smoothing groups previously assigned to the faces will be removed. If FALSE, Resmooth will strip all smoothing groups, leaving the faces faceted.

#### **bool targonly**

If TRUE, Resmooth will not affect faces that do not have the MN\_TARG flag set.

#### **DWORD targmask**

Resmooth will only affect faces whose smoothing groups are included in targmask. Some examples: with the default value of ~0x0, this is no restriction at all. With a value of 0, Resmooth will only affect faces that previously had no smoothing group assigned. With a value of 0x02, Resmooth will only affect faces that either had smoothing group 2 (and nothing else) or no smoothing group.

Note: if **targonly** is TRUE and targmask is not at the default, a face must both be targeted and have its smoothing groups fall into targmask in order to be affected.

### Prototype:

```
DWORD CommonSmoothing(bool targonly=FALSE);
```

### Remarks:

Finds what smoothing groups, if any, are common to all faces in this MNMesh.

**Parameters:**

**bool targonly**

If this is TRUE, this routine will find smoothing groups that are common to all faces with MN\_TARG set, ignoring the rest.

**Prototype:**

**DWORD GetAllSmGroups(bool targonly=FALSE);**

**Remarks:**

This method is available in release 2.5 and later only.

Find all smoothing groups used in this mesh.

**Parameters:**

**bool targonly=FALSE**

If TRUE, this method will return only smoothing groups set in targeted faces, i.e. those with the MN\_TARG flag set.

**Return Value:**

The return value is a DWORD with every used smoothing bit set.

**Prototype:**

**DWORD GetNewSmGroup(bool targonly=FALSE);**

**Remarks:**

Produces a single smoothing group that is not currently used in the MNMesh. If this is impossible because all smoothing groups are used (a rare condition), it produces the least commonly used group.

**Parameters:**

**bool targonly**

If TRUE, this routine will find a smoothing group not used by any of the faces with MN\_TARG set. If this is impossible because all smoothing groups are used in targeted faces, it produces the least commonly used group.

**Prototype:**

**DWORD FindReplacementSmGroup(int ff, DWORD os);**

**Remarks:**

This method is available in release 2.5 and later only.

Finds available smoothing groups you can use to replace the given smoothing

group without messing up the mesh's shading. This method recursively looks at the entire region of faces sharing this smoother with this face, and it finds all smoothing groups used by neighboring faces. Then it returns the bits which are not used by any of them.

This is useful if, for instance, you want to attach two separate mesh components, but you don't want smooth shading across the join. It's used internally by `SeparateSmGroups`.

**Parameters:**

**int ff**

The face to start the examination on.

**DWORD os**

The old smoothing group you wish to replace.

**Return Value:**

All bits that are available to replace the old smoothing group with.

**Prototype:**

**void PaintNewSmGroup(int ff, DWORD os, DWORD ns);**

**Remarks:**

This method is available in release 2.5 and later only.

Recursively replaces the old smoothing group with the new smoothing group. The recursion traverses all faces with the old smoother that share an edge or a vertex.

**Parameters:**

**int ff**

The face to begin the replacement on

**DWORD os**

The old smoothing group

**DWORD ns**

The new smoothing group

**Prototype:**

**bool SeparateSmGroups(int v1, int v2);**

**Remarks:**

This method is available in release 2.5 and later only.

Changes the smoothing groups on faces using v2 so that they are distinct from any smoothing groups on faces using v1. This is used, for instance, in joining the seam between the operands MakeBoolean. Before welding each pair of vertices, this method is called to prevent smoothing across the boolean seam.

**Return Value:**

If, due to overuse of the 32 smoothing groups, the algorithm can't find enough new ones to replace the overlapping smoothing groups around v2, it will do the best it can and return FALSE. If it succeeds, it returns TRUE.

**Prototype:**

**MtlID GetNewMtlID(bool targonly = FALSE);**

**Remarks:**

Produces the lowest material ID not used in any faces in the MNMesh.

**Parameters:**

**bool targonly**

If TRUE, this routine will instead find the lowest material ID not used in the targeted faces of this MNMesh.

**Prototype:**

**DWORD GetOldSmGroup(bool targonly=FALSE);**

**Remarks:**

Returns a smoothing group that is currently used somewhere in the mesh, or returns zero if all faces have no smoothing.

**Parameters:**

**bool targonly**

If TRUE, this routine will find a smoothing group used in one of the targeted faces, or return zero if all targeted faces have no smoothing.

**Prototype:**

**void AutoSmooth(float angle,BOOL useSel,BOOL preventIndirectSmoothing);**

**Remarks:**

This method is available in release 4.0 and later only.

This method performs an auto smooth on the mesh, setting the smoothing groups based on the surface normals.

**Parameters:**

**float angle**

The minimum angle between surface normals for smoothing to be applied, in radians.

**BOOL useSel**

If TRUE only the selected faces are smoothed.

**BOOL preventIndirectSmoothing=FALSE**

TRUE to turn on; FALSE to leave off. This matches the option in the Smooth Modifier UI -- use this to prevent smoothing "leaks" when using this method. If you use this method, and portions of the mesh that should not be smoothed become smoothed, then try this option to see if it will correct the problem. Note that the problem it corrects is rare, and that checking this slows the Auto Smooth process.

**Prototype:**

```
void ApplyMapper (UVWMapper & mp, BOOL channel=0,  
BOOL useSel=FALSE);
```

**Remarks:**

This method is available in release 4.0 and later only.

Applies UVW Mapper to create the desired mapping in the specified channel.

**Parameters:**

**UVWMapper & mp**

The mapping scheme. See [class UVWMapper](#) for details.

**BOOL channel=0**

The channel to apply the map to. Channel 0 is normally the vertex color channel. Channels 1-99 are the normal user-accessible mapping channels. Channels **MAP\_SHADING** and **MAP\_ALPHA** are used for vertex illumination and alpha.

**BOOL useSel=FALSE**

Indicates whether the specified mapping should be applied only to selected faces (instead of the whole object). In cases where the map channel was previously unused, a default map will be applied to nonselected faces, and the

specified map will be applied to the selected faces.

## Face-joining methods

### Prototype:

```
void MakePolyMesh();
```

### Remarks:

Turns a mesh with triangle faces into a mesh with (fewer) polygon faces by removing all hidden edges. This method can take unusually long if there are faces of ridiculously high degree, such as the top of a cylinder with 200 sides but only 1 cap segment. (It can be up to an n-squared routine where n is the number of invisible edges removed to make a given face.)

This routine is essentially a bunch of calls to **RemoveEdge()**, followed by a call to **EliminateCollinearVerts()**.

### Prototype:

```
void MakeConvexPolyMesh();
```

### Remarks:

This method is available in release 2.5 and later only.

Turns a mesh with triangle faces into a mesh with (fewer) convex polygon faces by removing all hidden edges that aren't necessary to maintain convexity. As of the 2.5 release this method has not been thoroughly tested to ensure that the result is indeed always convex and that it doesn't leave edges that should ideally be removed. Therefore a call to **MakeConvex** is recommended after use. What we can say about it is that it will not produce any hidden vertices, as **MakePolyMesh** does, and that if you're going to use **MakeConvex** anyway, **MakeConvexPolyMesh** and **MakeConvex** are more efficient than **MakePolyMesh** and **MakeConvex** are.

**MakeConvex** essentially undoes a lot of the work done by **MakePolyMesh**. **MakeConvexPolyMesh** gives it much less (if anything) to undo.

### Prototype:

```
void RemoveEdge(int edge);
```

### Remarks:

Kills the edge and joins the faces on either side. Does not work with one-sided

edges. Re-indexes triangulation in the resulting face to maintain MN\_MESH\_FILLED\_IN integrity. This routine also checks the resulting face for "dangling" edges: if two faces share a common boundary of 3 edges, and the middle edge is removed, the other two edges will actually have the same face on both sides. This is silly, so such edges are also removed & killed, reducing the degree of the face and adding a hidden vertex.

**Prototype:**

**void MakeConvex();**

**Remarks:**

Calls **MakeFaceConvex()** on all (non-dead) faces, producing a mesh with 100% convex faces.

**Prototype:**

**void MakeFaceConvex(int ff);**

**Remarks:**

Makes face ff convex, essentially by chopping off corners. The corners themselves become new faces. This routine is actually recursive: if this face is already convex, it returns without affecting it. If not, it finds a suitable line (between outside vertices) to divide the face along, and then calls **MakeFaceConvex()** again on both of the smaller faces. It is crucial that the face given to MakeFaceConvex has a valid triangulation.

**Prototype:**

**void EliminateCollinearVerts();**

**Remarks:**

This routine scans through a mesh from which invisible edges have been removed, looking for vertices that essentially seem to lie in the middle of edges. Such "collinear vertices" actually lie between two edges that are parallel, with the same faces on either side, and with no other edges incident on them. These are easily removed, and if not removed, they can cause unpleasant artifacts in some tessellation algorithms.

The removal of these vertices consists of deleting them, merging the parallel edges into one (longer) edge, and correcting the faces on either side to have

one less vertex and edge.

Note that this is essentially the opposite of the method `SplitEdge`.

**Prototype:**

**void EliminateCoincidentVerts(float thresh=MNEPS);**

**Remarks:**

This merges vertices that lie extremely close together, similar to what's done in Edit Mesh's `Weld` function, but it only affects vertices that are joined by an edge. Another way of looking at it is that it uses `weld` to remove extremely small edges.

**Parameters:**

**float thresh=MNEPS**

This is the maximum length of an edge that will be welded away. The default value of `MNEPS` is defined in `MNCommon.h` to be `.0001` -- generally we use this to delete edges of more or less zero length.

**Splitting methods**

**Prototype:**

**int SplitEdge(int ee, float prop=.5f);**

**Remarks:**

This method is available in release 2.5 and later only.

Creates a new vertex somewhere in the middle of an edge, splitting the edge into two edges. Incident faces are updated in their vertex, edge, mapping coordinate, and vertex color lists, as well as in their triangulation. This method requires that `MN_MESH_FILLED_IN` be set (otherwise it will cause an assertion failure), and maintains the combinatorics completely.

**Parameters:**

**int ee**

The edge to split.

**float prop=.5f**

The proportion along the edge where the new vertex should go. This proportion should be between 0 and 1. The new vertex location is  $P(E(ee) \rightarrow v1) * (1.0f - prop) + P(E(ee) \rightarrow v2) * prop$ .

**Return Value:**

The index of the new vertex.

**Prototype:**

```
int SplitEdge(int ff, int ed, float prop, bool right, int *nf=NULL, int *ne=NULL, bool neVis=FALSE, bool neSel=FALSE, bool allconvex=FALSE);
```

**Remarks:**

This method is available in release 2.5 and later only.

Splits an edge that is specified from the face level, and splits off triangles from the adjacent faces. This method is often preferable in convex meshes, when it's important that no nonconvex faces are introduced. The other SplitEdge methods leave faces with three vertices in a line, which is not strongly convex. This version makes a triangle out of half of the split edge, the next face edge, and a diagonal.

Note: This method absolutely requires that the faces on either side of the edge being split are Convex. Use MakeFaceConvex if needed.

**Parameters:**

**int ff**

One of the faces which uses the edge you wish to split.

**int ed**

The index of the edge in face ff.

**float prop**

The proportion along the edge where the new vertex should go. This proportion should be between 0 and 1. The new vertex location is  $P(E(ee) \rightarrow v1) * (1.0f - prop) + P(E(ee) \rightarrow v2) * prop$ .

**bool right**

If FALSE, the new triangle is formed from the lower or "left" half of the split edge. If TRUE, it's formed from the higher or "right" half. (These orientations make sense if you imagine viewing the face from above with the split edge on the bottom, as pictured.) If you want the triangle to be formed from the smaller half, use  $(prop \leq .5f)$  for this argument.

**int \*nf=NULL**

If non-NULL, this should point to an array of at least 2 elements where the new face indices should be put. Nf[0] is set to the new face created from face ff, while nf[1] is set to the new face created from the face on the other side of

the edge, if any.

**int \*ne=NULL**

If non-NULL, this should point to an array of at least 3 elements where the new edge indices should be put. Ne[0] is set to the new edge created from the second half of the edge we're splitting. Ne[1] represents the diagonal edge on face ff, while ne[2] represents the diagonal edge on the face on the other side of the split edge (if any).

**bool neVis=FALSE**

Indicates whether the new edges ne[1] and ne[2] should be visible.

**bool neSel=FALSE**

Indicates whether the new edges ne[1] and ne[2] should be selected.

**bool allconvex=FALSE**

Indicates whether the original faces on both sides of the edge are guaranteed to be convex. If so, these faces are retriangulated with BestConvexTriangulation; otherwise, RetriangulateFace is used.

**Return Value:**

The index of the new vertex.

**Prototype:**

**int SplitTriEdge(int ee, float prop=.5f, float thresh=MNEPS, bool neVis=TRUE, bool neSel=FALSE);**

**Remarks:**

Creates a new vertex in the middle of an edge, splitting the edge into two edges. Unlike SplitEdge, this routine requires a triangle face on each side. Since it has this, it also creates new edges connecting the new vertex with the point opposite the split edge on each face. This in turn splits the faces. So two triangles sharing this single edge become four triangles with four shared edges meeting at a single new vertex.

This method requires that MN\_MESH\_FILLED\_IN be set (otherwise it will cause an assertion failure), and maintains the combinatorics completely.

**Parameters:**

**int ee**

The edge to split. Note that if the faces on either side do not have deg=3 and hdeg=0, an assertion failure will result.

**float prop**

The proportion along the edge for the new vertex to be located. This ranges from 0, at v1, to 1, at v2. A value of .3, for instance, would create a new vertex at  $.7 * P(v1) + .3 * P(v2)$ .

**float thresh**

If prop is less than thresh or more than 1-thresh, the method will not split the edge, and will return v1 or v2 respectively. This is so that calling routines can safely pass values anywhere from 0 to 1, without worrying about creating tiny "shard" faces.

**bool neVis**

This tells whether the new edges that split the faces should be visible. (The new edge formed from part of the old edge takes such characteristics from the old edge.)

**bool neSel**

This tells whether the new edges that split the faces should be selected. (The new edge formed from part of the old edge takes such characteristics from the old edge.)

**Return Value:**

The index of the new vertex created, or the index of the old vertex you can use instead if you're within thresh of the endpoints.

**Prototype:**

```
int IndentFace(int ff, int ei, int nv, int *ne=NULL, bool nevis=TRUE, bool nesel=FALSE);
```

**Remarks:**

This method is available in release 2.5 and later only.

Uses a new vertex to "indent" one of the sides of the face. The indentation triangle is split off as a new face, which is returned.

**Parameters:****int ff**

The face to be "indented".

**int ei**

The index of the edge which will be replaced by the indentation.

**int nv**

The index of the new vertex. This vertex should lie within the face, in the face's plane, and should not be connected to any faces or edges.

**int \*ne=NULL**

If non-NULL, this should point to an array of at least 2 elements where the new edges should be stored.

**bool nevis=TRUE**

Indicates whether the new edges should be visible.

**bool nesel=FALSE**

Indicates whether the new edges should be selected.

**Return Value:**

The index of the new face representing the indentation triangle.

**Prototype:**

**void SeparateFace(int ff, int a, int b, int & nf, int & ne);**

**Remarks:**

This routine is used in MakeFaceConvex, but can be used outside of it too. It takes a larger face and divides it into two smaller faces, creating a new edge in the process.

It is crucial that the face given to SeparateFace has a valid triangulation.

**Parameters:**

**int ff**

The face to divide.

**int a,b**

The indices (in the face's vtx list) of the vertices used to divide the face. Note that a and b should be at least two units apart in each direction: a can't be (b+1) mod deg, and b can't be (a+1) mod deg. Also, the straight line connecting MNVerts V(vtx[a]) and V(vtx[b]) should not cross any of the edges of the face, otherwise the results will be truly screwed.

**int & nf**

A variable to hold the new face created by this division. (Half the face remains as face ff, the other half is the new face nf.)

**int & ne**

A variable to hold the new edge created by this division.

**Prototype:**

```
void SeparateFace(int ff, int a, int b, int & nf, int & ne, bool neVis,  
bool neSel);
```

**Remarks:**

This method is available in release 2.5 and later only.

This routine takes a larger face and divides it into two smaller faces, creating a new edge in the process. It is crucial that the face given to SeparateFace has a valid triangulation.

**Parameters:****int ff**

The face to divide.

**int a, b**

The indices (in the face's vtx list) of the vertices used to divide the face. Note that a and b should be at least two units apart in each direction: a can't be  $(b+1) \bmod \text{deg}$ , and b can't be  $(a+1) \bmod \text{deg}$ . Also, the straight line connecting MNVerts  $V(\text{vtx}[a])$  and  $V(\text{vtx}[b])$  should not cross any of the edges of the face, otherwise the results will not be valid.

**int & nf**

A variable to hold the new face created by this division. (Half the face remains as face ff, the other half is the new face nf.)

**int & ne**

A variable to hold the new edge created by this division.

**bool neVis**

Indicates whether the new edge should be visible. (The other version of SeparateFace always leaves the edge invisible.)

**bool neSel**

Indicates whether the new edge should be selected. (The other version of SeparateFace always leaves the edge unselected.)

**Prototype:**

```
void Slice(Point3 & N, float off, float thresh, bool split, bool  
remove, DWORD selLevel);
```

**Remarks:**

This method is available in release 2.5 and later only.

Slices the MNMesh along a specified plane, splitting edges and faces as needed to divide faces into those above and those below the plane. Equivalent to the Slice modifier.

**Parameters:**

**Point3 & N**

The normal to the slice plane. This should be a unit vector.

**float off**

The offset of the slice plane. For any point X in the plane,  $\text{DotProd}(\text{N}, \text{X}) = \text{off}$ .

**float thresh**

If a vertex lies within thresh of the splitting plane, ie if  $\text{abs}(\text{DotProd}(\text{N}, \text{X}) - \text{offset}) < \text{thresh}$  for a point X, it's considered to be on the plane. This prevents some points being created extremely close to each other. Zero is an acceptable value; the constant MNEPS is what's used in the Slice Modifier.

**bool split**

Indicates whether the points and edges along the slice should be replicated, dividing the mesh into separate "above" and "below" connected components. Equivalent to the Slice modifier's "Split Mesh".

**bool remove**

Indicates whether the portion of the mesh above the split plane should be removed. Equivalent to the Slice modifier's "Remove Top". (To "Remove Bottom", just multiply N and off by -1.)

**DWORD selLevel**

Indicates whether the Slice effect should be restricted to selected faces. There is no support currently for slicing selected edges or vertices, but if  $\text{selLevel} == \text{MNM\_SL\_FACE}$ , only selected faces will be sliced or removed.

**Prototype:**

```
int SplitTriFace(int ff, double *bary=NULL, float thresh=MNEPS, bool neVis=TRUE, bool neSel=FALSE);
```

**Remarks:**

Adds a vertex somewhere in a triangle face, and also adds edges from that vertex to each of the corners, splitting the face into 3 smaller triangle faces. The triangle face used here can have hidden vertices, as long as it is of degree

3. However, if the threshold parameter comes into play and this routine calls SplitTriEdge, both this face and the one across the relevant edge will need to have both degree 3. (Otherwise there's an assertion failure.)

This method requires that MN\_MESH\_FILLED\_IN be set (otherwise it will cause an assertion failure), and maintains the combinatorics completely.

**Parameters:**

**int ff**

The face to split.

**double \*bary**

The barycentric coordinates of the new point you wish to add. If this is NULL, the default values of (1/3, 1/3, 1/3) (the middle of the triangle) are used. These values MUST all fall between 0 and 1, and they MUST add up to 1 (give or take a floating point error) to get a sensible result.

**float thresh**

If one of the barycentric coordinates is greater than 1-thresh, that vertex dominates completely. No splitting is done, and that vertex is returned. If not, but if one of the barycentric coordinates is less than thresh, the new point must fall on the opposite edge. SplitTriEdge is therefore called on that edge, and passes along the value this returns.

**bool neVis**

Whether or not the new edges connecting the new vertex to the corners should be visible.

**bool neSel**

Whether or not the new edges connecting the new vertex to the corners should be selected.

**Return Value:**

The index of the new vertex created, or the index of the old vertex you can use instead if you're within thresh of one of the corners.

**Prototype:**

**void SplitTri6(int ff, double \*bary=NULL, int \*nv=NULL);**

**Remarks:**

This is another way to subdivide a face on a triangular mesh into sub-triangles. In this case, 4 new vertices are produced, and this face becomes 6 new faces.

**MN\_MESH\_FILLED\_IN** is required and preserved, and **MN\_MESH\_NONTRI** cannot be true when this method is called.

The subdivision technique is as follows: a new point is added at the barycentric coordinates given on the face given. New points are also added in each edge of the original face, such that a line from each of these three new edge points to the opposite original vertex passes through the new center point. These three edge points have edges drawn between them, and have edges to the new center point, dividing the face into 3 large outer triangles and 3 smaller inner triangles. Neighboring faces are split into 2 triangles, since the common edge is divided. This is a useful subdivision technique when you know you're going to want to add a lot of detail to a specific region of a face.

**Parameters:**

**int ff**

The face to split.

**double \*bary=NULL**

The barycentric coordinates for the center point. If bary is NULL, the default barycentric coordinates of (1/3, 1/3, 1/3) are used.

**int \*nv=NULL**

This is a pointer to an array of at least 4 int's in which the 4 new vertex indices should be stored. (This is mainly if the calling routine needs to know what these new vertices are.) If this is NULL, it is ignored.

## **Border methods**

**Prototype:**

```
void GetBorder(MNMeshBorder & brd, DWORD  
selLevel=MESH_OBJECT);
```

**Remarks:**

Finds border loops composed of all one-sided edges. (One-sided edges on a mesh must necessarily be organizable into closed loops along the borders of the mesh.)

**Parameters:**

**MNMeshBorder & brd**

The class in which to put the border loops. See [Class MNMeshBorder](#) for

details.

### **DWORD selLevel**

The selection level in the mesh to use to decide whether the border loops are targeted or not. For instance, with the default `MNM_SL_OBJECT`, all border loops are targeted, but with a value of `MNM_SL_VERTEX`, only those border loops containing at least one selected vertex will be targeted. Same for `MNM_SL_EDGE`. For `MNM_SL_FACE`, those loops touching at least one selected face will be targeted.

### **Prototype:**

```
void FillInBorders(MNMeshBorder *b=NULL);
```

### **Remarks:**

Fills in all the borders of a mesh. This is what the 3ds max CapHoles modifier does. It creates new faces on the other side of each of the loops.

### **Parameters:**

#### **MNMeshBorder \*b**

If this is `NULL`, `FillInBorders` finds all the borders with a call to `GetBorders`. If you have already found the borders, however, you can save time by passing a pointer to the `MNMeshBorder` class here. (This is especially convenient if you wish to modify the border targeting; only targeted borders are filled in.)

### **Prototype:**

```
void FindOpenRegions();
```

### **Remarks:**

This method is available in release 2.5 and later only.

Scans mesh to determine what parts of it are open or closed. Consider a "submesh" to be one connected components of faces and edges. Each submesh is open if it has any holes, i.e. if there's at least one edge which has only one face. (The hole is on the other side.) A submesh is closed if it doesn't have any. This method sets the `MN_FACE_OPEN_REGION` flag on all faces in open submeshes, and sets the `MN_MESH_HAS_VOLUME` flag if at least one submesh is closed.

### **Examples:**

A 3ds max box has one submesh, itself, and it is closed.

A 3ds max teapot has four distinct submeshes (handle, spout, lid, and body),

and they are all open.

**Prototype:**

```
void FindEdgeListTCVerts(const IntTab & lp, IntTab & tv, IntTab  
& cv);
```

**Remarks:**

Given a complete loop boundary of edges in the MNMesh, this method will find mapping coordinates and/or vertex colors for the vertices of the loop.

**Parameters:**

(**IntTab** means **Tab<int>**)

**const IntTab & lp**

The loop of edges.

**IntTab & tv**

The location to put the mapping coordinates' indices. Since each vertex on the loop lies can lie on two faces, one corresponding to each border edge touching the vertex, there are  $2 * lp.Count()$  of these values.  $tv[i*2]$  is the mapping vertex used for vertex  $E(lp[i]) \rightarrow v1$  in face  $E(lp[i]) \rightarrow f1$ , while  $tv[(i-1)*2+1]$  is the mapping vertex used for vertex  $E(lp[i-1]) \rightarrow v2$  (which is also  $E(lp[i]) \rightarrow v1$ ) in face  $E(lp[i-1]) \rightarrow f1$ .

**IntTab & cv**

The location to put the vertex colors' indices. Since each vertex on the loop lies can lie on two faces, one corresponding to each border edge touching the vertex, there are  $2 * lp.Count()$  of these values.  $cv[i*2]$  is the vertex color used for vertex  $E(lp[i]) \rightarrow v1$  in face  $E(lp[i]) \rightarrow f1$ , while  $cv[(i-1)*2+1]$  is the vertex color used for vertex  $E(lp[i-1]) \rightarrow v2$  (which is also  $E(lp[i]) \rightarrow v1$ ) in face  $E(lp[i-1]) \rightarrow f1$ .

## **Tessellation & related methods**

**Prototype:**

```
void Relax(float relaxval, bool targonly=TRUE);
```

**Remarks:**

Similar to the 3ds max "Relax" modifier: this modifier moves each MNVert towards the average of all vertices to which it is connected (by MNEdges).

**Parameters:****float relaxval**

The proportion to move it. If  $p$  is the original location of this vertex and  $q$  is the average point of all vertices it's connected to, this vertex is moved to  $p*(1-relaxval) + q*relaxval$ . A value of 0 generates no "relaxation", 1 is maximum relaxation, and values below 0 or above 1 generate non-relaxing results.

**bool targonly**

If this is set, only those MNVerts with the MN\_TARG flag set will be moved.

**Prototype:**

```
void FindEdgeListMapVerts(const Tab<int> & lp, Tab<int> & mv,  
int mp);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns map verts for both ends of each edge (from f1's perspective) (Very useful for creating new faces at borders.)  $mv[j*2]$  is the map vertex corresponding to edge  $j$ 's  $v1$ .

**Parameters:****const Tab<int> & lp**

A loop of border edges, such as is generated by MNMesh::FindBorders.

**Tab<int> & mv**

The table into which the map vertices should be put. The size is set to  $2*lp.Count()$ .

**int mp**

The map channel we're analyzing.

**Prototype:**

```
bool AndersonDo(float interp, DWORD selLevel,  
MeshOpProgress *mop=NULL, DWORD subdivFlags=0);
```

**Remarks:**

This is a tessellation routine. It breaks every MNFace into  $deg$  new faces with four sides. The four vertices of each of these new quads come from the center point of the original face, one corner of the original face, and the middles of

the two edges on either side of that corner. For a demonstration of the effect this algorithm has on meshes, apply 3ds max's MeshSmooth modifier with the "Quad Output" box checked. Note that this algorithm roughly quadruples the size of the MNMesh.

**Parameters:**

**float interp**

This is the proportion of relaxation applied to the original vertices to produce a smoothed result. Vertices in the result correspond to three sources: all the original vertices are still included, new vertices are produced at the center of each face, and new vertices are produced in the middle of each edge. Face-vertices are always at the exact center of the original faces, but edge-vertices and vertex-vertices are relaxed into the mesh to improve smoothness.

**DWORD selLevel**

The selection level that the Mesh we're modifying was set to. If this selection level is MNM\_SL\_FACE, we'd want to keep non-selected faces as unmodified as possible, whereas if the selection level is MNM\_SL\_OBJECT or MNM\_SL\_VERTEX, we could break off chunks of some unselected faces that touch modified vertices. Again, observe the result of MeshSmooth, with Quad Output, on various selection sets of a mesh for examples of this behavior.

**MeshOpProgress \*mop=NULL**

This optional parameter points to an instance of the virtual class MeshOpProgress, which is used to allow the user to abort out of lengthy calculations. (SabinDoo is a lengthy calculation.) See [class MeshOpProgress](#).

**DWORD subdivFlags=0**

This parameter is available in release 4.0 and later only.

There is currently only one flag for the extra argument,

**MNM\_SUBDIV\_NEWMAP**. If this flag is set, the new mapping scheme will be used.

**Prototype:**

```
void TessellateByEdges(float bulge, MeshOpProgress  
*mop=NULL);
```

**Remarks:**

This is a tessellation routine similar to AndersonDo. It produces the same topology, but bulges new vertices outward to maintain even curvature instead of bringing the original vertices inward. For a demonstration of the effect this algorithm has on meshes, apply 3ds max's Tessellate modifier with the "Edge" and "Operate On: Polygons" options selected. Note that this algorithm roughly quadruples the size of the MNMesh.

**Parameters:**

**float bulge**

The factor to "push out" new vertices by in the direction of the surface curvature. Values of about .25 are reasonable. This value is equivalent to one hundredth of the value specified as "Tension" in the Tessellate modifier.

**MeshOpProgress \*mop=NULL**

A pointer to a MeshOpProgress. See [Class MeshOpProgress](#) for details.

**Prototype:**

**void TessellateByCenters();**

**Remarks:**

This is a tessellation routine. Note that this algorithm roughly triples the size of the MNMesh. For an example of how it works, apply 3ds max's Tessellate modifier with the "Face" and "Operate On: Polygons" options selected. This algorithm splits each face into deg triangles, where each triangle contains the center of the face and one edge from the perimeter.

**Prototype:**

**void SabinDoo(float interp, DWORD selLevel, MeshOpProgress \*mop=NULL, Tab<Point3> \*offsets=NULL);**

**Remarks:**

This is a tessellation routine. Note that this algorithm roughly triples the size of the MNMesh. For an example of how it works, apply 3ds max's MeshSmooth modifier with "classic" MeshSmooth Type.

This technique for tessellation is based on a paper, "A Subdivision Algorithm for Smoothing Down Irregularly Shaped Polygons", published by D. W. H. Doo of Brunel University, Middlesex, England, in IEEE in 1978. It essentially creates a new face for every vertex, face, and edge of the original mesh. The

new vertices this technique requires are made by creating one vertex for each corner of each original face. These vertices are located on a line from the original face corner to its center. All the faces around a given (targeted) vertex will create such a point; these points form the corners of the face created from this vertex. The original faces become smaller, as they use the new points instead of their old corners. And the four new points created on the faces on either side of an edge, for the endpoints of that edge, become the four corners of the new face representing the edge. Apply a MeshSmooth with default values to a 3ds max box to see how this plays out.

### **Parameters:**

#### **float interp**

The position along the line from a vertex to the center of each face to create the new vertex at. If this value is near 0, the original faces will shrink very little, and the new faces at the vertices and edges will be very small. If this value is near 1, the original faces will shrink to almost nothing, and the new vertex faces will be dominant. Again, adjust the Strength parameter in MeshSmooth to gain an understanding of this parameter.

#### **DWORD selLevel**

This parameter is included so that special handling can be applied to selected faces, if we're at the MNM\_SL\_FACE selection level. Generally, this algorithm works on all targeted vertices, affecting all faces that contain at least one targeted vertex. However, if the vertices are targeted by whether or not they're on selected faces, we'll wind up "spilling" the algorithm over into the non-selected faces. Generally in such routines as MeshSmooth, the user would want non-selected faces not to be affected at all. If this is the case, setting this value to MNM\_SL\_FACE will prevent faces without the MN\_SEL flag from being affected.

#### **MeshOpProgress \*mop=NULL**

This optional parameter points to an instance of the virtual class MeshOpProgress, which is used to allow the user to abort out of lengthy calculations. (SabinDoo is a lengthy calculation.) See [Class MeshOpProgress](#).

#### **Tab<Point3> \*offsets=NULL**

The SabinDoo algorithm is typically used to go from one polygonal approximation of a smooth surface to another polygonal approximation of 4 times the size, but it never actually returns the vertices as they would appear on the "limit surface". If this parameter is non-NULL, it's filled in with offsets

that will, when added to the vertices in the SabinDoo result, take those vertices onto the limit surface. This makes a smoother result that matches better between different iterations.

**Prototype:**

```
void SabinDooVert(int vid, float interp, DWORD selLevel, Point3  
*ctr, MeshOpProgress *mop=NULL, Tab<Point3>  
*offsets=NULL);
```

**Remarks:**

This applies the Sabin-Doo tessellation technique described above to a single vertex. The vertex is split into as many new vertices as there are faces using this vertex. The incident faces are shrunk back to make use of these new vertices, and a new face is created from them representing the vertex. Incident edges are split into triangles using their other end and the appropriate edge from the new vertex-face. To see the result of this algorithm, apply MeshSmooth (with default parameters) to an EditableMesh with a single vertex selected.

**Parameters:**

**int vid**

The vertex to SabinDoo.

**float interp**

The proportion from this vertex to the center of each of the incident faces to put the new vertex for that face.

**DWORD selLevel**

If this is equal to MNM\_SL\_FACE, faces without the MN\_SEL flag will not be affected.

**Point3 \*ctr**

This is a list of centers of all the faces, which cannot be NULL. Since this algorithm changes the original faces, merely computing face centers on the fly won't work. (When SabinDooing two vertices on a given face, the second vertex will get bad center information.) Thus the user should compute all the face centers in advance using ComputeCenters or (preferably) ComputeSafeCenters.

**MeshOpProgress \*mop=NULL**

This optional parameter points to an instance of the virtual class MeshOpProgress, which is used to allow the user to abort out of lengthy calculations. (SabinDoo is a lengthy calculation.) See [Class MeshOpProgress](#).

**Tab<Point3> \*offsets=NULL**

The SabinDoo algorithm is typically used to go from one polygonal approximation of a smooth surface to another polygonal approximation of 4 times the size, but it never actually returns the vertices as they would appear on the "limit surface". If this parameter is non-NULL, it's filled in with offsets that will, when added to the vertices in the SabinDoo result, take those vertices onto the limit surface. This makes a smoother result that matches better between different iterations.

(In SabinDooVert, the offsets for the vertices created by this method are filled in. The offsets table is resized if necessary to accommodate this.)

**Prototype:**

**void CubicNURMS(MeshOpProgress \*mop=NULL, Tab<Point3> \*offsets=NULL, DWORD subdivFlags=0);**

**Remarks:**

This method is available in release 3.0 and later only.

This is a tessellation routine. Note that this algorithm roughly quadruples the size of the MNMesh. For an example of how it works, apply 3ds max's MeshSmooth modifier with "NURMS" MeshSmooth Type.

This technique for tessellation is based loosely on a 1998 SIGGraph paper, "Non-Uniform Recursive Subdivision Surfaces", by Sederberg, Zheng, Sewell, and Sabin, with additional work by Autodesk staff. Topologically, it's quite simple, as it creates a vertex in the center of every edge and face and connects these vertices with quads. However, the geometry is a non-uniform rational extension of the old Catmull-Clark standard cubic subdivision.

To set the vertex and edge weights used by the algorithm, use the EdgeKnots and VertexWeights methods of MNMesh. Like other subdivision routines, this routine only subdivides areas of the MNMesh indicated by the MN\_TARG flag on vertices, and uses the MNEdge::Uncrossable method to determine where the regional boundaries should be.

**Parameters:**

**MeshOpProgress \*mop=NULL**

See [Class MeshOpProgress](#) – provides a way to abort the calculation.

**Tab<Point3> \*offsets=NULL**

Unused for now.

**DWORD subdivFlags=0**

This parameter is available in release 4.0 and later only.

There is currently only one flag for the extra argument,

**MNM\_SUBDIV\_NEWMAP**. If this flag is set, the new mapping scheme will be used.

**Prototype:**

**BOOL CheckForDoubledMappingVerts();**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used as a debugging tool. All double mapping vertices will be debugprinted. "Doubled" mapping verts are individual map vertices that are used to correspond to different regular vertices. For example, the standard map on the box object uses the same mapping vertices on the top as on the bottom. This can be a problem if the user wants to change the map in one region, but not the other. It also causes problems in NURMS MeshSmooth, where the user can change weights in one region but not in the other.

**Prototype:**

**void EliminateDoubledMappingVerts();**

**Remarks:**

This method is available in release 4.0 and later only.

This method will allow you to eliminate double mapping vertices. "Doubled" mapping verts are individual map vertices that are used to correspond to different regular vertices. For example, the standard map on the box object uses the same mapping vertices on the top as on the bottom. This can be a problem if the user wants to change the map in one region, but not the other. It also causes problems in NURMS MeshSmooth, where the user can change weights in one region but not in the other.

**Prototype:**

**void EliminateIsoMapVerts();**

**Remarks:**

This method is available in release 4.0 and later only.

Eliminates mapping vertices that aren't used by any mapping faces.

**Prototype:**

**void EliminateIsoMapVerts(int mp);**

**Remarks:**

This method is available in release 4.0 and later only.

Eliminates mapping vertices that aren't used by any mapping faces.

**Parameters:**

**int mp**

If  $mp \geq 0$ , it indicates the mapping channel that should have its isolated verts removed. If left at the default, all mapping channels are cleared of iso verts.

## **Welding methods**

**Prototype:**

**bool WeldVerts(int a, int b);**

**Remarks:**

This method is available in release 2.5 and later only.

Welds vertices a and b, correcting the mesh topology to match. All references to b are moved to a, and b is killed. If there is a topological problem preventing the weld from occurring, the method does nothing and returns FALSE. If there is no such problem, the weld goes ahead and returns TRUE. Note that if a and b are joined by an edge, this method just calls WeldEdge on that edge.

**Prototype:**

**bool WeldEdge(int ee);**

**Remarks:**

This method is available in release 2.5 and later only.

Welds the endpoints of edge ee, correcting the mesh topology to match. All references to  $E(ee) \rightarrow v2$  are moved to  $E(ee) \rightarrow v1$ , and both ee and  $E(ee) \rightarrow v2$

are killed. If there is a topological problem preventing the weld from occurring, the method does nothing and returns FALSE. If there is no such problem, the weld goes ahead and returns TRUE.

**Prototype:**

**bool WeldBorderVerts (int v1, int v2, Point3 \*destination);**

**Remarks:**

This method is available in release 4.0 and later only.

This method welds vertices v1 and v2 together. Both vertices must be on borders, and they cannot share an edge. If destination isn't NULL, it indicates where the welded vertex should be located. (If it is NULL, the welded vertex is placed at the midpoint of the inputs.)

**Parameters:**

**int v1, v2**

The vertex indices to weld.

**Point3 \*destination**

The destination of the welded vertex.

**Return Value:**

TRUE if something was welded, FALSE otherwise.

**Prototype:**

**bool WeldBorderEdges (int e1, int e2);**

**Remarks:**

This method is available in release 4.0 and later only.

This method welds edges e1 and e2 together. Both edges must be on borders.

**Parameters:**

**int e1, e2**

The two edges you want to weld.

**Return Value:**

TRUE if something was welded, FALSE otherwise.

**Prototype:**

**bool WeldBorderVerts (float thresh, DWORD flag=MN\_SEL);**

**Remarks:**

This method is available in release 4.0 and later only.

This method welds all flagged border vertices within "thresh" of each other. Vertices are only welded pairwise. If vert A's closest target is vert B, but vert B is closer to vert C, verts B and C are welded and A is left out in the cold.

**Parameters:**

**float thresh**

The threshold.

**DWORD flag=MN\_SEL**

This indicates the vertices we look at. If left at the default, selected vertices' colors are analyzed. If flag were to equal **MN\_WHATEVER**, then vertices with the **MN\_WHATEVER** flag would have their colors analyzed.

**Return Value:**

TRUE if something was welded, FALSE otherwise.

**Prototype:**

**bool DetachElementToObject (MNMesh & nmesh, DWORD fflags=MN\_SEL, bool delDetached=true);**

**Remarks:**

This method is available in release 4.0 and later only.

Detaches specified faces (and related vertices and edges) to a new MNMesh. As indicated by the name, the specified region should be a single element; crashes will occur if faces that are flagged share vertices or edges with nonflagged faces.

**Parameters:**

**MNMesh & nmesh**

The new mesh. This is assumed to be empty. The flagged elements are put here, and reindexed so there are no unused components. (This occurs for all map channels as well - only the necessary map vertices are copied over.)

**DWORD fflags**

The flags that indicate the faces which compose the element we want to detach.

**bool delDetached=true**

Indicates whether the specified element should be deleted from this MNMesh.

**Return Value:**

true if anything was detached, false otherwise.

## **Boolean operations**

**Prototype:**

**void PrepForBoolean();**

**Remarks:**

This method is available in release 2.5 and later only.

Prepares a MNMesh for a Boolean operation. This is required for successful Booleans. Generally, it makes the mesh into a convex poly mesh, removing any hidden vertices, finds open regions, and collapses dead structures.

**Prototype:**

**bool BooleanCut(MNMesh & m2, int cutType, int fstart=0,  
MeshOpProgress \*mop=NULL);**

**Remarks:**

This method is available in release 3.0 and later only.

BooleanCut uses the same techniques as MakeBoolean to cut the faces of this with the faces of m2. As with Boolean, both this and m2 should be prepared with the PrepForBoolean method.

**Parameters:**

**MNMesh & m2**

The MNMesh to use to cut this one.

**int cutType**

One of the following:

**BOOLOP\_CUT\_REFINE**

Slice the faces of this with the surface of m2, but do not separate the mesh along the seam.

**BOOLOP\_CUT\_SEPARATE**

Slice and separate the faces of this along the surface of m2.

## **BOOLOP\_CUT\_REMOVE\_IN**

Slice the faces of this with the surface of m2, then remove all the faces of this that are inside m2.

## **BOOLOP\_CUT\_REMOVE\_OUT**

Slice the faces of this with the surface of m2, then remove all the faces of this that are inside m2.

### **int fstart=0**

Indicates the face (of this) we should start checking for cuts by m2. This can be useful if, for example, you know that the first 100 faces of this are nowhere near m2.

### **MeshOpProgress \*mop=NULL**

See [Class MeshOpProgress](#). Provides a way to abort the calculation.

### **Return Value:**

TRUE if finished successfully; FALSE if aborted by the MeshOpProgress.

### **Prototype:**

```
bool MakeBoolean(MNMesh & m1, MNMesh & m2, int type,  
MeshOpProgress *mop=NULL);
```

### **Remarks:**

This method is available in release 2.5 and later only.

Makes this MNMesh into a Boolean result of the given MNMeshes. The operands are not modified during this process. The Boolean algorithm is identical to the one used in the Boolean 2 compound object. Notice that there are no transform arguments, as there are in the previously used CalcBoolOp – transforms should be applied to the operands beforehand using the MNMesh::Transform method.

### **Parameters:**

#### **MNMesh & m1**

The first operand mesh, which should have had PrepForBoolean called.

#### **MNMesh & m2**

The second operand mesh, which should have had PrepForBoolean called.

#### **int type**

The type of Boolean. The Boolean types are defined in mesh.h, and are one of MESHBOOL\_UNION, MESHBOOL\_INTERSECTION, or

MESHBOOL\_DIFFERENCE. If MESHBOOL\_DIFFERENCE is selected, operand m2 is subtracted from operand m1. (To get the opposite result, just switch the order of m1 and m2.)

### **MeshOpProgress \*mop=NULL**

A pointer to a MeshOpProgress. See [Class MeshOpProgress](#) for details. The mop is initialized with the number of faces in mesh 1 plus the number of faces in mesh 2 plus 10. MakeBoolean aborts acceptably if mop->Progress returns FALSE, in which case the partially Booleaned MNMesh is returned. (Usually this consists of mesh 1 partially cut by mesh 2, or mesh 1 fully cut and mesh 2 partially cut, with no faces removed.) This argument may safely be left at NULL if you do not wish to be updated on the Boolean progress or have the capability to abort.

### **Return Value:**

Returns FALSE if the operation was cancelled by the MeshOpProgress or if it was unable to match the seams of the two operands at the end, resulting in a mesh with holes. It returns TRUE if everything went well, producing a solid, hole-free mesh.

Note that it will always return FALSE if one of the operands has holes.

### **Sample Code:**

Sample code: the following code can be used to replace the old CalcBoolOp method.

```
BOOL CalcNewBooleanOp(Mesh & mesh, Mesh & mesh1, Mesh  
& mesh2, int op, MeshOpProgress *prog=NULL, Matrix3  
*tm1=NULL, Matrix3 *tm2=NULL, int whichinv=0) {  
    MNMesh m1(mesh1);  
    MNMesh m2(mesh2);  
    if(tm1) m1.Transform(*tm1);  
    if(tm2) m2.Transform(*tm2);  
    m1.PrepForBoolean();  
    m2.PrepForBoolean();  
    MNMesh mOut;  
    mOut.MakeBoolean(m1, m2, op, prog);  
    if(whichinv==0) mOut.Transform(Inverse(*tm1));  
    if(whichinv==1) mOut.Transform(Inverse(*tm2));  
    mOut.OutToTri(mesh);
```

```
    return TRUE;
}
```

**Prototype:**

```
void ConnectLoops (Tab<int> & loop1, Tab<int> & loop2, int segs,  
float tension, DWORD smGroup, MtlID mat, bool sm_ends);
```

**Remarks:**

This method is available in release 4.0 and later only.

Connects two border loops by filling in new geometry between them, as is done in the Connect compound object.

**Parameters:**

**Tab<int> & loop1**

**Tab<int> & loop2**

These parameters, which are interchangeable, are lists of edges that represent border loops in the MNMesh. It is assumed that the MNMesh is filled in, that each of the edges in the loops is in fact one-sided ( $e[\text{loop1}[i]].f2 == -1$ ), and that the edges go in counterclockwise order when looking from outside the mesh (so  $e[\text{loop1}[i]].v1 == e[\text{loop1}[i+1]].v2$ ). (These are the sort of border loops you find in an MNMeshBorder class.)

**int segs**

The number of segments in the bridge.

**float tension**

The tension of the bridge - this controls how much the surface tangents at each end of the bridge affect the bridge's shape.

**DWORD smGroup**

These smoothing group(s) should be applied to the entire bridge. (Use 0 for a faceted bridge.)

**MtlID mat**

This is the material used for the bridge faces.

**bool sm\_ends**

If TRUE, additional smoothing groups are applied to the end faces of the bridge to ensure that the bridge smoothes with the existing faces around each border loop.

**Prototype:**

```
void Connect (MNMeshBorder & borderList, int segs, float  
tension, bool sm_bridge, bool sm_ends, Tab<int> *vsep=NULL);
```

**Remarks:**

This method is available in release 4.0 and later only.

Connect automatically figures out which loops in the given MNMeshBorder would make good candidates for connecting, and calls ConnectLoop on them all to connect them. Good candidates are pairs of loops that face each other. Centers and normals of each of the "holes" are compared to find the best matches.

**Parameters:****MNMeshBorder & borderList**

The boundary information for the MNMesh mesh.

**int segs**

The number of segments in each bridge.

**float tension**

The tension of each bridge - this controls how much the surface tangents at each end of the bridge affect the bridge's shape.

**bool sm\_bridge**

If TRUE, each bridge should be smoothed with some unused smoothing group.

**bool sm\_ends**

If TRUE, additional smoothing groups are applied to the end faces of each bridge to ensure that the bridge smoothes with the existing faces around each border loop.

**Tab<int> \*vsep=NULL**

If not NULL, this points to a "separation list" of vertices. Frequently the user wishes to connect two distinct object (as with the Connect compound object). In these cases, the vertices of the different connected components are often grouped in distinct sets: vertices 0 through 26 come from the first component, 27 through 118 from the second, etc. Generally the user would not want a component to connect to itself, but rather only to other components. So if non-NULL, this is assumed to point to a list describing the vertex ranges for each component - (0, 27, 119, etc). The list should have size (number of

components+1), and the last element should be MNMesh::numv. Given such an input, the Connect algorithm will only connect loops to loops in other components.

**Prototype:**

**void InvalidateGeomCache();**

**Remarks:**

This method is available in release 3.0 and later only.

Invalidates information, like bounding box and face normals, that's dependent on Geometry. Call after changing the geometry.

**Prototype:**

**void InvalidateTopoCache();**

**Remarks:**

This method is available in release 3.0 and later only.

Invalidates information, like the edge list and the render verts, that's dependent on Topology. Clears the MN\_MESH\_FILLED\_IN and related flags in MN\_MESH\_CACHE\_FLAGS.

**Prototype:**

**void allocRVerts();**

**Remarks:**

This method is available in release 3.0 and later only.

Allocates the "render vertices" used to optimize display routines. Called by the system as needed.

**Prototype:**

**void updateRVerts(GraphicsWindow \*gw);**

**Remarks:**

This method is available in release 3.0 and later only.

Fills in the render vertices with display coordinates based on the specified view.

**Prototype:****void freeRVerts();****Remarks:**

This method is available in release 3.0 and later only.

Frees the "render vertices" used to optimize display routines.

**Debugging Methods**

Please see the [MNMesh Notes on Debugging](#) for more information.

**Prototype:****void MNDebugPrint(bool triprint=FALSE);****Remarks:**

This method is available in release 2.5 and later only.

Uses DebugPrint to print out the entire MNMesh to the Debug Results window in DevStudio. This can be useful for tracking down bugs. Be careful not to leave MNDebugPrint calls in your final build; they will slow down your effect to no purpose.

**Parameters:****bool triprint=FALSE**

Controls whether or not triangulation information is printed out for each face. Usually this information isn't desired.

**Prototype:****void MNDebugPrintVertexNeighborhood(int vv, bool triprint=FALSE);****Remarks:**

This method is available in release 2.5 and later only.

Uses DebugPrint to print out information about all the edges, faces, and vertices in the immediate vicinity of vertex vv.

**Parameters:****int vv**

The vertex whose information is output.

**bool triprint=FALSE**

Controls whether or not triangulation information is printed out for each face.

Usually this information isn't desired.

**Prototype:**

**bool CheckAllData();**

**Remarks:**

This method is available in release 2.5 and later only.

Thoroughly checks a MNMesh to make sure that no components refer to dead components; that when one component refers to another, the other refers back to the first; and that orientation is correctly matched between faces and edges. If everything checks out, TRUE is returned. If any errors are detected, DebugPrint is used to send a message to the DevStudio Debug Results window and FALSE is returned. Be careful not to leave CheckAllData calls in your final build; they will slow down your effect to no purpose.

Here is the sort of debug output you can expect. CheckAllData always outputs one of the following messages:

Checking all data in filled-in MNMesh: (if the mesh has the MN\_MESH\_FILLED\_IN flag set)

Checking all data in MNMesh: (if the mesh does not)

These errors will be detected for any mesh:

Face %d has an out-of-range vertex: %d

Face %d has an out-of-range tvert: %d

Face %d has an out-of-range cvert: %d

Face %d refs dead edge %d.

Face %d has bad triangulation index: %d

These errors will be detected only for filled-in meshes:

Face %d has an out-of-range edge: %d

Face %d uses dead edge %d.

Face %d refs edge %d, but edge doesn't ref face.

Face %d refs edge %d from vert %d to vert %d, but edge doesn't go there.

Orientation mismatch between face %d and edge %d.

Face %d refs vertex %d, but vertex doesn't ref face.

Edge %d refs dead vertex %d.

Edge %d refs vertex %d, but vertex doesn't ref edge.

Edge %d has no face-1

Edge %d refs dead face %d.

Edge %d refs face %d, but face doesn't ref edge.

Vertex %d refs dead edge %d.

Vertex %d refs edge %d, but edge doesn't ref vertex.

Vertex %d references more faces than edges.

Vertex %d refs dead face %d.

Vertex %d refs face %d, but face doesn't ref vertex.

## Operators

### Prototype:

**MNMesh & operator=(const MNMesh & from);**

### Remarks:

Assignment operator. Allocates space & copies over all data from "from".

### Prototype:

**MNMesh & operator+=(const MNMesh & from);**

### Remarks:

Union operator. Adds all data from "from" to this MNMesh. Flags MN\_MESH\_NONTRI and MN\_MESH\_RATSNEST are or'd together, while flags MN\_MESH\_FILLED\_IN, MN\_MESH\_NO\_BAD\_VERTS, and MN\_MESH\_VERTS\_ORDERED are and'd together. (Uniting a rat's nest with a non-rat's nest makes a rat's nest, but uniting a mesh with mapping coordinates and one without makes one without.)

### Prototype:

**void ClearSpecifiedNormals ();**

### Remarks:

This method is available in release 5 and later only.

Clears out the specified normal interface, if present. Removes it

completely, so it won't flow up the stack, etc.

**Prototype:**

**void SpecifyNormals ();**

**Remarks:**

This method is available in release 5 and later only.

Creates the user-specified normal interface in this mesh. Initializes the MNNormalSpec's "Parent" to this mesh, but does not allocate normal faces or otherwise prepare the normals. Note that this interface will flow up the pipeline, in the PART\_GEOM and PART\_TOPO channels.

**Prototype:**

MNNormalSpec \*GetSpecifiedNormals();

**Remarks:**

This method is available in release 5 and later only.

Returns a pointer to the user-specified normal interface, if present.

**Return Value:**

A pointer to this mesh's MNNormalSpec interface, or NULL if the interface has not been created in this mesh.

**Prototype:**

**MNNormalSpec \*GetSpecifiedNormalsForDisplay();**

**Remarks:**

This method is available in release 5 and later only.

Returns a non-NULL MNNormalSpec interface pointer only if the interface is present, and is prepared for use in display - otherwise, it returns NULL, and we fall back on the smoothing groups.

A MNNormalSpec is considered "prepared for display" only if the MNNORMAL\_NORMALS\_BUILT and MNNORMAL\_NORMALS\_COMPUTED flags are set.  
(See the MNNormalSpec methods BuildNormals and ComputeNormals.)

**Prototype:**

**void CopyBasics (const MNMesh & from, bool copyEdges = false);**

**Remarks:**

This method is available in release 4.5 and later only.  
Copies bare geometry only - no per-vertex data, maps, or normals.  
Useful for keeping low-memory local caches of a mesh where map, vertex data, edge data, normals, and other interfaces are not required.

**Parameters:**

**const MNMesh & from**

The mesh to copy into this mesh.

**bool copyEdges = false**

Indicates if the winged-edge array, as well as vertex and face references to edges, should be copied. If false, the MN\_MESH\_FILLED\_IN flag is cleared in this mesh.

## Structure NotifyInfo

See Also: [Class Interface \(callback methods\)](#), [List of Notification Codes](#).

### Description:

3ds max supports a system where a plug-in can ask to receive a callback when certain events occur. These are events such as the system unit settings changing, system time setting changing, or the user executing File/Reset, File/New, etc.

This structure is part of this system. It is available in release 2.0 and later only.

```
typedef struct {  
    int intcode;  
    void *callParam;
```

This pointer is available in release 3.0 and later only.

This parameter can be passed in with the function

```
BroadcastNotification(int code, void *callParam).
```

```
} NotifyInfo;
```

The plug-in creates a callback function to process the notification. The notification callback function (**NOTIFYPROC**) is defined as follows:

```
typedef void (* NOTIFYPROC)(void *param, NotifyInfo *info);
```

The **NotifyInfo** structure is passed to the **NOTIFYPROC** to inform it of what it's being notified about.

The sample code below shows how this system may be used.

### Sample Code:

```
// Declare the callback function  
static void TimeUnitsChanged(void *param, NotifyInfo *info) {  
    // Handle the units changing...  
}  
  
// Register the callback  
RegisterNotification(TimeUnitsChanged,this,  
    NOTIFY_TIMEUNITS_CHANGE);  
  
// When done, unregister the callback  
UnRegisterNotification(TimeUnitsChanged,this,  
    NOTIFY_TIMEUNITS_CHANGE);
```

## Related Functions:

### Function:

**int RegisterNotification(NOTIFYPROC proc, void \*param, int code);**

### Remarks:

This global function is called to establish the connection between the event and the callback.

### Parameters:

**NOTIFYPROC proc**

The callback function called when the event occurs.

**void \*param**

A pointer to a parameter which will be passed to the callback function.

**int code**

Specifies which notification to register. See [List of Notification Codes](#).

### Return Value:

Nonzero if the event was registered; otherwise zero.

### Function:

**int UnRegisterNotification(NOTIFYPROC proc, void \*param, int code);**

### Remarks:

This global function is called to break the connection between the event and the callback. After this function executes the callback is no longer invoked when the event occurs.

### Parameters:

**NOTIFYPROC proc**

The callback function called when the event occurs.

**void \*param**

This parameter must be identical to the param sent into

**RegisterNotification()**. This function will only unregister a callback if this parameter equals the param sent in to the **RegisterNotification()** function.

**int code**

Specifies which notification to unregister. See [List of Notification Codes](#).

**Return Value:**

Nonzero if the event was unregistered; otherwise zero.

**Function:**

**void BroadcastNotification(int code);**

**Remarks:**

Calling this global function causes the callback corresponding to the specified code to be called.

**Parameters:**

**int code**

Specifies which notification to broadcast. See [List of Notification Codes](#).

**Prototype:**

**void BroadcastNotification(int code, void \*callParam);**

**Remarks:**

This global function is available in release 3.0 and later only.

This causes the callback corresponding to the specified code to be called and passes the specified **void\*** parameter along to the callback.

**Parameters:**

**int code**

Specifies which notification to broadcast. See [List of Notification Codes](#).

**void \*callParam**

This parameter is passed to the callback. See the code

**NOTIFY\_BITMAP\_CHANGED** for an example of this in use.

**Function:**

**int UnRegisterNotification(NOTIFYPROC proc, void \*param);**

**Remarks:**

This global function unregisters the callback from **all** codes

**Parameters:**

**NOTIFYPROC proc**

The callback function called when the event occurs.

**void \*param**

A pointer to a parameter which will be passed to the callback function.

**Return Value:**

Nonzero if the events were unregistered; otherwise zero.

## Class SimpleOSMToWSMObject

See Also: [Class SimpleWSMObject](#), [Class SimpleMod](#), [Class Deformer](#), [Class IParamMap](#).

```
class SimpleOSMToWSMObject : public SimpleWSMObject
```

### Description:

This class is used to allow any Object Space Modifier derived from **SimpleMod** to easily be turned into a World Space Modifier (Space Warp).

This is very simple to do because a modifier version already contains just about everything that needs to be done. This is because the modifier works the same -- it is just in world space instead of object space.

All a developer needs to do to turn their **SimpleMod** modifier into the WSM version is implement a class derived from this one and call the **SimpleOSMToWSMObject** constructor from their constructor. See the sample code below (for the full sample code using this class see `\MAXSDK\SAMPLES\MODIFIERS\BEND.CPP`).

```
class BendWSM : public SimpleOSMToWSMObject {
public:
    BendWSM() {}
    BendWSM(BendMod *m) : SimpleOSMToWSMObject(m) {}
    void DeleteThis() { delete this; }
    SClass_ID SuperClassID() {return
WSM_OBJECT_CLASS_ID;}
    Class_ID ClassID() {return BENDWSM_CLASSID;}
    TCHAR *GetObjectName() {return
GetString(IDS_RB_BEND2);}
};
```

These new modifier-based space warps are accessed in the drop-down category list of the Space Warps branch of the Create command panel. Choose Modifier-Based from the list to display buttons for each of the new space warps.

### Data Members:

```
public:
```

```
    SimpleMod *mod;
```

Points to the simple modifier instance this is based on.

**static IParamMap \*pmapParam;**

Points to the parameter map used to handle the user interface for this WSM.

These are the parameter block indices for the pmap:

**#define PB\_OSMTOWSM\_LENGTH 0**

**#define PB\_OSMTOWSM\_WIDTH 1**

**#define PB\_OSMTOWSM\_HEIGHT 2**

**#define PB\_OSMTOWSM\_DECAY 3**

**Methods:**

**Prototype:**

**SimpleOSMToWSMObject();**

**Remarks:**

Constructor.

**Prototype:**

**SimpleOSMToWSMObject(SimpleMod \*m);**

**Remarks:**

Constructor.

**Parameters:**

**SimpleMod \*m**

This is a pointer to the **SimpleMod** instance this WSM is based on.

**Prototype:**

**Deformer &GetDecayDeformer(TimeValue t,Deformer  
&mdef,Point3 origin,Interval &iv);**

**Remarks:**

Implemented by the System.

This class enhances the deformation done by the object space modifier to include a decay parameter. This allows the deformation to decay over distance. This helper method is used internally in this.

## Class ImageFilterInfo

See Also: [Class ImageFilter](#), [Class BitmapInfo](#), [Class FrameRange](#), [Class ITrackViewNode](#), [Class Class\\_ID](#).

class ImageFilterInfo

### Description:

This class provides information to an image filter plug-in. This is information such as state of any masks used, and the various frame ranges for the video post queue. It is analogous to the BitmapInfo class in the Bitmap Manager.

### Data Members:

public:

#### **BOOL maskenabled;**

TRUE if the filter has a mask; otherwise FALSE.

#### **BOOL evCopy;**

This is used internally as a flag indicating this object is a temporary copy, and not the real thing. It is only an issue when filters have Track View Nodes. This is only used internally.

#### **BOOL invertedmask;**

TRUE if the mask is inverted; otherwise FALSE.

#### **BitmapInfo mask;**

The image used as the mask.

#### **WORD maskflag;**

This is used internally. It indicates what part of the mask image is used to create the grayscale mask. It may be one of the following values: **MASK\_R**, **MASK\_G**, **MASK\_B**, **MASK\_A**, **MASK\_L**, **MASK\_Z**, **MASK\_MTL\_ID**, **MASK\_NODE\_ID**.

#### **BitmapInfo imgQueue;**

This is a BitmapInfo that holds information about the current Video Post main queue image buffer. This can be used to get Video Post's (or the target image's) resolution, etc.

#### **FrameRange QueueRange;**

This defines the entire Video Post Queue range. This is the range defined between VP Start Time and VP End Time in the video post user interface.

**FrameRange ExecutionRange;**

When the queue is executed, this is the range of frames being rendered.

**FrameRange FilterRange;**

The FilterRange is where this filter starts and ends.

**Methods:****Prototype:**

**virtual void SetResource(const TCHAR \*n);**

**Remarks:**

Filters may want to identify themselves by something more specific than their names when they appear in the video post queue. By default, the name of a filter is used to identify it in the video post queue, ie the Negative filter appears as Negative. Some filters may want a more descriptive name to appear. For instance a gradient filter that allows the user to save named settings may want the name of the set to appear rather than simply the name of the filter itself. Thus, "Flowing gradient - Red to Blue" may appear rather than "Gradient". This method is available for filters that that give such names to parameter sets. If not empty, the resource name will be used to identify the filter in the Video Post Queue. This is saved along with everything else by the system (3ds max).

**Parameters:**

**const TCHAR \*n**

The name to appear, instead of the filter name, in the video post queue.

**Prototype:**

**virtual const TCHAR \*Resource();**

**Remarks:**

Returns the resource name.

**Prototype:**

**virtual const TCHAR \*Name();**

**Remarks:**

Returns the name of the filter.

**Operators:**

**Prototype:**

**virtual ImageFilterInfo &operator= ( ImageFilterInfo &from );**

**Remarks:**

Assignment operator.

**Parameters:**

**ImageFilterInfo &from**

The source ImageFilterInfo.

**Prototype:**

**ITrackViewNode \*Node();**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used to return the Track View node for this filter. Because Video Post Filter plug-ins have a short life, in other words, they are only loaded when they are actually needed and deleted right after, the Track View node information is kept in the **ImageFilterInfo** class kept by Video Post for each filter event.

**Prototype:**

**void SetNode(ITrackViewNode \*n);**

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the Track View node associated with this ImageFilter.

**Parameters:**

**ITrackViewNode \*n**

The Track View node to set.

**Prototype:**

**Class\_ID NodeID();**

**Remarks:**

This method is available in release 2.0 and later only.  
Returns the Class\_ID of the Track View node (if any).

**Prototype:**

**void SetNodeID(Class\_ID id);**

**Remarks:**

This method is available in release 2.0 and later only.  
Sets the stored Class\_ID of the Track View node (if any).

**Parameters:**

**Class\_ID id**

The id to set.

**Prototype:**

**TCHAR \*UserLabel();**

**Remarks:**

This method is available in release 2.0 and later only.  
Returns the optional label entered by the user while adding or editing a filter.  
This label replaces the filter's name in Video Post's tracks in Track View for easier identification. This is the name that is entered in the 'Edit Filter Event' dialog Filter Plug-In Label field. The label defaults to Unnamed in which case the Filter's name appears (for example 'Negative').

**Prototype:**

**int FilterType();**

**Remarks:**

This method is available in release 2.0 and later only.  
This method is used so dual mode filters can detect what mode they are running in (as a filter or as a compositor). In the 3ds max 1.x SDK, filters that ran both as filters and layers had no way to determine what mode they were running while in "Setup" mode (in **ShowControl()**). At run time they would check for a foreground bitmap. If it was NULL, they were to assume they were running as simple filters. Now this method may be used to determine what mode they are running in.

**Return Value:**

One of the following values:

**FLT\_FILTER**

**FLT\_LAYER**

**Prototype:**

**void SetFilterType(int type);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used internally.

## Class ITrackViewNode

See Also: [Class ReferenceTarget](#), [Class Interface](#), [Class Control](#), [Class ImageFilter](#), [Class TVNodeNotify](#).

```
class ITrackViewNode : public ReferenceTarget
```

### Description:

This class provides an interface to Track View Nodes. A Track View Node is simply a class that has zero or more sub-track view nodes and zero or more sub-controllers. This is mainly used to provide a place for Global Variable tracks (labeled "Global Tracks" in Track View) and Video Post tracks (labelled "Video Post" in Track View).

The TrackViewNode sub-nodes and sub-controllers are identified by a unique ID in the form of a Class\_ID variable. This does not necessarily have to be the Class\_ID of an existing plug-in, however plug-ins may wish to use their Class\_ID for any items they add to be sure they are unique.

The Interface class provides access to the root track view node:

```
virtual ITrackViewNode *GetTrackViewRootNode()=0;
```

From the root track view node, new nodes may be added. There are two defined sub nodes identified by the following **#defined** Class\_IDs:

```
#define GLOBAL_VAR_TVNODE_CLASS_ID
```

```
Class_ID(0xb27e9f2a, 0x73fad370)
```

```
#define VIDEO_POST_TVNODE_CLASS_ID
```

```
Class_ID(0x482b8d30, 0xb72c8511)
```

These can be retrieved by calling **GetNode()** on the track view root node and passing in one of the above IDs.

All methods of this class are implemented by the system.

Note: Developers can also create their own track view node using the following global function:

### Function:

```
ITrackViewNode *CreateITrackViewNode(BOOL  
hidden=FALSE);
```

### Remarks:

This method is used to add a track view node.

**Parameters:**

**BOOL hidden=FALSE**

If FALSE the node is hidden; otherwise it is visible in the viewports.

**Methods:**

**Prototype:**

```
virtual void AddNode(ITrackViewNode *node, TCHAR *name,  
Class_ID cid, int pos=TVNODE_APPEND)=0;
```

**Remarks:**

This method is used to add a track view node.

**Parameters:**

**ITrackViewNode \*node**

Points to the Track View Node to add.

**TCHAR \*name**

The name for the node that appears in Track View.

**Class\_ID cid**

The Class\_ID which identifies the plug-in that added the node.

**int pos=TVNODE\_APPEND**

The position in the list of nodes where this one is added. If this defaults to **TVNODE\_APPEND** the node is added at the end of the list.

**Prototype:**

```
virtual void AddController(Control *c, TCHAR *name, Class_ID  
cid, int pos=TVNODE_APPEND)=0;
```

**Remarks:**

This method is used to add a track view controller.

**Parameters:**

**Control \*c**

Points to the controller to add.

**TCHAR \*name**

The name that will appear in Track View.

**Class\_ID cid**

The Class\_ID of the plug-in that adds the controller.

**int pos=TVNODE\_APPEND**

The position in the list where the controller is added. If this defaults to **TVNODE\_APPEND** the controller is added at the end of the list.

**Prototype:****virtual int FindItem(Class\_ID cid)=0;****Remarks:**

A Track View Node maintains a table that contains the sub-nodes and sub-controllers. This method returns the index into the table of the node or controller whose Class\_ID is passed. If the Class\_ID could not be found then -1 is returned.

**Parameters:****Class\_ID cid**

The Class\_ID to find.

**Prototype:****virtual void RemoveItem(int i)=0;****Remarks:**

A Track View Node maintains a table that contains the sub-nodes and sub-controllers. This method removes the 'i-th' sub-node or sub-controller of the table.

**Parameters:****int i**

The zero based index into the table of the item to remove.

**Prototype:****virtual void RemoveItem(Class\_ID cid)=0;****Remarks:**

A Track View Node maintains a table that contains the sub-nodes and sub-controllers. This method removes the sub-node or sub-controller whose

Class\_ID is passed from the table.

**Parameters:**

**Class\_ID cid**

The Class\_ID used when the node or controller was added.

**Prototype:**

**virtual Control \*GetController(int i)=0;**

**Remarks:**

This method returns a pointer to the 'I-th' sub-controller.

**Parameters:**

**int i**

The zero based index of the sub-controller.

**Prototype:**

**virtual Control \*GetController(Class\_ID cid)=0;**

**Remarks:**

This method returns a pointer to the sub-controller whose Class\_ID is passed.

**Parameters:**

**Class\_ID cid**

The Class\_ID used when the controller was added.

**Prototype:**

**virtual ITrackViewNode \*GetNode(int i)=0;**

**Remarks:**

This method returns a pointer to the 'i-th' sub-node.

**Parameters:**

**int i**

The zero based index of the sub-node.

**Prototype:**

**virtual ITrackViewNode \*GetNode(Class\_ID cid)=0;**

**Remarks:**

This method returns a pointer to the sub-node whose Class\_ID is passed.

**Parameters:**

**Class\_ID cid**

The Class\_ID used when the controller was added.

**Prototype:**

**virtual int NumItems()=0;**

**Remarks:**

This method returns the total number of sub-nodes and/or sub-controllers in the table.

**Prototype:**

**virtual void SwapPositions(int i1, int i2)=0;**

**Remarks:**

This method is used to rearrange the elements in the table so item **i1** is where **i2** was and **i2** is where **i1** was.

**Parameters:**

**int i1**

The zero based index into the table of one of the items to swap.

**int i2**

The zero based index into the table of the other item to swap.

**Prototype:**

**virtual TCHAR \*GetName(int i)=0;**

**Remarks:**

Returns the name of the 'i-th' sub-node or sub-controller.

**Parameters:**

**int i**

The zero based index into the table of the item whose name to return.

**Prototype:**

**virtual void SetName(int i,TCHAR \*name)=0;**

**Remarks:**

Sets the name of the 'i-th' sub-node or sub-controller to the name passed.

**Parameters:**

**int i**

The zero based index into the table of the item whose name to set.

**TCHAR \*name**

The new name for the sub-node or sub-controller.

**Prototype:**

**virtual void RegisterTVNodeNotify(TVNodeNotify \*notify)=0;**

**Remarks:**

Registers the track view notify callback object so it receives reference messages.

**Parameters:**

**TVNodeNotify \*notify**

Points to the callback object to register.

**Prototype:**

**virtual void UnRegisterTVNodeNotify(TVNodeNotify \*notify)=0;**

**Remarks:**

Un-Registers the track view notify callback object.

**Parameters:**

**TVNodeNotify \*notify**

Points to the callback object to register.

**Prototype:**

**virtual void HideChildren(BOOL chide)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This may be used to prevent child nodes from showing up in Track View.

**Parameters:**

**BOOL `hide`**

Pass TRUE to have children hidden; FALSE to have them visible.

## List of Image (G-Buffer) Channels

See Also: [Class GBuffer](#), [Class ImageFilter](#), [Class Bitmap](#), [Class MtlBase](#), [Class Interface](#), [Class INode](#), [Structure RealPixel](#), [Structure Color24](#).

Below is an overview of the image channels. The number of bits per pixel occupied by the channel is listed. The way the channel is accessed and cast to the appropriate data type is also shown.

Note: 3ds max users may store the G-Buffer data in offline storage in RLA or RPF files. For the definition of the RLA or RPF format you can look at the source code in `\MAXSDK\SAMPLES\IO\RLA\RLA.CPP`.

Also Note: The term 'fragment' is used in the descriptions of the channels below. A 'fragment' is the portion of a triangle of a mesh that's seen by a particular pixel being rendered. It's as if the pixel was a cookie-cutter and chopped a visible section of the triangle off for rendering -- that cut piece is called a fragment.

### **BMM\_CHAN\_Z**

Z-buffer, stored as a float. The size is 32 bits per pixel. This is the channel that would be used by a depth of field blur routine for instance. The Z value is at the center of the fragment that is foremost in the sorted list of a-buffer fragments. The Z buffer is an array of float values giving the Z-coordinate in camera space of the point where a ray from the camera through the pixel center first intersects a surface. All Z values are negative, with more negative numbers representing points that are farther from the camera. The Z buffer is initialized with the value **-1.0E30**. Note that this is a change over 3ds max 1.x where the Z buffer was previously initialized with **1.0E30**. The negative value is more appropriate since more negative values represent points farther from the camera.

Note that for non-camera viewports (such as Front, User, Grid, Shape, etc.) the values may be both positive and negative. In such cases the developer may as well add a large value onto all the values to make them all positive. This is because positive versus negative doesn't really mean anything. It is just the distance between values that matters.

As noted above, the Z values in the A buffer are in camera space. The projection for a point in camera space to a point in screen space is:

```
Point2 RenderInfo::MapCamToScreen(Point3 p) {  
    return (projType==ProjPerspective)?
```

```

Point2(xc + kx*p.x/p.z, yc + ky*p.y/p.z):
Point2(xc + kx*p.x, yc + ky*p.y);
}

```

This function is supplied by the RenderInfo data structure which can be obtained from the bitmap output by the renderer using the function **Bitmap::GetRenderInfo()**. Note that this outputs a **Point2**. There is no projection for Z. As noted before, the Z buffer just uses the camera space Z.

```

float *zbuffer = (float *)GetChannel(BMM_CHAN_Z,type);
BMM_CHAN_MTL_ID

```

The ID assigned to the material via the Material Editor. The size is 8 bits per pixel. This channel is currently settable to a value between 0 and 8 by the 'Material Effects Channel' flyoff in the Material Editor. A plug-in material can generated up to 255 different material ID's (since this is an 8-bit quantity). This channel would be used to apply an effect (i.e., a glow) to a specific material.

```

BYTE *bbuffer = (BYTE
*)GetChannel(BMM_CHAN_MTL_ID,type);
BMM_CHAN_NODE_ID

```

This is the ID assigned to node via the Object Properties / G-buffer ID spinner. The size is 16 bits per pixel. This channel would be used to perform an effect (for example a flare) on a specific node.

```

WORD *wbuffer = (WORD
*)GetChannel(BMM_CHAN_NODE_ID,type);
BMM_CHAN_UV

```

UV coordinates, stored as a Point2. The size is 64 bits per pixel. If you have UV Coordinates on your object this channel provides access to them. This channel could be used by 3D paint programs or image processing routines to affect objects based on their UVs. The UV coordinate is stored as a Point2, using Point2::x for u and Point2::y for v. The UV coordinates are values prior to applying the offset, tiling, and rotation associated with specific texture maps.

```

Point2 *pbuffer = (Point2
*)GetChannel(BMM_CHAN_UV,type);
BMM_CHAN_NORMAL

```

Normal vector in view space, compressed. The size is 32 bits per pixel. Object normals are available for image processing routines that take advantage of the normal vectors to do effects based on curvature (for example), as well as for 3D paint programs. The normal value is at the center of the fragment that is foremost in the sorted list of a-buffer fragments.

**DWORD \*dbuffer = (DWORD \*)GetChannel(BMM\_CHAN\_NORMAL,type);**

Note: The following function is available to decompress this value to a standard normalized **Point3** value (**DWORD** and **ULONG** are both 32 bit quantities):

**Point3 DeCompressNormal(ULONG n);**

The decompressed vector has absolute error < 0.001 in each component.

### **BMM\_CHAN\_REALPIX**

Non clamped colors in "RealPixel" format. The size is 32 bits per pixel. See [Structure RealPixel](#). These are 'real' colors that are available for physically-correct image processing routines to provide optical effects that duplicate the way the retina works.

**RealPixel \*rbuffer =**

**(RealPixel \*)GetChannel(BMM\_CHAN\_REALPIX,type);**

### **BMM\_CHAN\_COVERAGE**

Pixel coverage of the front surface. This provides an 8-bit value (0..255) that gives the coverage of the surface fragment from which the other G-buffer values are obtained. This channel is being written and read with RLA files, and shows up in the Virtual Frame Buffer. This may be used to make the antialiasing in 2.5D plug-ins such as Depth Of Field filters much better.

**UBYTE \*gbufCov =**

**(UBYTE\*)GetChannel(BMM\_CHAN\_COVERAGE,type);**

### **BMM\_CHAN\_BG**

The RGB color of what's behind the front object. The size is 24 bits per pixel.

If you have the image color at a pixel, and the Z coverage at the pixel, then when the Z coverage is < 255, this channel tells you the color of the object that was partially obscured by the foreground object. For example, this info will let you determine what the "real" color of the foreground object was before it was blended (antialiased) into the background.

```
Color24 *bgbuffer = (Color24  
*)GetChannel(BMM_CHAN_BG,type);  
BMM_CHAN_NODE_RENDER_ID
```

System node number (valid during a render). The size is 16 bits per pixel.

The renderer will set the RenderID of all rendered nodes, and will set all non-rendered nodes to **0xffff**. Video Post plug-ins can use the

**Interface::GetINodeFromRenderID()** method to get a node pointer from an ID in this channel. Note that this channel is NOT saved with RLA files, because the IDs would not be meaningful unless the scene was the one rendered.

```
UWORD *renderID = (UWORD  
*)GetChannel(BMM_CHAN_NODE_RENDER_ID,type);  
INode *node = ip->GetINodeFromRenderID(*renderID);  
BMM_CHAN_COLOR
```

This option is available in release 3.0 and later only.

This is the color returned by the material shader for the fragment. It is a 24 bit RGB color (3 bytes per pixel).

```
Color24 *c1 = (Color24  
*)GetChannel(BMM_CHAN_COLOR,type);  
BMM_CHAN_TRANSP
```

This option is available in release 3.0 and later only.

This is the transparency returned by the material shader for the fragment. It is a 24 bit RGB color (3 bytes per pixel).

```
Color24 *transp = (Color24  
*)GetChannel(BMM_CHAN_TRANSP,type);  
BMM_CHAN_VELOC
```

This option is available in release 3.0 and later only.

This gives the velocity vector of the fragment relative to the screen, in screen coordinates. It is a Point 2 (8 bytes per pixel).

```
Point2 *src = (Point2  
*)GetChannel(BMM_CHAN_VELOC,type);  
BMM_CHAN_WEIGHT
```

This option is available in release 3.0 and later only.

This is the sub-pixel weight of a fragment. It is a 24 bit RGB color (3 bytes per pixel). It is the fraction of the total pixel color contributed by the fragment. The sum of (color \*weight) for all the fragments should give the final pixel color. The weight ( which is an RGB triple) for a given fragment takes into account the coverage of the fragment and the transparency of any fragments which are in front of the given fragment.

If c1, c2, c3.. etc are the fragment colors, and w1, w2, w3... etc are the fragment weights, then

$$\text{pixel color} = c1*w1 + c2*w2 + c3*w3 + \dots + cN*wN;$$

The purpose of the sub-pixel weight is to allow post processes to weight the contribution of a post-effect from a particular fragment. It may also be necessary to multiply by the fragment's own transparency, which is *not* included in its weight. Note that for fragments that have no transparent fragments in front of them, the weight will be equal to the coverage.

**Color24 \*w1 = (Color24  
\*)GetChannel(BMM\_CHAN\_WEIGHT,type);**

**BMM\_CHAN\_NONE**

None of the channels above.

**BMM\_CHAN\_MASK**

This option is available in release 4.0 and later only.

The 4x4 (16 bits = 1 word) pixel coverage mask.

# Required Changes to MAX 2.x Plug-Ins to Run in MAX 3.0

See Also: [What's New in the MAX 3.0 SDK](#).

## Overview

This section provides general information on the changes required to all plug-ins to get them running in 3ds max 3.0. It also provides links to topics which discuss the specific changes for many affected plug-in types. Some of these changes are required while others are optional but advantageous.

## Current Compiler Version

A new version of the compiler is required. Visual C++ 6.0 is used for 3ds max 3.0. Prior versions of the compiler are no longer supported.

## DLL and File Loading Changes

The following changes were made affecting the loading of previous version DLLs and 3ds max files:

The value **MAX\_API\_NUM** was changed to 6 and thus DLL's compiled with the R2 API won't load.

The value **MAX\_RELEASE** was changed from 2500 to 3000 to indicate the current release.

The name of the class directory in 3ds max R3 files from was changed from "ClassDirectory2" to "ClassDirectory3". This will prevent 3ds max R3 files from being loaded into 3ds max R2 (3ds max will put up an Invalid File message).

## Library Files Renamed and Eliminated

The old **\MAXSDK\LIB\UTIL.LIB** file has been renamed to **MAXUTIL.LIB**. This prevents a conflict with a library file provided by VC++ 6.0. Therefore, developers will need to update their projects to point to the new LIB file instead.

**PATCH.LIB** has been eliminated from the SDK. Now simply use **CORE.LIB**.

## Parameter Block and Parameter Map Improvements

There is a new scheme for parameter blocks and parameter maps in release 3. While the old system still exists and will run fine, developers are encouraged to support this new system in their plug-ins. This makes them available to the macro recorder, 3ds max Scripter and Schematic View. See the Advanced Topics section [Parameter Blocks and Maps in Release 3](#) for implementation details.

## Miscellaneous Changes in R3 That May Affect Plug-Ins

### Floating Dialogs

This section relates to two common bugs in previous versions of 3ds max and thus likely many plug-ins that have copied or imitated 3ds max's source code. The problems are that modal dialogs aren't "truly" modal, and that modeless dialogs aren't getting disabled when other modal dialogs are presented.

The first condition to be on the look out for is the "Orphaned Dialog" type of bug. An example of this occurs when a user brings up a modal dialog, then hits a 3ds max toolbar button like "Undo", and the object that brought the dialog up goes away. The problem here is that the dialogs aren't "modal enough". That is, when a modal dialog is displayed, one should never be able to hit any toolbar button or menu entry from 3ds max's main user interface. The problem is that modal dialogs created with "DialogBox" or "DialogBoxParam" are only modal relative to their parent window. The solution is to use 3ds max's main window as the parent window of any modal dialog created. Developers can get the proper window handle to use with **Interface::GetMAXHwnd()**. Once this is done, users can't hit "Undo" or any other operation from 3ds max's main user interface while a modal dialog is displayed.

The second condition, where modeless dialogs aren't disabled, occurs because the dialog is not registered with Windows. Developers using modeless dialogs must call **Interface::RegisterDlgWnd()** to prevent this problem.

**In Summary:** ALL modeless dialogs in 3ds max have to be registered with **Interface::RegisterDlgWnd()** and ALL modal dialogs in 3ds max have to be a direct child of 3ds max (developers must use **Interface::GetMAXHwnd()** when the dialog is created).

### Getting the Mesh from a TriObject

The method **TriObject::Mesh()** method has been renamed to **TriObject::GetMesh()**. Plug-ins using the old method name will have to be changed.

### **SaveRequired API Change**

Two existing functions have been renamed and one has been added. This change will cause many existing plug-ins to fail to build, however the fix is trivial. See below:

The existing function:

**void SetSaveRequired(int b=TRUE)**

has been renamed to:

**void SetSaveRequiredFlag(BOOL b=TRUE)**

The existing function:

**BOOL GetSaveRequired()**

has been renamed to:

**BOOL GetSaveRequiredFlag()**

There is also a new function:

**BOOL IsSaveRequired().**

The reason for these changes is that these two functions had misleading names. **GetSaveRequired()** does not tell you if saving is required, it just returns the value of the 'save required' bit. To really know if saving is required, you have to check the undo buffer as well as this flag. This is what the new function **IsSaveRequired()** is doing. If this function reports true, then you'll get the prompt at reset, file open, etc. The same goes for **SetSaveRequired()**, **SetSaveRequired(TRUE)** will cause the "save changes" prompt to appear, but **SetSaveRequired(FALSE)** will not prevent it from coming up unless you also reset the undo buffer.

### **Middle Mouse Button**

The middle mouse button may now be used to perform the arc-rotate function (by holding down the Alt-key). This new functionality necessitated an internal change where the viewport transformation might change inside a plug-ins mouse proc. To avoid problems plug-ins should not cache the viewport transformation at the start of the mouse proc as it may change due to the user performing an arc-rotate. Very few plug-ins do this currently, but those that do must not rely on this any longer.

## **Links to Topics By Plug-In Type**

This section provides links to topics on the required and optional changes for each plug-in type.

Note that all developers should be sure to look over the "Miscellaneous Changes in R3 That May Affect Plug-ins" section above since there may be important changes listed there as well.

[Required Changes To Atmospheric Plug-Ins](#)

[Required Changes To Bitmap Loader/Saver Plug-Ins](#)

[Required Changes To Controller Plug-Ins](#)

[Required Changes To File Export Plug-Ins](#)

[Required Changes To File Import Plug-Ins](#)

[Required Changes To Geometric Objects](#)

[Required Changes To Helper Objects](#)

[Required Changes To Light Plug-Ins](#)

[Required Changes To Materials Plug-Ins](#)

[Required Changes To NURBS Related Plug-Ins](#)

[Required Changes To Particles](#)

[Required Changes To Patch Related Plug-Ins](#)

[Required Changes To Procedural Shapes](#)

[Required Changes To Renderer Plug-Ins](#)

[Required Changes To Shape Objects](#)

[Required Changes To Snap Plug-Ins](#)

[Required Changes To Space Warps](#)

[Required Changes To Texture Plug-Ins](#)

# MAXScript SDK

See Also: [Must Read Sections for All Developers](#).

The MAXScript SDK is a set of Visual C++ headers and import libraries that C++ programmers can use to extend MAXScript. These extensions can be in the form of new built-in functions, new system globals or descriptors for new MAX plug-in class properties. This is useful for 3rd-party plug-in developers to write custom scripting interfaces for their plug-ins, and for programmers to do custom C++ performance code and drive it with scripts for hybrid tools.

The scripter SDK allows extensions to be added either incrementally through a MAXScript-specific DLL file type that is loaded by MAXScript or by runtime calls directly to MAXScript from within an existing plug-in.

The following topics describe the SDK:

[DLL Setup](#)

[Libraries](#)

[Build Configurations](#)

[MAXScript Value Constructors](#)

[Distinguished Values](#)

[Coercion to C++ Types](#)

[Protecting Newly Created Values from the Collector](#)

[Collector-Safe Value Local Macros](#)

[Value Local Arrays](#)

[Marking Values as Permanent or Collectable](#)

[Scripter-Callable Functions](#)

[Working with MAX Objects in the SDK](#)

[Constructing Wrappers](#)

[Retrieving Wrapped Objects](#)

[Handling Deleted Objects](#)

[MAX ClassIDs and Superclass IDs](#)

[The MAXClass Constructor](#)

[classOf\(\) and superClassOf\(\) Methods for MAX Objects](#)

[Collection Mapping](#)

[Array Access and Construction](#)

[Stream I/O](#)

[Access to the Compiler and Interpreter](#)

[Calling Scripted Functions](#)

[MAXScript Header Files](#)

[Defining New System Globals](#)

[Core Names](#)

## Class DataClassDesc

See Also: [Class ClassDesc](#), [Class ClassEntry](#), [Class BitmapIO](#), [Class SceneImport](#), [Class SceneExport](#).

```
class DataClassDesc : public ClassDesc
```

### Description:

This class is available in release 3.0 and later only.

This is a "partial" class descriptor for classes in plug-ins described by Registry entries. The registry is a system-defined database that applications and system components use to store and retrieve configuration data. This class provides a "stub" or partial class description, as read from the Registry, which provides basic descriptive information about the class (which proves adequate for certain purposes). Proper use of the method **ClassEntry::FullCD()** ensures that the partial class description will be replaced by the full one, which includes the **ClassDesc::Create()** method and other functionality. Note: With the addition of delay-loaded plugins, the method **ClassEntry::CD()** may return a pointer to an instance of this class. This derived class implements **Create()** by returning NULL. Developers need to call **FullCD()** in order to ensure that the class is actually there.

For additional details see [Class ClassEntry](#) and the Advanced Topics section [Deferred Loading of Plug-Ins](#).

### Data Members:

public:

These data members are initialized from the Registry.

**TSTR category;**

The category string (**ClassDesc::Category()**).

**DWORD classIDA;**

The first ulong of the Class\_ID (**ClassDesc::ClassID()**).

**DWORD classIDB;**

The second ulong of the Class\_ID (**ClassDesc::ClassID()**).

**DWORD superClassID;**

The SuperClassID (**ClassDesc::SuperClassID()**).

**TSTR className;**

The class name (**ClassDesc::ClassName()**).

**DWORD isPublic;**

The **ClassDesc::ISPublic()** return value.

**DWORD okToCreate;**

The **ClassDesc::OkToCreate()** return value.

**DWORD extCount;**

Returns the number of file name extensions supported

(**SceneImport::ExtCount()**,**SceneExport::ExtCount()**,

**BitmapIO::ExtCount()**).

**TSTR ext;**

The file name extension (**SceneImport::Ext(0)**, **SceneExport::Ext(0)**,

**BitmapIO::Ext(0)**).

**TSTR shortDesc;**

The short ASCII description (**SceneImport::ShortDesc()**,

**SceneExport::ShortDesc()**).

**TSTR longDesc;**

The long ASCII description (**SceneImport::LongDesc()**,

**SceneExport::LongDesc()**).

**DWORD supportsOptions;**

The export options (**SceneExport::SupportsOptions()**).

**DWORD capability;**

The BitmapIO module capability flags (**BitmapIO::Capability()**).

**DWORD inputTypeA;**

The first ulong of the **Modifier::InputType()** Class\_ID.

**DWORD inputTypeB;**

The second ulong of the **Modifier::InputType()** Class\_ID.

## **Methods:**

public:

The following are the meaningful methods from the base class ClassDesc which are provided by this class. Other methods not listed have basically NULL (i.e. non-functional) implementations.

## **Prototype:**

**int IsPublic();**

**Remarks:**

Returns the **isPublic** setting.

**Prototype:**

**const TCHAR \*ClassName();**

**Remarks:**

Returns the class name for the plug-in (**className**).

**Prototype:**

**SClass\_ID SuperClassID();**

**Remarks:**

Returns the SuperClassID (**superClassID**). Note: **typedef ulong SClass\_ID;**

**Prototype:**

**Class\_ID ClassID();**

**Remarks:**

Returns the Class\_ID. This is **Class\_ID(classIDA, classIDB)**.

**Prototype:**

**const TCHAR \*Category();**

**Remarks:**

Returns the **category** string.

**Prototype:**

**BOOL OkToCreate(Interface \*i);**

**Remarks:**

Returns the **okToCreate** state.

## Class ClassEntry

See Also: [Class DllDir](#), [Class SubClassList](#), [Class ClassDesc](#), [Class DataClassDesc](#), [Class Class\\_ID](#).

class ClassEntry

### Description:

This class is available in release 2.0 and later only.

This class provides information about a class in the **SubClassList** table of classes. This is information such as the class ID, category, public status, usage count, etc.

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**ClassDesc \*CD();**

#### Remarks:

Returns a pointer to the class descriptor for this entry. Note: With the addition of deferred-loaded plug-ins introduced in 3ds max 3.0, this method may return a pointer to an instance of class **DataClassDesc** (a sub-class of **ClassDesc**). A developer can tell if the returned class is a **DataClassDesc** object if **ClassEntry::IsLoaded()** returns FALSE. This derived class implements **ClassDesc::Create()** by returning NULL. Developers need to call **ClassEntry::FullCD()** in order to ensure that the class is actually present. Developers can call this method to retrieve data about the plug-in without forcing a load. It may also be called to get a 'full' **ClassDesc** if it is known that the plug-in is loaded (**IsLoaded()** returns TRUE).

#### Prototype:

**ClassDesc \*FullCD();**

#### Remarks:

This method is available in release 3.0 and later only.

This method fetches the class descriptor from a class entry. Unlike **CD()** above, this method will load the relevant DLL if its loading was deferred, and

will return a pointer to the loaded class descriptor. See [Deferred Loaded of Plug-Ins](#) for more details.

**Prototype:**

**int IsPublic();**

**Remarks:**

Returns nonzero if the class is public; otherwise zero. Non-public classes are those that are meant for private use by other plug-ins.

**Prototype:**

**Class\_ID ClassID();**

**Remarks:**

Returns the Class\_ID of the entry.

**Prototype:**

**TSTR &ClassName();**

**Remarks:**

Returns the class name of the entry.

**Prototype:**

**TSTR &Category();**

**Remarks:**

Returns a reference to the category string for this entry.

**Prototype:**

**int UseCount();**

**Remarks:**

Returns the number of instance of this class used in 3ds max.

**Prototype:**

**void IncUseCount();**

**Remarks:**

This is used internally to increment the usage count for this entry.

**Prototype:**

**void SetUseCount(int i);**

**Remarks:**

This is used internally to set the usage count for this entry.

**Prototype:**

**int IsAccType(int accType);**

**Remarks:**

Returns nonzero if this entry matches the specified access type; otherwise zero.

**Parameters:**

**int accType**

One of the following values:

**ACC\_PUBLIC** - public classes

**ACC\_PRIVATE** - non-public classes

**Prototype:**

**int DllNumber();**

**Remarks:**

Returns the index into the master DLL list of this entry.

**Prototype:**

**bool IsLoaded();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if loaded; otherwise FALSE.

**Prototype:**

**int ClassNumber();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the index of the class within the DLL

**Prototype:**

**void Set(ClassDesc \*cld, int dllN, int index);**

**Remarks:**

This method is used internally.

**Prototype:**

**int GetScroll();**

**Remarks:**

This method is used internally.

**Prototype:**

**void SetScroll(int s);**

**Remarks:**

This method is used internally.

**Prototype:**

**BOOL GetPageState(int i);**

**Remarks:**

This method is used internally.

**Prototype:**

**void SetPageState(int i,BOOL state);**

**Remarks:**

This method is used internally.

**Operators:**

**Prototype:**

**ClassEntry& operator=(const ClassEntry &ce);**

**Remarks:**

Assignment operator.

**Prototype:**

**int operator==(const ClassEntry &ce) const;**

**Remarks:**

Equality operator.

## Class NURBSTextureChannelSet

See Also: [Class NURBSTextureChannel](#), [NURBSSurface](#), [Template Class Tab](#).

```
class NURBSTextureChannelSet
```

### Description:

This class is available in release 3.0 and later only.

This class holds a table of pointers to all the **NURBSTextureChannel** data for a surface. There are methods to returns the table data by channel or by index and a method to add a new texture channel.

### Friend Classes:

```
friend class NURBSSurface;
```

### Data Members:

```
private:
```

```
    Tab<NURBSTextureChannel*> mTextureChannels;
```

A table of pointers to the texture channel data.

### Methods:

```
private:
```

### Prototype:

```
    NURBSTextureChannel* GetChannelByIndex(int index);
```

### Remarks:

Returns a pointer to the texture channel object whose index is specified.

### Parameters:

```
    int index
```

The zero based index into the table of texture channels in the set.

### Prototype:

```
    NURBSTextureChannel* GetChannel(int channel);
```

### Remarks:

Returns a pointer to the specific texture channel object. If not found a new channel is added with the specified index.

**Parameters:**

**int channel**

The channel to get. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

**NURBSTextureChannel\* AddChannel(int channel);**

**Remarks:**

Adds the specified channel and returns a pointer to the allocated texture object.

**Parameters:**

**int channel**

The texture channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

**int NumChannels();**

**Remarks:**

Returns the number of channels in the set.

## Class NURBSTextureChannel

See Also: [Class NURBSTextureSurface](#), [Class NURBSTextureChannelSet](#), [Class NURBSSurface](#), [Class Point2](#).

```
class NURBSTextureChannel
```

### Description:

This class is available in release 3.0 and later only.

This class holds the data associated with a single texture channel. This includes the **NURBSTextureSurface**, texture vertices, channel number, tiling, offset, and rotation settings. The tiling, offset and angle values are applied after the texture surface is applied. This gives the user more control of the mapping.

### Friend Classes:

```
friend class NURBSSurface;
```

```
friend class NURBSTextureChannelSet;
```

### Data Members:

```
private:
```

```
int mChannel;
```

The UV coordinates channel. This value can range from 0 to 98. A single surface can use up to 99 texture channels.

```
BOOL mGenUVs;
```

TRUE if the generate UV setting is on for this channel; otherwise FALSE.

```
Point2 mTexUVs[4];
```

The texture vertices (UV coordinates).

```
float mUTile;
```

The U tiling.

```
float mVTile;
```

The V tiling.

```
float mUOffset;
```

The U offset.

```
float mVOffset;
```

The V offset.

```
float mAngle;
```

The rotation angle in radians.

**NURBSTextureSurface mTexSurface;**

The texture surface associated this channel.

**Methods:**

private:

**Prototype:**

**NURBSTextureChannel(int channel);**

**Remarks:**

Constructor. The data members are initialized as follows:

```
mChannel = channel;  
mGenUVs = FALSE;  
mTexUVs[0] = Point2(0.0f, 0.0f);  
mTexUVs[1] = Point2(1.0f, 0.0f);  
mTexUVs[2] = Point2(0.0f, 1.0f);  
mTexUVs[3] = Point2(1.0f, 1.0f);  
mUTile = 1.0f;  
mVTile = 1.0f;  
mUOffset = 0.0f;  
mVOffset = 0.0f;  
mAngle = 0.0f;
```

**Prototype:**

**int GetChannel();**

**Remarks:**

Returns the UV coordinate channel.

**Prototype:**

**BOOL GenerateUVs();**

**Remarks:**

Returns TRUE if the generate UVs state is on; otherwise FALSE.

**Prototype:**

**void SetGenerateUVs(BOOL state);**

**Remarks:**

Sets the generate UV state.

**Parameters:**

**BOOL state**

TRUE for on; FALSE for off.

**Prototype:**

**Point2 GetTextureUVs(TimeValue t, int i);**

**Remarks:**

Returns the 'i-th' texture corner at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the texture vertex.

**int i**

The index of the texture vertex. This is a value in the range 0 to 3.

**Prototype:**

**void SetTextureUVs(TimeValue t, int i, Point2 pt);**

**Remarks:**

Sets the specified texture coordinate.

**Parameters:**

**TimeValue t**

The time at which to set the texture UV.

**int i**

The index of the UV to set. This is a value in the range 0 to 3.

**Point2 pt**

The UV point to set.

**Prototype:**

**void GetTileOffset(TimeValue t, float &ut, float &vt, float &uo,**

**float &vo, float &a);**

**Remarks:**

Retrieves the tiling, offset and angle values at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the values.

**float &ut**

The U tiling.

**float &vt**

The V tiling.

**float &uo**

The U offset.

**float &vo**

The V offset.

**float &a**

The angle.

**Prototype:**

**void SetTileOffset(TimeValue t, float ut, float vt, float uo, float vo, float a);**

**Remarks:**

Sets the tiling, offset and angle values at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the values.

**float ut**

The U tiling.

**float vt**

The V tiling.

**float uo**

The U offset.

**float vo**

The V offset.

**float a**

The angle.

**Prototype:**

**NURBSTextureSurface& GetTextureSurface();**

**Remarks:**

Returns a reference to the **NURBSTextureSurface** maintained by the class.

**Prototype:**

**void SetTextureSurface(NURBSTextureSurface& texSurf);**

**Remarks:**

Set the **NURBSTextureSurface** maintained by the class.

**Parameters:**

**NURBSTextureSurface& texSurf**

The texture surface to set.

## Class NURBSTexturePoint

See Also: [Class NURBSTextureSurface](#), [Class NURBSObject](#), [Class Point2](#).

```
class NURBSTexturePoint : public NURBSObject
```

### Description:

This class is available in release 3.0 and later only.

This class holds a single texture vertex in a NURBS texture surface. Methods are available to get and set the point in various formats.

### Data Members:

protected:

**double mX, mY;**

The position of the point.

### Methods:

public:

### Prototype:

**NURBSTexturePoint();**

### Remarks:

Constructor. The data members (from **NURBSObject**) are initialized as follows:

**mKind = kNURBSTexturePoint;**

**mType = kNTexturePoint;**

### Prototype:

**virtual Point2 GetPosition(TimeValue t);**

### Remarks:

Returns the position of the texture point at the specified time (as a Point2).

### Parameters:

**TimeValue t**

The time at which to get the position.

### Prototype:

**virtual void GetPosition(TimeValue t, float& x, float& y);**

**Remarks:**

Retrieves the position of the texture point at the specified time (as floats).

**Parameters:**

**TimeValue t**

The time at which to get the position.

**float& x**

The x position is returned here.

**float& y**

The y position is returned here.

**Prototype:**

**virtual void GetPosition(TimeValue t, double& x, double& y);**

**Remarks:**

Retrieves the position of the texture point at the specified time (as doubles).

**Parameters:**

**TimeValue t**

The time at which to get the position.

**double& x**

The x position is returned here.

**double& y**

The y position is returned here.

**Prototype:**

**void SetPosition(TimeValue t, Point2 pt);**

**Remarks:**

Sets the position of the point at the specified time (using a Point2).

**Parameters:**

**TimeValue t**

The time at which to set the position.

**Point2 pt**

The position to set.

**Prototype:**

**void SetPosition(TimeValue t, float x, float y);**

**Remarks:**

Sets the position of the point at the specified time (using floats).

**Parameters:**

**TimeValue t**

The time at which to set the position.

**float x**

The x coordinate to set.

**float y**

The y coordinate to set.

**Prototype:**

**void SetPosition(TimeValue t, double x, double y);**

**Remarks:**

Sets the position of the point at the specified time (using doubles).

**Parameters:**

**TimeValue t**

The time at which to set the position.

**double x**

The x coordinate to set.

**double y**

The y coordinate to set.

## Class NURBSSurfaceEdgeCurve

See Also: [Class NURBSCurve](#).

```
class NURBSSurfaceEdgeCurve : public NURBSCurve
```

### Description:

This class is available in release 3.0 and later only.

This class defines a dependent edge curve. There are methods available to get/set the parent surface index and id, and get/set the parameter which determines the location on the surface the curve matches.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSSurfaceEdgeCurve();
```

### Remarks:

Constructr. The data members are initialized as follows:

```
mType = kNSurfaceEdgeCurve;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mSeed = Point2(0.0, 0.0);
```

**Prototype:**

```
virtual ~NURBSSurfaceEdgeCurve();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the **NURBSSet** of the parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Prototype:**

**Point2 GetSeed();**

**Remarks:**

Returns the UV location of the seed value on the curve.

**Prototype:**

**void SetSeed(Point2& seed);**

**Remarks:**

Sets the UV location of the seed value on the curve.

**Parameters:**

**Point2& seed**

The seed value to set.

**Operators:**

**Prototype:**

**NURBSSurfaceEdgeCurve &operator=(const  
NURBSSurfaceEdgeCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSSurfaceEdgeCurve& curve**

The curve to assign from.

## Class NURBSFilletSurface

See Also: [Class NURBSSurface](#).

```
class NURBSFilletSurface : public NURBSSurface
```

### Description:

This class is available in release 3.0 and later only.

This class provides access to the NURBS Fillet Surface. Methods are provided to get / set the parent surfaces, get / set the radius, and get / set the trim and trim flip settings.

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSFilletSurface();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNFilletSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
```

```
mParentId[0] = mParentId[1] = 0;
```

```
mParentIndex[0] = mParentIndex[1] = -1;
```

```
mRadius[0] = mRadius[1] = 10.0f;  
mSeed[0] = mSeed[1] = Point2(0.0, 0.0);  
mCubic = TRUE;
```

**Prototype:**

```
virtual ~NURBSFilletSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the specified parent surface (by **NURBSSet** index) for the fillet.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**int index**

The index in the **NURBSSet**.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the specified parent surface (by **NURBSId**) for the fillet.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**NURBSId id**

The ID of the surface.

**Prototype:**

```
int GetParent(int pnum);
```

**Remarks:**

Returns the index in the **NURBSSet** of the specified parent surface.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the NURBSId of the specified parent surface.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**Prototype:**

**Point2 GetSeed(int pnum);**

**Remarks:**

Returns the UV location of the seed value on the specified surface.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**Prototype:**

**void SetSeed(int pnum, Point2& seed);**

**Remarks:**

Sets the UV location of the seed value on the specified surface.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**Point2& seed**

The seed value to set.

**Prototype:**

**BOOL IsCubic();**

**Remarks:**

Returns TRUE if the cubic setting is on; FALSE if off. When off, the radius is always linear. When on, the radius is treated as a cubic function, allowing it to change based on the parent surface's geometry.

**Prototype:**

**void SetCubic(BOOL cubic);**

**Remarks:**

Sets the cubic setting to on or off.

**Parameters:**

**BOOL cubic**

TRUE for on; FALSE for off.

**Prototype:**

**float GetRadius(TimeValue t, int rnum);**

**Remarks:**

Returns the specified radius at the time passed.

**Parameters:**

**TimeValue t**

The time at which to get the radius.

**int rnum**

Specifies which radius to get: 0 for start, 1 for end.

**Prototype:**

**void SetRadius(TimeValue t, int rnum, float radius);**

**Remarks:**

Sets the specified radius at the time passed.

**Parameters:**

**TimeValue t**

The time at which to set the radius.

**int rnum**

Specifies which radius to set: 0 for start, 1 for end.

**float radius**

The radius to set.

**Prototype:**

**BOOL GetTrimSurface(int pnum);**

**Remarks:**

Returns the trim surface setting for the specified parent surface. TRUE if on (trims the parent surface at the edge of the fillet); FALSE if off.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**Prototype:**

**void SetTrimSurface(int pnum, BOOL trim);**

**Remarks:**

Sets the trim surface setting for the specified parent surface.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**BOOL trim**

TRUE for on (trims the parent surface at the edge of the fillet); FALSE for off.

**Prototype:**

**BOOL GetFlipTrim(int pnum);**

**Remarks:**

Returns the state of the trim flip setting. When set this reverses the direction of the trim.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**Prototype:**

**void SetFlipTrim(int pnum, BOOL flip);**

**Remarks:**

Sets the state of the trim flip setting. When set this reverses the direction of the trim.

**Parameters:**

**int pnum**

The parent surface: 0 or 1.

**BOOL flip**

TRUE for on; FALSE for off.

**Prototype:**

**NURBSFilletSurface & operator=(const NURBSFilletSurface& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSFilletSurface& curve**

The surface to assign.

# Class NURBSProceduralCurve

See Also: [Class NURBSCVCurve](#), [Class NURBSCurve](#).

class NURBSProceduralCurve

## Description:

This class is available in release 3.0 and later only.

This is the base class for procedurally defined curves. Note that this is not subclassed from **NURBSObject**. You must use the **GenNURBSCVCurveProcedurally()** function.

The following typedef-ed functions are used by the **GenNURBSCVCurveProcedurally()** function.

## Prototype:

```
typedef NURBSResult (*CurveParamRangeProc)(double& tMin,  
double& tMax);
```

## Remarks:

The curve parameter range procedure. This retrieves the minimum and maximum valid values for u as passed to the **CurveEvalProc()**.

## Parameters:

**double& tMin**

The minimum value.

**double& tMax**

The maximum value.

## Return Value:

See [List of NURBS Results](#) .

## Prototype:

```
typedef NURBSResult (*CurveEvalProc)(double u, Point3& pt);
```

## Remarks:

The curve evaluation procedure. This retrieves the point on the curve based on the u parameter.

## Parameters:

**double u**

Specifies the U point along the curve to evaluate.

**Point3& pt**

The output point on the curve at U.

**Return Value:**

See [List of NURBS Results](#) .

**Prototype:**

```
typedef NURBSResult (*CurveEvalTan)(double u, Point3& pt,  
Point3& tan);
```

**Remarks:**

The curve point and tangent evaluation procedure. This retrieves the point and tangent at the point on the curve based on the U parameter.

**Parameters:**

**double u**

Specifies the point along the curve to evaluate.

**Point3& pt**

The output point at U.

**Point3& tan**

The output tangent at U.

**Return Value:**

See [List of NURBS Results](#) .

**Prototype:**

```
typedef NURBSResult (*CurveArcLengthProc)(double&  
arcLength);
```

**Remarks:**

Retrieves the length of the curve.

**Parameters:**

**double& arcLength**

The output arc length.

**Return Value:**

See [List of NURBS Results](#) .

## Data Members:

public:

**CurveParamRangeProc mParamProc;**

The curve parameter range procedure. Note: This procedure must be implemented.

**CurveEvalProc mEvalProc;**

The curve evaluate procedure. Note: This procedure must be implemented.

**CurveEvalTan mEvalTanProc;**

The curve evaluate tangent procedure. Note: This procedure is optional.

**CurveArcLengthProc mArcLengthProc;**

The curve arc length procedure. Note: This procedure is optional.

## Methods:

public:

### Prototype:

**NURBSProceduralCurve(CurveParamRangeProc param,  
CurveEvalProc eval, CurveEvalTan tan, CurveArcLengthProc  
arclen);**

### Remarks:

Constructor. The data members are initialized to the values passed.

**The following global function is not part of this class but is available for use:**

### Function:

**NURBSResult  
GenNURBSCVCurveProcedurally(NURBSProceduralCurve  
\*pCrv, double tolerance, NURBSCVCurve& crv);**

### Remarks:

This global function is available in release 3.0 and later only.

To use this API you need to create an instance of

**NURBSProceduralCurve** setting at least the ParameterRange and Eval procedures. The others can be NULL unless you have fast versions of them.

The call to this function then fills in a **NURBSCVCurve** with the curve that is defined by the procedures and the tolerance.

**Parameters:**

**\*pCrv**

Points to the **NURBSProceduralCurve** object used to generate the curve procedurally.

**double tolerance**

The tolerance is the allowable deviation of the approximating NURBS surface to the surface defined by the procs.

**NURBSCVCurve& crv**

The generated curve is returned here.

**Return Value:**

See [List of NURBS Results](#) .

## Class NURBSProceduralSurface

See Also: [Class NURBSCVSurface](#), [Class NURBSSurface](#).

class NURBSProceduralSurface

### Description:

This class is available in release 3.0 and later only.

This is the base class for a procedurally defined surface. Note: This class is not subclassed from **NURBSObject**. You must use the **GenNURBSCVSurfaceProcedurally()** function.

The following typedef-ed functions are used by the **GenNURBSCVSurfaceProcedurally()** function.

### Prototype:

```
typedef NURBSResult (*SurfParamRangeProc)(double& uMin,  
double& uMax, double& vMin, double& vMax);
```

### Remarks:

The surface parameter range procedure. This retrieves the minimum and maximum valid values for u and v as passed to the **SurfEvalProc()**.

### Parameters:

**double& uMin**

The min U value.

**double& uMax**

The max U value.

**double& vMin**

The min V value.

**double& vMax**

The max V value.

### Return Value:

See [List of NURBS Results](#) .

### Prototype:

```
typedef NURBSResult (*SurfEvalProc)(double u, double v,  
Point3& pt);
```

**Remarks:**

This evaluates the surface for the point or position at a given U and V parameter.

**Parameters:****double u**

Specifies the U point along the surface to evaluate. This value must be between the **uMin** and **uMax** as returned from **SurfParamRangeProc()**.

**double v**

Specifies the V point along the surface to evaluate. This value must be between the **vMin** and **vMax** as returned from **SurfParamRangeProc()**.

**Point3& pt**

The output point on the surface at (u,v).

**Return Value:**

See [List of NURBS Results](#) .

**Prototype:**

```
typedef NURBSResult (*SurfEvalTan)(double u, double v,  
Point3& uTan, Point3& vTan);
```

**Remarks:**

This evaluates the surface for the partial derivative with respect to U and the partial derivative with respect to V at a given U and V parameter. That partial derivative with respect to U is a tangent vector in the U direction and the partial derivative with respect to V is a tangent vector in the V direction.

**Parameters:****double u**

Specifies the U point along the curve to evaluate. This value must be between the **uMin** and **uMax** as returned from **SurfParamRangeProc()**.

**double v**

Specifies the V point along the surface to evaluate. This value must be between the **vMin** and **vMax** as returned from **SurfParamRangeProc()**.

**Point3& uTan**

The tangent along u.

**Point3& vTan**

The tangent along v.

**Return Value:**

See [List of NURBS Results](#) .

**Prototype:**

```
typedef NURBSResult (*SurfEvalMixedProc)(double u, double v,  
Point3& mixed);
```

**Remarks:**

This evaluates the surface for the mixed partial derivative with respect to U and V at a given U and V parameter.

**Parameters:**

**double u**

Specifies the U point along the curve to evaluate. This value must be between the **uMin** and **uMax** as returned from **SurfParamRangeProc()**.

**double v**

Specifies the V point along the surface to evaluate. This value must be between the **vMin** and **vMax** as returned from **SurfParamRangeProc()**.

**Point3& mixed**

This is the resulting mixed partial derivative vector that has been evaluated.

**Return Value:**

See [List of NURBS Results](#) .

**Data Members:**

public:

**SurfParamRangeProc mParamProc;**

The surface parameter range procedure. Note: This procedure must be implemented.

**SurfEvalProc mEvalProc;**

The surface evaluation procedure. Note: This procedure must be implemented.

**SurfEvalTan mEvalTanProc;**

The surface tangent evaluation procedure. Note: This procedure is optional.

**SurfEvalMixedProc mEvalMixedProc;**

The mixed partial derivative procedure. Note: This procedure is optional.

**Methods:**

public:

**Prototype:**

**NURBSProceduralSurface(SurfParamRangeProc param,  
SurfEvalProc eval, SurfEvalTan tan, SurfEvalMixedProc mixed);**

**Remarks:**

Constructor. The data members are initialized to the values passed.

Both (**\*SurfEvalTan**) and (**\*SurfEvalMixedProc**) are optional. If NULLs are provided for either of these functions then 3ds max will resort to a finite differences based solution.

**Parameters:**

**The following global function is not part of this class but is available for use:**

**Function:**

**NURBSResult  
GenNURBSCVSurfaceProcedurally(NURBSProceduralSurface  
\*pSurf, double tolerance, NURBSCVSurface& surf);**

**Remarks:**

This global function is available in release 3.0 and later only.

To use this function you need to create an instance of

**NURBSProceduralSurface** setting at least the ParameterRange and Eval procedures. The others can be NULL unless you have fast versions of them.

This call then fills in a NURBSCVSurface with the surface that is defined by the procedures and the tolerance.

**Parameters:**

**NURBSProceduralSurface \*pSurf**

Points to the **NURBSProceduralSurface** to generate CV surfaces from.

**double tolerance**

The tolerance is the allowable deviation of the approximating NURBS surface to the surface defined by the procs.

**NURBSCVSurface& surf**

The generated NURBSCVSurface is returned here.

**Return Value:**

See [List of NURBS Results](#) .

## List of Notification Codes

See Also: [Structure NotifyInfo](#), [Class Interface](#), [Class Bitmap](#), [Class RenderGlobalContext](#).

The following pre-defined system notification codes may be passed to the global functions **RegisterNotification()**, **UnRegisterNotification()**, **BroadcastNotification()**.

### **NOTIFY\_UNITS\_CHANGE**

Sent if the user changes the unit setting

### **NOTIFY\_TIMEUNITS\_CHANGE**

Sent if the user changes the time format setting

### **NOTIFY\_VIEWPORT\_CHANGE**

Sent if the user changes the viewport layout

### **NOTIFY\_SPACEMODE\_CHANGE**

Sent if the user changes the reference coordinate system.

### **NOTIFY\_SYSTEM\_PRE\_RESET**

Sent before 3ds max is reset. This is available in release 2.0 and later only.

### **NOTIFY\_SYSTEM\_POST\_RESET**

Sent after 3ds max is reset. This is available in release 2.0 and later only.

### **NOTIFY\_SYSTEM\_PRE\_NEW**

Sent before 3ds max is NEW'd-out. This is available in release 2.0 and later only.

### **NOTIFY\_SYSTEM\_POST\_NEW**

Sent after 3ds max is NEW'd-out. This is available in release 2.0 and later only.

### **NOTIFY\_FILE\_PRE\_OPEN**

Sent before a new file is opened. This is available in release 2.0 and later only.

### **NOTIFY\_FILE\_POST\_OPEN**

Sent after a new file is opened successfully. This is available in release 2.0 and later only.

### **NOTIFY\_FILE\_PRE\_MERGE**

Sent before a file is merged. This is available in release 2.0 and later only.

Note: When merge is called to load an XRef, developers can be aware of this by testing **void \* callParam** to see if it's non-null. If it is then an XRef is

being loaded.

### **NOTIFY\_FILE\_POST\_MERGE**

Sent after a file is merged. This is available in release 2.0 and later only.

Note: When merge is called to load an XRef, developers can be aware of this by testing **void \* callParam** to see if it's non-null. If it is then an XRef is being loaded.

### **NOTIFY\_FILE\_PRE\_SAVE**

Sent before a file is saved. This is available in release 2.5 and later only.

### **NOTIFY\_FILE\_POST\_SAVE**

Sent after a file is saved. This is available in release 2.5 and later only.

### **NOTIFY\_FILE\_PRE\_SAVE\_OLD**

This option is reserved for future use as of 3ds max 3.0.

### **NOTIFY\_FILE\_POST\_SAVE\_OLD**

This option is reserved for future use as of 3ds max 3.0.

### **NOTIFY\_SELECTIONSET\_CHANGED**

This option is available in release 3.0 and later only.

Sent after the selection set has changed.

### **NOTIFY\_BITMAP\_CHANGED**

This option is available in release 3.0 and later only.

Sent after the bitmap is reloaded (the **NotifyInfo** structure pointer **callParam** is passed the **TCHAR \*** to the bitmap file name). This is used for updating bitmaps that have changed. The **callParam** is used to pass the name of the bitmap file in case it is used in multiple changes. If the **callParam** is NULL, this notification will apply to all bitmaps (as is the case when the input file gamma changes).

### **NOTIFY\_PRE\_RENDER**

This option is available in release 3.0 and later only.

Sent before rendering is started.

### **NOTIFY\_POST\_RENDER**

This option is available in release 3.0 and later only.

Sent after rendering has finished.

### **NOTIFY\_PRE\_RENDERFRAME**

This option is available in release 3.0 and later only.

Sent before rendering each frame (the `NotifyInfo` structure pointer **callParam** is passed a pointer to the **RenderGlobalContext**).

Note: At the time of this call the scene **cannot** be modified. The renderer has already called **GetRenderMesh()** on all the object instances, and the materials and lights are already updated. If you don't modify anything that is rendered, then it is okay to use this callback.

### **NOTIFY\_POST\_RENDERFRAME**

This option is available in release 3.0 and later only.

Sent after rendering each frame (the `NotifyInfo` structure pointer **callParam** is passed a pointer to the **RenderGlobalContext**). See the note in **NOTIFY\_PRE\_RENDERFRAME** above.

### **NOTIFY\_PRE\_IMPORT**

This option is available in release 3.0 and later only.

Sent before a file is imported (always).

### **NOTIFY\_POST\_IMPORT**

This option is available in release 3.0 and later only.

Sent after a file is imported successfully.

### **NOTIFY\_IMPORT\_FAILED**

This option is available in release 3.0 and later only.

Sent if an import fails or is otherwise cancelled.

### **NOTIFY\_PRE\_EXPORT**

This option is available in release 3.0 and later only.

Sent before a file is exported (always).

### **NOTIFY\_POST\_EXPORT**

This option is available in release 3.0 and later only.

Sent after a file is exported successfully.

### **NOTIFY\_EXPORT\_FAILED**

This option is available in release 3.0 and later only.

Sent if an export fails or is otherwise cancelled.

### **NOTIFY\_NODE\_RENAMED**

This option is available in release 3.0 and later only.

Sent if a node is renamed. (the `NotifyInfo` structure pointer **callParam** is passed a pointer to a `struct{ TCHAR* oldname; TCHAR* newname;`

} See **\MAXSDK\SAMPLES\OBJECTS\LIGHT.CPP** for an example of this notification in use.

### **NOTIFY\_PRE\_PROGRESS**

This option is available in release 3.0 and later only.

Sent before the progress bar is displayed. The progress bar is displayed, for example, when the Render Preview command is run. Note: If a plug-in uses a modeless window it should hide the window between this event and

### **NOTIFY\_POST\_PROGRESS.**

### **NOTIFY\_POST\_PROGRESS**

This option is available in release 3.0 and later only.

Sent after the progress bar is finished.

### **NOTIFY\_MATLIB\_PRE\_OPEN**

This option is available in release 3.0 and later only.

Sent before loading a material library.

### **NOTIFY\_MATLIB\_POST\_OPEN**

This option is available in release 3.0 and later only.

Sent after loading a material library. The **callParam** is a pointer to the MtlBaseLib if successful, otherwise it's NULL.

### **NOTIFY\_MATLIB\_PRE\_SAVE**

This option is available in release 3.0 and later only.

Sent before saving a material library.

### **NOTIFY\_MATLIB\_POST\_SAVE**

This option is available in release 3.0 and later only.

Sent after saving a material library.

### **NOTIFY\_MATLIB\_PRE\_MERGE**

This option is available in release 3.0 and later only.

Sent before merging a material library.

### **NOTIFY\_MATLIB\_POST\_MERGE**

This option is available in release 3.0 and later only.

Sent after merging a material library.

### **NOTIFY\_MODPANEL\_SEL\_CHANGED**

This option is available in release 3.0 and later only.

Sent when the Modify Panel focuses on a new object (via opening the Modify

Panel or changing the selection)

### **NOTIFY\_RENDER\_PREEVAL**

This option is available in release 4.0 and later only.

Sent before the render start evaluating objects

### **NOTIFY\_NODE\_CREATED**

This option is available in release 4.0 and later only.

Sent when a node is created (callParam is pointer to node)

### **NOTIFY\_NODE\_LINKED**

This option is available in release 4.0 and later only.

Sent when a node is linked (callParam is pointer to node)

### **NOTIFY\_NODE\_UNLINKED**

This option is available in release 4.0 and later only.

Sent when a node is unlinked (callParam is pointer to node)

### **NOTIFY\_NODE\_HIDE**

This option is available in release 4.0 and later only.

Sent when a node is hidden (callParam is pointer to node)

### **NOTIFY\_NODE\_UNHIDE**

This option is available in release 4.0 and later only.

Sent when a node is unhidden (callParam is pointer to node)

### **NOTIFY\_NODE\_FREEZE**

This option is available in release 4.0 and later only.

Sent when a node is frozen (callParam is pointer to node)

### **NOTIFY\_NODE\_UNFREEZE**

This option is available in release 4.0 and later only.

Sent when a node is unfrozen (callParam is pointer to node)

### **NOTIFY\_NODE\_PRE\_MTL**

This option is available in release 4.0 and later only.

Node is about to get a new material (callParam is pointer to node)

### **NOTIFY\_NODE\_POST\_MTL**

This option is available in release 4.0 and later only.

Node just got a new material (callParam is pointer to node)

### **NOTIFY\_WM\_ENABLE**

This option is available in release 4.0 and later only.

Sent when the main window gets an WM\_ENABLE (BOOL enabled)

### **NOTIFY\_SYSTEM\_SHUTDOWN**

This option is available in release 4.0 and later only.

Sent when 3ds max is about to exit

### **NOTIFY\_SYSTEM\_SHUTDOWN2**

This option is available in release 4.0 and later only.

Sent after the scene is destroyed. Most plug-ins will most likely not live long enough to receive the notification. It is important to unregister this notification when your plug-in dies. If not, 3ds max will try to notify objects that don't exist anymore.

### **NOTIFY\_SYSTEM\_STARTUP**

This option is available in release 4.0 and later only.

Sent when 3ds max starts up.

### **NOTIFY\_PLUGIN\_LOADED**

This option is available in release 4.0 and later only.

Sent when a plug-in was just loaded. (callParam is pointer to DllDesc)

### **NOTIFY\_SEL\_NODES\_PRE\_DELETE**

This option is available in release 4.0 and later only.

Sent when selected nodes will be deleted. (callParam is pointer to Tab<INode\*>)

### **NOTIFY\_SEL\_NODES\_POST\_DELETE**

This option is available in release 4.0 and later only.

Sent when selected nodes have just been deleted.

### **NOTIFY\_SYSTEM\_SHUTDOWN2**

This option is available in release 4.0 and later only.

Sent when the 3ds max system shutdown is completed. This is the last broadcast sent before exit.

### **NOTIFY\_ANIMATE\_ON**

This option is available in release 4.0 and later only.

Sent when the Animate user interface mode is activated.

### **NOTIFY\_ANIMATE\_OFF**

This option is available in release 4.0 and later only.

Sent when the Animate user interface mode is de-activated.

## **NOTIFY\_COLOR\_CHANGE**

This option is available in release 4.0 and later only.

Sent when the system is updating its custom colors.

If a plug-in has created a toolbar with any **MaxBmpFileIcons** on it, it should register this callback for color changing, and call **ICustToolbar::ResetIconImages()** on the toolbar. See [Class ICustToolbar](#), and [Class MAXBmpFileIcon](#).

## **REFMSG\_NODE\_MATERIAL\_CHANGED**

This option is available in release 4.0 and later only.

This notification indicates that the material changed.

## **REFMSG\_NODE\_WSCACHE\_UPDATED**

This option is available in release 4.0 and later only.

Sent whenever the mod stack gets re-evaluated.

## **NOTIFY\_PRE\_EDIT\_OBJ\_CHANGE**

This option is available in release 4.0 and later only.

Sent just before the current edit object is about to change. This notifications are sent whenever the object returned by **Interface::GetCurEditObject()** changes.

## **NOTIFY\_POST\_EDIT\_OBJ\_CHANGE**

This option is available in release 4.0 and later only.

Sent just after the current edit object changes. This notifications are sent whenever the object returned by **Interface::GetCurEditObject()** changes.

## **NOTIFY\_BEGIN\_RENDERING\_REFLECT\_REFRACT\_MAP**

This option is available in release 5.0 and later only.

Sent when the render begins to render the Reflect/Refract map

## **NOTIFY\_BEGIN\_RENDERING\_ACTUAL\_FRAME**

This option is available in release 5.0 and later only.

Sent when the render begins to render the full frame

## **NOTIFY\_BEGIN\_RENDERING\_TONEMAPPING\_IMAGE**

This option is available in release 5.0 and later only.

Sent when the render begins to render the Tone map

## **NOTIFY\_RADIOSTYPROCESS\_STARTED**

This option is available in release 5.0 and later only.

Sent when radiosity processing is started

**NOTIFY\_RADIOSITYPROCESS\_STOPPED**

This option is available in release 5.0 and later only.

Sent when radiosity processing is stopped, but not done

**NOTIFY\_RADIOSITYPROCESS\_RESET**

This option is available in release 5.0 and later only.

Sent when radiosity processing is reset

**NOTIFY\_RADIOSITYPROCESS\_DONE**

This option is available in release 5.0 and later only.

Sent when radiosity processing is done

**NOTIFY\_RADIOSITY\_PLUGIN\_CHANGED**

This option is available in release 5.0 and later only.

Sent when radiosity plugin is changed.

**NOTIFY\_LIGHTING\_UNIT\_DISPLAY\_SYSTEM\_CHANGE**

This option is available in release 5.0 and later only.

Sent when the lighting unit display system is changed

**NOTIFY\_TIMERANGE\_CHANGE**

This option is available in release 5.1 and later only.

Sent after the animate time range has been changed

# Class ViewWindow

See Also: [Class Interface](#).

```
class ViewWindow : public InterfaceServer
```

## Description:

This class is available in release 3.0 and later only.

This is the base class for the creation of non-3D windows that appear in a 3ds max viewport. These views are called "Extended Viewports". In order for a window to appear inside a viewport, you need to derive a class from this class. An instance of the derived class must be registered via the **RegisterViewWindow()** call in the **Interface** class. A given **ViewWindow** derivative should only be registered once.

When developers have registered their window types, the list of available extended views will appear in the view selection pop-up (either in the right-click viewport menu or the Viewport Configuration dialog) as a submenu of the "Extended" view label.

There are two items which should be made in the extended viewport dialog procedure:

**Interface::MakeExtendedViewportActive()** should be called whenever the user clicks in the non-3D window (so as to deactivate the current 3D window, and redirect commands like the Min/Max toggle to the non-3D viewport window).

**Interface::PutUpViewMenu()** should be called when the user right-clicks in a dead region of the non-3D window. This brings up the view selection menu so that the user can choose to replace the current window with a 3D or other non-3D window without having to go to the Views | Viewport Config dialog directly.

All methods of this class are implemented by the plug-in.

## Sample Code:

```
class TestViewWindow : public ViewWindow {
public:
    TCHAR *GetName() { return _T("TestViewWindow"); }
    HWND CreateViewWindow(HWND hParent, int x, int y, int w,
```

```
int h);  
void DestroyViewWindow(HWND hWnd);  
};
```

```
HWND TestViewWindow::CreateViewWindow(HWND hParent,  
int x, int y, int w, int h) {  
    return CreateWindow("button", "Test Button", WS_VISIBLE  
    | WS_CHILD, x, y, w, h, hParent, NULL,  
    (HINSTANCE)GetWindowLong(hParent,  
    GWL_HINSTANCE), NULL);  
}
```

```
void TestViewWindow::DestroyViewWindow(HWND hWnd) {  
    DestroyWindow(hWnd);  
}  
static TestViewWindow tww;
```

**Methods:**

public:

**Prototype:**

```
virtual TCHAR *GetName()=0;
```

**Remarks:**

Returns the name of the window type. For example, "Asset Manager".

**Prototype:**

```
virtual HWND CreateViewWindow(HWND hParent, int x, int y,  
int w, int h)=0;
```

**Remarks:**

Creates and returns a handle to a new extended view window.

**Parameters:**

**HWND hParent**

The handle of the parent window.

**int x**

The x coordinate of the window's upper left corner.

**int y**

The y coordinate of the window's upper left corner.

**int w**

The window width.

**int h**

The window height.

**Prototype:**

**virtual void DestroyViewWindow(HWND hWnd)=0;**

**Remarks:**

Destroys the previously created window as specified by the handle.

**Parameters:**

**HWND hWnd**

The handle of the window to destroy.

**Prototype:**

**virtual BOOL CanCreate();**

**Remarks:**

Returns TRUE if the ViewWindow can be created; otherwise FALSE. This method can be overridden to return FALSE if a ViewWindow can only have a single instance, and that instance is already present. If this method returns FALSE, then the menu item for this ViewWindow will be grayed out.

**Default Implementation:**

**{ return TRUE; }**

**Prototype:**

**virtual int NumberCanCreate();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the number of instances of a given window that can be

created. This allows, for example, the UI to know without physically creating a window that only a limited number of windows of that type can be created. A -1 implies that the count is not known.

**Default Implementation:**

```
{ return -1; }
```

# Keyboard Shortcut System

[Class ShortcutTable](#), [Class ShortcutOperation](#), [Class ShortcutCallback](#)

**Important Note:** A new system to handle keyboard accelerators was added for R4. This system supercedes those used in previous release. For information on this system see the [class ActionTable](#). The information shown below applies to previous version of the SDK APIs.

## Overview

The following is a discussion of the plug-in definable keyboard shortcut system available in release 3.0 and later. The pre-R3 system remains in place for backwards compatibility.

The keyboard preferences dialog has been expanded to allow the editing of keyboard shortcuts for plug-ins. There is a new radio button added to the set of shortcuts, labeled "Plug-Ins". When this is selected a drop-down list next to the button is activated which lists all the plug-ins that have registered keyboard shortcut tables. After selecting one from the drop-down list, you can change the shortcuts in the same way the main UI shortcuts are changed. This provides a clean way for all plug-ins to make their keyboard shortcuts available in a uniform manner.

The basic process works like this: A developer assigns an ID and name to each shortcut (these are stored in a **Shortcut Description**). The ID identifies each shortcut and the name appears in the 3ds max user interface where the shortcuts are assigned to keys by the user. An array of these shortcuts is used to make up something called a **Shortcut Table**. This table stores and provides access to the shortcut data. These data items are referred to as **Shortcut Operations**.

A developer provides 3ds max with the shortcut table data for a plug-in at Dll load time using two new methods of the Class Descriptor. These methods return the number of shortcut tables used by the plug-in and return a pointer to the 'i-th' table. These methods are **ClassDesc::NumShortcutTables()** and **ClassDesc::GetShortcutTable(int i)**.

The tables become active and inactive as a whole by using two methods provided by the **Interface** class. These are **ActivateShortcutTable()** and **DeactivateShortcutTable()**. These will normally be called by a developer inside **BeginEditParams()** and **EndEditParams()**.

## Principal Classes

The following is an overview of the principal classes and structures involved in making this happen:

[Class ShortcutTable](#): This class provides storage for and access to the **ShortcutOperations**.

[Class ShortcutOperation](#): This class describes an operation that can be attached to a keyboard shortcut. There is an ID sent to the window proc when the shortcut is executed, a name for the operation that appears in the 3ds max user interface where the shortcuts are assigned.

[Class ShortcutCallback](#): When the user presses a shortcut key that is assigned in an active table, it calls this class's **KeyboardShortcut(int id)** method with the id of the operation. Here the plug-in should execute the operation. If the state of the plug-in is such that the operation should not be performed it can return FALSE from this method. In this case any system defined shortcut will be executed instead. If the plug-in returns TRUE then only the plug-in defined shortcut is executed. A developer derives a class from this class and passes an instance of it to the method

**Interface::ActivateShortcutTable()**.

[Structure ShortcutDescription](#): This structure provides a description of a command for building shortcut tables from static data. The structure holds the command ID that is sent to

**ShortcutCallback::KeyboardShortcut()** method and a resource ID for the string that describes the shortcut. A pointer to an array of these is passed to the **ShortcutTable** constructor.

[Class ClassDesc](#): Two methods have been added to this class that are called at Dll load time to get the shortcut table(s) from the plug-in.

[Class Interface](#): There are new functions added to this class to activate and deactivate a shortcut table. These are normally called by a plug-in in **BeginEditParams()** and **EndEditParams()**. There are also methods that allow one to access the names and command IDs from a specific keyboard shortcut table. See [Keyboard Shortcut Table Related Methods](#) in class Interface.

## Sample Code

For an example of using this mechanism see the code in the directory

**\MAXSDK\SAMPLES\MODIFIERS\FD.**

## Class MacroButtonData

See Also: [Class ToolMacroItem](#), [Class MacroEntry](#), [Class MacroDir](#), [Class ICustButton](#), [Class ICustomControl](#), [Class ShortcutTable](#), [Class ShortcutCallback](#), [Class ActionItem](#), [Keyboard Shortcut System](#).

class MacroButtonData

### Description:

This class is available in release 3.0 and later only.

A Macro Button is a button which can execute either a keyboard macro or macro script. This class contains the data and access methods for such a UI button. This data includes a macro type, command ID, macro script ID, label, tooltip, image name, and image ID.

This object is used in the **ToolMacroItem** constructor. There are also methods of class **ICustButton** to get and set the macro button data.

### Data Members:

public:

**int macroType;**

The macro type. One of the following values:

**MB\_TYPE\_KBD**

A keyboard macro.

**MB\_TYPE\_SCRIPT**

A Script macro.

**ActionTableId tblID;**

The Shortcut Action Table ID.

**void \*cb;**

The ShortcutCallback. See [Class ShortcutCallback](#). This is currently not used.

**int cmdID;**

The command ID. There are method of class **Interface** that provide access to the command IDs for various keyboard shortcut tables. See [Keyboard Shortcut Related Methods](#).

**int macroScriptID;**

The macroScriptID holds the id of the macroScript associated with this button. This id is the **MacroID** that is used by the methods in the **MacroDir** and

**MacroEntry** classes (at one time it was an indirect reference to this id and so was typed as an int). The id can have values from 0 to the number of macro scripts currently defined in the running 3ds max or the special value **UNDEFINED\_MACRO**.

**TCHAR \*label;**

The label text for a text button. This is used if **imageID** is -1.

**TCHAR \*tip;**

The tooltip text.

**TCHAR \*imageName;**

This is the name for the button image. This is the 'base' name only. For example if the actual image name was **Spline\_16i.bmp** then the name supplied here would be **Spline**. See the remarks in [Class CUIFrameMgr](#) for details on the image naming scheme the CUI system uses.

**int imageID;**

The image ID. If this is set to -1 it indicates to use the **label**. If it is set to 0 or greater it indicates this is an image based button and this is the zero based index of the button that was added. This then is an ID into an image group as specified by **imageName**. Said another way, 3ds max builds one large image list internally and uses the **imageName** to get an offset into the list and then uses this **imageID** as an additional offset from the start as indicated by the name (each **imageName** may contain multiple icons in the single BMP).

**ActionItem\* actionItem;**

This data member is available in release 4.0 and later only.

A pointer to the ActionItem.

**DWORD flags;**

This data member is available in release 4.0 and later only.

These flags contain the last state when redrawing

**Methods:**

public:

**Prototype:**

**MacroButtonData();**

**Remarks:**

Constructor. The data members are initialized as follows:

```
label = tip = imageName = NULL; imageID = -1;
```

**Prototype:**

```
MacroButtonData(long tID, void *cb, int cID, TCHAR *lbl,  
TCHAR *tp=NULL, int imID=-1, TCHAR *imName=NULL);
```

**Remarks:**

Constructor. This one is used for keyboard macro buttons (**MB\_TYPE\_KBD**). The data members are initialized to the values passed as shown:

```
macroType=MB_TYPE_KBD; tblID=tID; this->cb=cb;  
cmdID=cID; imageID=imID; label=NULL; SetLabel(lbl);  
tip=NULL; SetTip(tp); imageName=NULL;  
SetImageName(imName);
```

**Prototype:**

```
MacroButtonData(int msID, TCHAR *lbl, TCHAR *tp=NULL,  
int imID=-1, TCHAR *imName=NULL);
```

**Remarks:**

Constructor. This one is used for macro script buttons (**MB\_TYPE\_SCRIPT**). The data members are initialized to the values passed as shown:

```
macroType=MB_TYPE_SCRIPT; macroScriptID=msID;  
imageID=imID; label=NULL; SetLabel(lbl); tip=NULL;  
SetTip(tp); imageName=NULL; SetImageName(imName);
```

**Prototype:**

```
~MacroButtonData();
```

**Remarks:**

Destructor. Any label, tooltip or image name strings are deleted.

**Prototype:**

**void SetLabel(TCHAR \*lbl);**

**Remarks:**

Sets the label text.

**Parameters:**

**TCHAR \*lbl**

The label to set.

**Prototype:**

**TCHAR \*GetLabel();**

**Remarks:**

Returns the label text.

**Prototype:**

**void SetTip(TCHAR \*tp);**

**Remarks:**

Sets the tooltip text.

**Parameters:**

**TCHAR \*tp**

The text to set.

**Prototype:**

**TCHAR \*GetTip();**

**Remarks:**

Returns the tooltip text.

**Prototype:**

**void SetCmdID(int id);**

**Remarks:**

Sets the command ID.

**Parameters:**

**int id**

The command ID to set.

**Prototype:**

**int GetCmdID();**

**Remarks:**

Returns the command ID.

**Prototype:**

**void SetScriptID(int id);**

**Remarks:**

Sets the script ID.

**Parameters:**

**int id**

The script ID to set.

**Prototype:**

**int GetScriptID();**

**Remarks:**

Returns the script ID.

**Prototype:**

**void SetImageName(TCHAR \*imName);**

**Remarks:**

Sets the image name. See the **imageName** data member above for details on the name format.

**Parameters:**

**TCHAR \*imName**

The name to set.

**Prototype:**

**TCHAR \*GetImageName();**

**Remarks:**

Returns the image name.

**Prototype:**

**void SetImageID(int id);**

**Remarks:**

Sets the image ID.

**Parameters:**

**int id**

The image ID to set.

**Prototype:**

**int GetImageID();**

**Remarks:**

Returns the image ID.

**Prototype:**

**void SetTblID(ActionTableId id);**

**Remarks:**

This method is available in release 4.0 and later only.

This method sets the ActionTableID ID.

**Parameters:**

**ActionTableId id**

The ActionTableID ID to set.

**Prototype:**

**ActionTableId GetTblID();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the ActionTableID ID.

**Prototype:**

**void SetActionItem(ActionItem\* pAction);**

**Remarks:**

This method is available in release 4.0 and later only.  
This method allows you to set the ActionItem.

**Parameters:**

**ActionItem\* pAction**

A point to the ActionItem to set.

**Prototype:**

**ActionItem\* GetActionItem();**

**Remarks:**

This method is available in release 4.0 and later only.  
This method returns a pointer to the ActionItem.

**Prototype:**

**BOOL IsActionButton();**

**Remarks:**

This method is available in release 4.0 and later only.  
This method returns TRUE if the button is an Action button. FALSE if it is not.

**Prototype:**

**MacroButtonData & operator=(const MacroButtonData& mbd);**

**Remarks:**

Assignment operator.

## Class ICUIFrame

See Also: [Class CUIFrameMgr](#), [Class CUIFrameMsgHandler](#), [Class ICustToolbar](#), [Class ICustomControl](#).

```
class ICUIFrame : public ICustomControl
```

### Description:

This class is available in release 3.0 and later only.

This class provides access to an individual CUI Frame (the name given to the windows that contain toolbars, menus, the command panel, etc.)

To create a floating tool palette, for example, you create a CUIFrame, create a toolbar, and then attach the two (similar to creating a custom edit field, and custom spinner, then attaching the two). This is done using the method

**ICustToolbar::LinkToCUIFrame()**.

When a Toolbar is part of a CUI frame it's called a Tool Palette. Tool Palettes can either float or dock (whereas a Toolbar must be placed by the developer in a dialog using the resource editor).

**The following global functions are not part of this class but are available for use:**

### Function:

```
ICUIFrame *GetICUIFrame(HWND hCtrl);
```

### Remarks:

Initializes the pointer to the CUI Frame control.

### Parameters:

**HWND hCtrl**

The window handle of the control.

### Function:

```
void ReleaseICUIFrame(ICUIFrame *icf);
```

### Remarks:

Releases the specified control.

### Parameters:

## **ICUIFrame \*icf**

Points to the frame to release.

### **Function:**

**HWND CreateCUIFrameWindow(HWND hParent, TCHAR \*title, int x, int y, int cx, int cy);**

### **Remarks:**

Creates a CUI Frame Window with the specified window handle, size and title parameters. Values of 0 may be passed for x, y, cx and cy. This indicates that the initial size doesn't matter. For example, when the 3ds max CUI is created initially everything is docked. 3ds max then calls

**CUIFrameMgr::RecalcLayout()** which computes all the sizes. Thus the values passed don't matter since they are all going to be recalculated anyway.

### **Parameters:**

#### **HWND hParent**

The handle of the parent window for the frame.

#### **TCHAR \*title**

The title for the frame. This effectively calls **SetName()** below to establish a name for the frame.

#### **int x**

The x coordinate of the upper left corner.

#### **int y**

The y coordinate of the upper left corner.

#### **int cx**

The x size.

#### **int cy**

The y size.

### **Return Value:**

If the function succeeds, the return value is the window handle to the dialog box. If the function fails, the return value is NULL.

### **Methods:**

public:

**Prototype:**

**virtual void SetName(TCHAR \*name)=0;**

**Remarks:**

Sets the name for the frame. This name shows up as a tooltip and also on the window border if the frame is floated. Note that the name is also used to store position information in the CUI file. Therefore developers must use a name and not simply set it to NULL.

**Parameters:**

**TCHAR \*name**

The name to set.

**Prototype:**

**virtual TCHAR \*GetName()=0;**

**Remarks:**

Returns the name of the frame.

**Prototype:**

**virtual void SetPosType(DWORD t)=0;**

**Remarks:**

Sets the position type. This determines the possible locations for the frame.

**Parameters:**

**DWORD t**

The position to set. See [List of CUI Frame Position Types](#).

**Prototype:**

**virtual DWORD GetPosType()=0;**

**Remarks:**

Returns a DWORD which describes the position options for this CUI Frame. See [List of CUI Frame Position Types](#).

**Prototype:**

**virtual void SetPosRank(int rank, int subrank=0)=0;**

**Remarks:**

This method is for internal use only. Developers must not assign the rank and subrank as these are computed internally. Developers should create their toolbars 'floating'. Then when a user docks the toolbar it will be docked automatically when 3ds max starts up the next time.

**Prototype:**

**virtual int GetPosRank()=0;**

**Remarks:**

Returns the position rank. Consider three docked toolbars, one alone on the top line, and two side by side on the line below. The top toolbar would have a rank of 0 (and a subrank of 0). The toolbar on the left in the line below would have a rank of 1 and a subrank of 0. The toolbar beside it to the right would have a rank of 1 and a subrank of 1.

**Prototype:**

**virtual int GetPosSubrank()=0;**

**Remarks:**

Returns the position subrank. See **GetPosRank()** above.

**Prototype:**

**virtual BOOL IsFloating()=0;**

**Remarks:**

Returns TRUE if the frame is floating (not docked); otherwise FALSE.

**Prototype:**

**virtual void Hide(BOOL b)=0;**

**Remarks:**

Sets the frame to hidden or not. Note that if a developer is doing something like showing their toolbars at **BeginEditParams()** and hiding them at **EndEditParams()** then this method can be used. In such a case, if the toolbar is docked then **RecalcLayout()** needs to be called to update the layout. If the toolbars are floating then **RecalcLayout()** does not need to be called.

**Parameters:**

**BOOL b**

TRUE for hidden; FALSE for not hidden.

**Prototype:**

**virtual BOOL IsHidden()=0;**

**Remarks:**

Returns TRUE if the frame is hidden; FALSE if visible.

**Prototype:**

**virtual void SetCurPosition(DWORD pos)=0;**

**Remarks:**

This method is for internal use only.

**Prototype:**

**virtual DWORD GetCurPosition()=0;**

**Remarks:**

Returns the current position of the frame. One of the following values.

**CUI\_TOP\_DOCK**

Docked at the top.

**CUI\_BOTTOM\_DOCK**

Docked at the bottom.

**CUI\_LEFT\_DOCK**

Docked on the left.

**CUI\_RIGHT\_DOCK**

Docked on the right.

**CUI\_FLOATING**

The frame is floating (not docked).

**Prototype:**

**virtual void SetContentType(DWORD t)=0;**

**Remarks:**

Sets the frame contents type. This specifies if this frame holds a toolbar, menu or a floating panel.

**Parameters:**

**DWORD t**

One or more of the following flags:

**CUI\_TOOLBAR**

Set if frame holds toolbars and / or tool palettes.

**CUI\_MENU**

This is used internally to set if the frame holds a menu. Note: Developers should not create their own menus. 3ds max assumes that only one menu exists.

**CUI\_HWND**

Set if frame hold a generic window handle. The command panel (which can be floated) is an example of a generic window.

**Prototype:**

**virtual DWORD GetContentType()=0;**

**Remarks:**

Returns a value which indicates the frame contents type. One of the following flags:

**CUI\_TOOLBAR**

Set if frame holds toolbars and / or tool palettes.

**CUI\_MENU**

Set if the frame holds a menu.

**CUI\_HWND**

Set if frame hold a generic window handle.

**Prototype:**

**virtual void SetContentHandle(HWND hContent)=0;**

**Remarks:**

Sets the content handle. This is the window handle for the toolbar, or menu handle for a menu. Developers typically create Tool Palettes by linking a toolbar to a CUIFrame using **ICustToolbar::LinkToCUIFrame()** and do

not need to call this method as it's done automatically.

**Parameters:**

**HWND hContent**

The handle to set.

**Prototype:**

**virtual HWND GetContentHandle()=0;**

**Remarks:**

Returns the content handle.

**Prototype:**

**virtual void SetTabbedToolbar(BOOL b)=0;**

**Remarks:**

Sets if this frame represents a tabbed toolbar. A tabbed toolbar may have individual tabs added and deleted.

**Parameters:**

**BOOL b**

TRUE for tabbed; FALSE for not tabbed.

**Prototype:**

**virtual BOOL GetTabbedToolbar()=0;**

**Remarks:**

Returns TRUE if this frame is a tabbed toolbar; otherwise FALSE.

**Prototype:**

**virtual void AddToolbarTab(HWND hTBar,  
CUIFrameMsgHandler \*msgHandler, TCHAR \*name, int pos =  
-1)=0;**

**Remarks:**

Adds the toolbar tab whose window handle is passed to the list of tabs maintained by this class.

**Parameters:**

**HWND hTBar**

The window handle of the toolbar tab to add.

**CUIFrameMsgHandler \*msgHandler**

The message handler for the tab or NULL if not used.

**TCHAR \*name**

The name for the tab.

**int pos = -1**

The position for the tab. This is the zero based index where 0 is at left edge of those in the frame and the max value is **GetToolBarCount()-1**. A value of -1 adds the tab to the end of the list of tabs. Also, for example, if you specify a value of 0 for an existing tabbed toolbar, the new tab is inserted at position 0 and the others are moved to the right.

**Prototype:**

```
virtual void DeleteToolBarTab(int pos)=0;
```

**Remarks:**

Deletes the specified toolbar tab.

**Parameters:****int pos**

Specifies the position of the tab to delete. This is the zero based index where 0 is at left edge of those in the frame and the max value is **GetToolBarCount()-1**.

**Prototype:**

```
virtual int GetToolBarCount()=0;
```

**Remarks:**

Returns the number of toolbar tabs in this frame.

**Prototype:**

```
virtual HWND GetToolBarHwnd(int pos)=0;
```

**Remarks:**

Returns the window handle of the toolbar tab whose position is passed.

**Parameters:**

**int pos**

Specifies the position of the tab. This is the zero based index where 0 is at left edge of those in the frame and the max value is **GetToolBarCount()-1**.

**Prototype:**

**virtual TCHAR \*GetTabName(int pos)=0;**

**Remarks:**

Returns the name of the specified toolbar tab.

**Parameters:**

**int pos**

Specifies the position of the tab. This is the zero based index where 0 is at left edge of those in the frame and the max value is **GetToolBarCount()-1**.

**Prototype:**

**virtual void SetCurrentTab(int pos)=0;**

**Remarks:**

Sets the currently active tab.

**Parameters:**

**int pos**

Specifies the position of the tab. This is the zero based index where 0 is at left edge of those in the frame and the max value is **GetToolBarCount()-1**.

**Prototype:**

**virtual int GetCurrentTab()=0;**

**Remarks:**

Returns the position of the currently active tab. This is the zero based index where 0 is at left edge of those in the frame and the max value is **GetToolBarCount()-1**.

**Prototype:**

**virtual int GetSize(int sizeType, int dir, int orient)=0;**

**Remarks:**

Returns the size of this frame for the specified size type, direction (width or height) and orientation.

Note: If this frame has a custom message handler (a **CUIFrameMsgHandler** object, it's **ProcessMessage()** method is called passing **CUI\_POSDATA\_MSG** which is used to determine the size.

**Parameters:****int sizeType**

The size type. See [List of CUI Frame Size Types](#).

**int dir**

The direction of the frame.

**CUI\_HORIZ**: Width.

**CUI\_VERT**: Height.

**int orient**

The orientation. See [List of CUI Frame Orientations](#).

**Prototype:**

```
virtual BOOL InstallMsgHandler(CUIFrameMsgHandler  
*msgHandler)=0;
```

**Remarks:**

Installs a custom message processing routine.

**Parameters:****CUIFrameMsgHandler \*msgHandler**

Points to the handler to install. See [Class CUIFrameMsgHandler](#) for details.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
virtual BOOL SetMenuDisplay(int md)=0;
```

**Remarks:**

Sets the menu display state.

**Parameters:**

**int md**

One of the following values:

**CUI\_MENU\_HIDE**

**CUI\_MENU\_SHOW\_ENABLED**

**CUI\_MENU\_SHOW\_DISABLED**

**Prototype:**

**virtual int GetMenuDisplay()=0;**

**Remarks:**

Returns the state of the menu display. One of the following values:

**CUI\_MENU\_HIDE**

**CUI\_MENU\_SHOW\_ENABLED**

**CUI\_MENU\_SHOW\_DISABLED**

**Prototype:**

**virtual void SetSystemWindow(BOOL b)=0;**

**Remarks:**

System windows are those that come up automatically inside MAX. So command panel, the main toolbar, the tab panel and the main menu are all system windows. It is not therefore appropriate for developers to declare themselves as system windows and this method should not be called passing TRUE.

**Parameters:**

**BOOL b**

TRUE to set as a system window; otherwise FALSE.

**Prototype:**

**virtual BOOL GetSystemWindow()=0;**

**Remarks:**

Returns TRUE if this CUIFrame is a system window; otherwise FALSE. See **SetSystemWindow()** above.

**Prototype:**

**virtual BOOL ReadConfig(TCHAR \*cfg, int startup=FALSE)=0;**

**Remarks:**

This method is for internal use only.

**Prototype:**

**virtual void WriteConfig(TCHAR \*cfg)=0;**

**Remarks:**

This method is for internal use only.

## Class CUIFrameMgr

See Also: [Class ICUIFrame](#), [Class CUIFrameMsgHandler](#), [Class ICustToolbar](#), [Class ICustomControl](#), [Class ICustStatus](#), [Class MAXBmpFileIcon](#).

```
class CUIFrameMgr : public BaseInterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

\*\*\*reflect changes with MAXBMPFileIcon class\*\*\*

This object controls the overall operation of the individual CUI Frames (the name given to the windows that contain toolbars, menus, the command panel, etc.). There is one instance of this CUIFrameMgr class (obtained by calling the global function **GetCUIFrameMgr()**). Methods of this class are available to do things like float and dock individual windows, get pointers to frames, get pointers to button and status controls, and bring up the standard toolbar right click menu .

Note: Developers may use their own images on icon buttons that are managed by this class but the following guidelines must be followed:

BMP files must be put in the **\UIIcons** folder. This is the UI directory under the 3ds max EXE directory. This is hard coded because it must be retrieved before 3ds max is fully started and thus there is no configurable path for it. There is a command line option however, (-c), which specifies for 3ds max to look in an alternate directory for the CUI file. In that case the bitmap files should be located in the same directory.

For more information on the new icon image system refer to the chapter on [external icons](#).

**The following global functions are not part of this class but are available for use:**

### Function:

```
CUIFrameMgr *GetCUIFrameMgr();
```

### Remarks:

Returns a pointer to the CUIFrameMgr which controls the overall operation of CUI Frames (the windows which contain toolbars, menus, the command panel, etc).

**Function:**

**void DoCUICustomizeDialog();**

**Remarks:**

This global function presents the Customize User Interface dialog.

**Function:**

**BOOL AllFloatersAreHidden();**

**Remarks:**

This global function is available in release 4.0 and later only.  
Returns TRUE if all floaters are hidden; otherwise FALSE.

**Methods:**

public:

**Prototype:**

**CUIFrameMgr();**

**Remarks:**

Constructor.

**Prototype:**

**~CUIFrameMgr();**

**Remarks:**

Destructor.

**Prototype:**

**CUIFrameMsgHandler \*GetDefaultMsgHandler();**

**Remarks:**

Returns a pointer to the default CUI Frame Message Handler.

**Prototype:**

**void SetAppHWND(HWND hApp);**

**Remarks:**

This method is for internal use only.

**Prototype:****TCHAR \*GetCUIDirectory();****Remarks:**

Returns the directory name of the custom user interface (CUI) file location.

**Prototype:****void ProcessCUIMenu(HWND hWnd, int x, int y);****Remarks:**

This brings up the CUI right click menu (with the Add Tab, Delete Tab, etc selections). Also see the global function **DoCUICustomizeDialog()**.

**Parameters:****HWND hWnd**

The handle of the window where the mouse was clicked.

**int x**

The x coordinate of the mouse when right clicked.

**int y**

The y coordinate of the cursor when right clicked.

**Prototype:****void DockCUIWindow(HWND hWnd, int panel, RECT \*rp = NULL, int init = FALSE);****Remarks:**

This method docks the CUI window whose handle is passed. Developers who want to dock a window should use this method by passing a rectangle which specifies the general area of the screen where the window is to be docked. This will cause 3ds max reorganize the existing windows.

**Parameters:****HWND hWnd**

The handle of the window to dock.

**int panel**

The CUI docking panel location. One of the following values:

**CUI\_TOP\_DOCK**

Docked at the top.

### **CUI\_BOTTOM\_DOCK**

Docked at the bottom.

### **CUI\_LEFT\_DOCK**

Docked on the left.

### **CUI\_RIGHT\_DOCK**

Docked on the right.

### **RECT \*rp = NULL**

This is the rectangle which specifies where to dock the window. This is the rectangle that a user moves around the screen when dragging a floating window over top of a docking region. This is specified in screen coordinates. If NULL is passed the window is docked using **CUI\_TOP\_DOCK**.

### **int init = FALSE**

This is used internally by 3ds max when it's starting up. This should always default to FALSE (don't override this and pass TRUE).

### **Prototype:**

```
void FloatCUIWindow(HWND hWnd, RECT *rp = NULL, int init = FALSE);
```

### **Remarks:**

Floats (un-docks) the specified CUI Window.

### **Parameters:**

#### **HWND hWnd**

The window to float.

#### **RECT \*rp = NULL**

Specifies the rectangle in screen coordinates where the floating window should reside. If NULL is passed the window is restored to the position it was before it was docked (this information is stored in the CUI file).

Note: Calling this method on an already floating window will explicitly NOT resize the window, but rather just move it to the new origin. Said another way, only the left and top members of the rectangle are used on an already floating window. Developers should call the Win32 API MoveWindow or SetWindowPlacement to size the window. See

**GetFloatingCUIFrameSize()** below to compute a size.

**int init = FALSE**

This is used internally by 3ds max when it's starting up. This should always default to FALSE (don't override this and pass TRUE).

**Prototype:**

**virtual void GetFloatingCUIFrameSize(SIZE \*sz, int rows=1)=0;**

**Remarks:**

Computes the required size of a floating CUI Frame which is linked to a toolbar. The values returned will be zero if the frame is not linked to a toolbar.

**Parameters:**

**SIZE \*sz**

The computed size is returned here. **sz.cx** is the width, **sz.cy** is the height.

**int rows=1**

The number of rows for the toolbar used in the computation.

**Prototype:**

**void SetReservedSize(int panel, int size);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int GetReservedSize(int panel);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int GetPanelSize(int panel, int incReserved = FALSE);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**void RecalcLayout(int entireApp=FALSE);**

**Remarks:**

This method may be called to recalculates the layout of the CUI. Developers need to call this method after they, for example, add new tool palettes. A developer would create the new palettes and then call this method when done. Otherwise the changes wouldn't be reflected until the user redrew the viewports or resized MAX.

**Parameters:**

**int entireApp=FALSE**

TRUE to recalculate the entire application, including the viewports. This can be expensive (basically like an **Interface::ForceCompleteRedraw()**); FALSE will recalculate the top, botton, left and right panels but won't redraw the viewports.

**Prototype:**

**void DrawCUIWindows(int panels=CUI\_ALL\_PANELS);**

**Remarks:**

This method redraws the specified panels. Typically developers don't need to call this method.

**Parameters:**

**int panels=CUI\_ALL\_PANELS**

See [List of CUI Docking Panel Locations](#).

**Prototype:**

**void SetMacroButtonStates(BOOL force);**

**Remarks:**

This method is available in release 4.0 and later only.

This is a very important method. It redraws all the visible CUI buttons in MAX, calling the "IsEnabled" and "IsChecked" handlers on the ActionItems associated with each button (if it has one). If a the "IsEnabled" handler returns FALSE, the button is grayed out. If the "IsChecked" handler return TRUE, the button is draw pressed in.

This method is called internally by the system on selection changes and command mode changes. This handles the majority of the cases where buttons need to be redrawn. However, if a 3rd party plug-in changes some sort of

internal state that might affect the return value of an ActionItem's IsEnables or IsChecked handler, then the plug-in should call this method to update the button states. If this method isn't called, buttons may look disabled or pressed (or visa versa) when they shouldn't be. See [Class ActionItem](#).

**Parameters:**

**BOOL force**

This parameter, if TRUE, tells the system to redraw the button even if its state hasn't changed since the last time it was redrawn. Normally this argument is FALSE so it only redraws the buttons that changed state.

**Prototype:**

**void ResetIconImages();**

**Remarks:**

This method is available in release 4.0 and later only.

This method is for internal use only. This is automatically called when the system changes its custom colors. It tells all the buttons on toolbars to toss their icon image cache.

This method only resets the icons for toolbars that are part of the CUI system, not for toolbars created by other code, which is why the **ICustToolbar** method **ResetIconImages()** is needed. See the method **ICustToolbar::ResetIconImages**.

**Prototype:**

**int OverDockRegion(LPPOINT pt, DWORD posType, int override = FALSE);**

**Remarks:**

Given a point and a position type this method returns nonzero if the point is over the specified docking region; otherwise zero.

**Parameters:**

**LPPOINT pt**

The input point to check in screen coordinates.

**DWORD posType**

See [List of CUI Frame Position Types](#).

**int override = FALSE**

Passing TRUE overrides the docking function so that it won't dock. Passing FALSE will cause it to dock. Also note that if the UI layout is locked, passing TRUE here will override that lock.

In the code fragment below the state of the Ctrl key is checked and used as the docking override.

**Sample Code:**

```
GetCursorPos(&pt);  
overDockRegion = GetCUIFrameMgr()->OverDockRegion(&pt,  
cf->GetPosType(), (GetKeyState(VK_CONTROL) & 0x8000));
```

**Prototype:**

```
HWND GetItemHwnd(int id);
```

**Remarks:**

Returns the window handle for the item whose ID is passed. This correspond to the method in **ICustToolBar** but which should no longer be called for Tool Palettes. It is now also a method of this class because the CUI system doesn't know which toolbar a particular button is on. For example, a 3ds max user in 3.0 can drag a button from one tool palette to another. No longer then can one use the previous **GetItemHwnd()** method since the button has moved to a different toolbar.

**Parameters:**

**int id**

The ID of the control.

**Prototype:**

```
ICustButton *GetICustButton(int id);
```

**Remarks:**

Returns a pointer to the custom button whose ID is passed (or NULL if not found). In the **CUIFrameMgr** implementation of this method it loops through each toolbar that it has control over and calls **ICustToolBar::GetICustButton()** on it. That method returns NULL if it doesn't find the specified ID. The CUIFrameMgr keeps looping through the

toolbars until it gets a non-NULL value. When it finds it it returns the **ICustButton** pointer.

**Parameters:**

**int id**

The ID of the control.

**Prototype:**

**ICustStatus \*GetICustStatus(int id);**

**Remarks:**

Returns a pointer to the custom status control whose ID is passed.

Returns a pointer to the custom status control whose ID is passed (or NULL if not found). In the **CUIFrameMgr** implementation of this method it loops through each toolbar that it has control over and calls

**ICustToolBar::GetICustStatus()** on it. That method returns NULL if it doesn't find the specified ID. The **CUIFrameMgr** keeps looping through the toolbars until it gets a non-NULL value. When it finds it it returns the **ICustStatus** pointer.

**Parameters:**

**int id**

The ID of the control.

**Prototype:**

**ICUIFrame \*GetICUIFrame(int i);**

**Remarks:**

Returns a pointer to the CUI Frame as specified by the index passed.

**Parameters:**

**int i**

The zero based index in the list of frames (between **0** and **GetCount()-1**).

**Prototype:**

**ICUIFrame \*GetICUIFrame(TCHAR \*name);**

**Remarks:**

Returns a pointer to the CUI Frame as specified by the name passed.

**Parameters:**

**TCHAR \*name**

The name of the frame.

**Prototype:**

**ICUIFrame \*GetICUIFrame(int panel, int rank, int subrank);**

**Remarks:**

Returns a pointer to the CUI Frame as specified by the panel, rank and subrank passed.

**Parameters:**

**int panel**

One of the following values:

**int rank**

The zero based rank index.

**int subrank**

The zero based sub-rank index.

**Prototype:**

**TCHAR \*GetConfigFile();**

**Remarks:**

This returns the path to the CUI file in use. This may be a UNC name.

**Prototype:**

**int GetButtonHeight(int sz=0);**

**Remarks:**

Returns the bitmap button image height for the specified size.

**Parameters:**

**int sz=0**

The size to check. If 0 is passed then the current icon size is checked. One of the following values:

**CUI\_SIZE\_16**

## **CUI\_SIZE\_24**

### **Prototype:**

**int GetButtonWidth(int sz=0);**

### **Remarks:**

Returns the bitmap button image width for the specified size.

### **Parameters:**

**int sz=0**

The size to check. One of the following values:

**CUI\_SIZE\_16**

**CUI\_SIZE\_24**

### **Prototype:**

**void SetMode(int md);**

### **Remarks:**

This method is for internal use only.

### **Prototype:**

**int GetMode();**

### **Remarks:**

This method is for internal use only.

### **Prototype:**

**void ExpertMode(int onOff);**

### **Remarks:**

This method is for internal use only. Calling this method alone will not put 3ds max in Expert mode.

### **Prototype:**

**void HorizTextButtons(BOOL b);**

### **Remarks:**

This method is for internal use only.

**Prototype:**

**int GetHorizTextButtons();**

**Remarks:**

This method is for internal use only.

**Prototype:**

**void FixedWidthTextButtons(BOOL b);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int GetFixedWidthTextButtons();**

**Remarks:**

This method is for internal use only.

**Prototype:**

**void SetTextButtonWidth(int w);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int GetTextButtonWidth();**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int GetCount();**

**Remarks:**

Returns the number of frames that exist.

**Prototype:**

**int SetConfigFile(TCHAR \*cfg);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int DeleteSystemWindows(int toolbarsOnly = TRUE);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int CreateSystemWindows(int reset = FALSE);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**void SetImageSize(int size);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int GetImageSize();**

**Remarks:**

This method is for internal use only.

**Prototype:**

**void SetDefaultData(CUIFrameMsgHandler \*msg,  
HIMAGELIST img16, HIMAGELIST img24=NULL);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int GetDefaultImageListBaseIndex(SClass\_ID sid, Class\_ID cid);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used internally to create a MaxBmpFileIcon for a given object type. These methods retrieve the file name and base index in the file of the icon for the given object class. They are used in the constructor for MaxBmpFileIcon that takes a class ID and super class ID. This method is for internal use only.

**Prototype:**

**TSTR\* GetDefaultImageListFilePrefix(SClass\_ID sid, Class\_ID cid);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used internally to create a MaxBmpFileIcon for a given object type. These methods retrieve the file name and base index in the file of the icon for the given object class. They are used in the constructor for MaxBmpFileIcon that takes a class ID and super class ID. This method is for internal use only.

**Prototype:**

**int AddToRawImageList(TCHAR\* pFilePrefix, int sz, HBITMAP image, HBITMAP mask);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is for internal use only. It is used to add images to the icon manager. The icon manager, which is used to implement the **MaxBmpFileIcon** class, reads all the .bmp files in the UI/Icons directory at startup time. These icons are specified by an image file and an alpha mask. The icons support two sizes. Large, which is 24 by 24 and small, which is 15 by 16. The icon manager stores the unprocessed image and alpha masks (the "raw" images). Whenever an instance of MaxBmpFileIcon needs to draw itself, it gets the image list and index of the icon in the imagelist using

**GetSmallImageIndex** or **GetLargeImageIndex**.

**Prototype:**

**int LoadBitmapFile(TCHAR \*filename);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int LoadBitmapImages();**

**Remarks:**

This method is for internal use only.

**Prototype:**

**int ReadConfig();**

**Remarks:**

Plug-In developers should not call this method -- it is for internal use only.

**Prototype:**

**int WriteConfig();**

**Remarks:**

Plug-In developers should not call this method -- it is for internal use only.

**Prototype:**

**void SetLockLayout(BOOL lock);**

**Remarks:**

This method is for internal use only.

**Prototype:**

**BOOL GetLockLayout();**

**Remarks:**

Returns TRUE if the layout is locked; FALSE if unlocked.

**Prototype:**

**void EnableAllCUIWindows(int enabled);**

**Remarks:**

This method is for internal use only.

## Class CUIFrameMsgHandler

See Also: [Class ICUIFrame](#), [Class CUIFrameMgr](#), [Class CUIPosData](#), [Class ICustomControl](#).

class CUIFrameMsgHandler

### Description:

This class is available in release 3.0 and later only.

This class provides a way for messages received by a **CUIFrame** to be processed in a context-specific fashion.

Since the CUI Frame is just a window, it needs a window proc. There is one built into the CUI system, but it may need additional information that is specific to how the frame is being used. For example, in 3ds max the command panel can't be resized horizontally and the default window proc can't manage this.

For such situations, the application must install a **CUIFrameMsgHandler** object. You establish that this is the handler for the frame using the method **ICUIFrame::InstallMsgHandler(CUIFrameMsgHandler \*msgHandler)**.

These message handlers have one significant class method: **ProcessMessage()**. If **ProcessMessage()** returns TRUE, then the CUI system assumes that the message is completely handled. If it returns FALSE, then the standard CUI processing takes place. (Note that the message handler may still return FALSE, even if it does some processing...).

There is a special message (**CUI\_POSDATA\_MSG**) that is sent by the CUI system to the message handler to get information on window size constraints, etc. An example of processing this message is shown below. In this case **editPosData** is a static instance of **CUIPosData**. That object has **GetWidth()** and **GetHeight()** methods which return the proper width and height size for various orientations. See [Class CUIPosData](#) for details.

```
case CUI_POSDATA_MSG: {
    CUIPosData **cpd = (CUIPosData **)lParam;
    cpd[0] = &editPosData;
}
return TRUE;
```

**Methods:**

public:

**Prototype:**

```
virtual int ProcessMessage(UINT message, WPARAM wParam,  
LPARAM lParam);
```

**Remarks:**

This method is called to handle any processing not done by the default CUI window proc.

This method should return TRUE if the message is handled and FALSE if not. If FALSE is returned (or no handler is defined), then the CUIFrame simply passes WM\_COMMAND messages on to its parent. Window position messages are passed from the CUIFrame to the HWND of the 'content' (either a toolbar or menu). Other messages are passed on to the default window proc.

Note: Developers should **not** return TRUE for the **entire** ProcessMessage routine, since if this is done, the right-click menu functionality will not work (e.g. docking, floating, move-to-shelf, customize, etc.).

Also Note: Developers should not use IDs that may conflict with the ones used by the default processing provide by MAX. **The IDs which should be avoided are in the 40000, 47000, and 61000 range.** For instance the following IDs are all invalid since they are in those ranges: 40005, 47900, 61102. The reason this is a problem is that if you return FALSE after processing a particular ID, then 3ds max will go ahead and process that ID also. And if the ID matches one already in MAX, an unintended function may get called.

**Parameters:****UINT message**

Specifies the message.

**WPARAM wParam**

Specifies additional message information. The contents of this parameter depend on the value of the **message** parameter.

**LPARAM lParam**

Specifies additional message information. The contents of this parameter depend on the value of the **message** parameter.

**Default Implementation:**

```
{ return FALSE; }
```

## Class CUIPosData

See Also: [Class CUIFrameMsgHandler](#), [Class ICustomControl](#).

class CUIPosData

### Description:

This class is available in release 3.0 and later only.

This is the object that provides the position data when the **CUIFrameMsgHandler::ProcessMessage()** method receives a **CUI\_POSDATA\_MSG** message. The developer creates an instance of this class and implements the **GetWidth()** and **GetHeight()** methods which return size information based on the size type and orientation passed.

### Methods:

public:

### Prototype:

```
virtual ~CUIPosData();
```

### Remarks:

Destructor.

### Default Implementation:

```
{}
```

### Prototype:

```
virtual int GetWidth(int sizeType, int orient);
```

### Remarks:

Returns the width for the specified size type and orientation. A return value of -1 indicates that the frame doesn't have a specific needed value (it doesn't care).

### Parameters:

**int sizeType**

The size type. [See List of CUI Frame Size Types](#).

**int orient**

The orientation. [See List of CUI Frame Orientations](#).

**Default Implementation:**

```
{ return 50; }
```

**Prototype:**

```
virtual int GetHeight(int sizeType, int orient);
```

**Remarks:**

Returns the height for the specified size type and orientation.

**Parameters:**

**int sizeType**

The size type. See [List of CUI Frame Size Types](#).

**int orient**

The orientation. See [List of CUI Frame Orientations](#).

**Default Implementation:**

```
{ return 50; }
```

## Class ToolMacroItem

See Also: [Class ToolItem](#), [Class MacroButtonData](#), [Class ICustButton](#), [Custom Controls](#).

```
class ToolMacroItem : public ToolItem
```

### Description:

This class is available in release 3.0 and later only.

This class allows a macro item control to be added to the toolbar.

### Data Members:

public:

**MacroButtonData md;**

Points to the macro button data for this tool item.

### Methods:

public:

### Prototype:

```
ToolMacroItem(int wd, int ht, MacroButtonData *data, int or =  
CTB_HORIZ|CTB_VERT|CTB_FLOAT);
```

### Remarks:

Constructor.

### Parameters:

**int wd**

The width of the item.

**int ht**

The height of the item.

**MacroButtonData \*data**

Points to the macro button data.

**int or = CTB\_HORIZ|CTB\_VERT|CTB\_FLOAT**

Specifies the orientation. One or more of the following values:

**CTB\_HORIZ**

**CTB\_VERT**

**CTB\_FLOAT**

## Class MacroDir

See Also: [Class MacroEntry](#), [Class MacroButtonData](#).

```
class MacroDir : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This class provides access to Macro scripts. Macro scripts (or macros) are scripts that live in buttons and menus in the customizable UI. Methods of this class are available to access macros using IDs or category and name strings, methods to edit macro scripts, methods to execute macros, and methods for directory scanning and loading.

The directory instance (access via the global function **GetMacroScriptDir()**) is used by the CUI to provide a list of available macros in the toolbar/menu editor. The API also provides a way for the CUI to open a macro editor to allow on-the-fly creation of macro scripts.

The macro script manager keeps a directory of all known macros and provides an API for running and editing macros and for accessing and updating the directory.

Macros are normally entered into the directory by the MAXScript compiler as a side-effect of compiling a macro definition. Anyone using MAXScript can at any time eval a macro definition and thereby add CUI macro scripts.

Consequently, macros can be stored in any script file and be loaded just by executing the file. The macro definition syntax permits any number of macros per file.

Most macros will be stored in files in a special CUI macro or config directory so that a user can take all his custom UI stuff with him by copying directories. (This directory supports recursive scanning of sub-dirs, so users can organize their macros). On-the-fly macro creation in the CUI editor or by text drag-and-drop onto the shelf or by evaling a definition in the listener will generate a file in this directory to provide permanent storage.

Note: **typedef short MacroID;**

Note: In order to use these methods you need to **#include "IMACROSCRIPT.H"** and link to **"MAXSCRIPT.LIB"**.

**The following global functions are not part of this class but are available for use:**

**Function:**

**MacroDir& GetMacroScriptDir();**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns a reference to the macro script directory.

**Function:**

**void InitMacroScriptDir();**

**Remarks:**

This global function is available in release 3.0 and later only.

This is an internal only function used by 3ds max during startup to get the macroScript system initialized.

**Methods:**

public:

**Prototype:**

**virtual MacroEntry\* GetMacro(MacroID mid)=0;**

**Remarks:**

Returns a pointer to the MacroEntry for the macro script whose ID is passed.

**Parameters:**

**MacroID mid**

The ID of the macro.

**Prototype:**

**virtual MacroEntry\* FindMacro(TCHAR\* category, TCHAR\* name)=0;**

**Remarks:**

Returns a pointer to the MacroEntry corresponding to the given category and name strings passed (or NULL if not found).

**Parameters:**

**TCHAR\* category**

The category name.

**TCHAR\* name**

The macro script name.

**Prototype:**

**virtual BOOL ValidID(MacroID mid)=0;**

**Remarks:**

Returns TRUE if the macro ID is valid (or unused); otherwise FALSE.

**Parameters:**

**MacroID mid**

The ID to check.

**Prototype:**

**virtual int Count()=0;**

**Remarks:**

Returns the number of macro entries in this macro directory.

**Prototype:**

**virtual MacroEntry\* GetMacro(int index)=0;**

**Remarks:**

Returns a pointer to the macro entry whose index in the directory is passed.

**Parameters:**

**int index**

The zero based index of the entry. This is a value between 0 and **Count()-1**.

**Prototype:**

**virtual MacroEntry\* AddMacro(TCHAR\* category, TCHAR\* name, TCHAR\* tooltip, TCHAR\* buttonText, TCHAR\* sourceFile, int sourceOffset)=0;**

**Remarks:**

Adds the macro whose parameters are passed and returns a pointer to the new

macro entry. This form allows you to define a macro that is already in a file, by giving a source file name and seek offset into that file. This is typically used by the MAXScript compiler and .mcr file scanner to register macro definitions they come across.

**Parameters:**

**TCHAR\* category**

The category for the macro.

**TCHAR\* name**

The name for the macro.

**TCHAR\* tooltip**

The tooltip text.

**TCHAR\* buttonText**

The button text.

**TCHAR\* sourceFile**

The source file name.

**int sourceOffset**

The line number of the first line of the script in the source file.

**Prototype:**

```
virtual MacroEntry* AddMacro(TCHAR* category, TCHAR*  
name, TCHAR* tooltip, TCHAR* buttonText, TCHAR*  
sourceText)=0;
```

**Remarks:**

Add or replaces a macro using given source text as the body of the macro. In this overload, name can be NULL in which case a unique name is generated.

This form takes the body of the actual macro script as the **sourceText** argument and places it in a newly-created file in the UI directory and registers that file and a zero offset as the macro definition. All macroScripts need to be in files somewhere so that they are persistent if referenced in a CUI toolbar that the user saves. This form is used, for example, by the toolbar manager when you drag a piece of selected text onto a toolbar to cause a script button to be created.

**Parameters:**

**TCHAR\* category**

The category for the macro.

**TCHAR\* name**

The name for the macro or NULL to generate a unique name.

**TCHAR\* tooltip**

The tooltip text.

**TCHAR\* buttonText**

The button text.

**TCHAR\* sourceText**

The source text.

**Prototype:**

```
virtual BOOL SetMacro(MacroID mid, TCHAR* tooltip,  
TCHAR* btnText, TCHAR* sourceFile, int sourceOffset)=0;
```

**Remarks:**

Sets the parameters for the macro entry whose ID is passed.

**Parameters:**

**MacroID mid**

The macro ID.

**TCHAR\* tooltip**

The tooltip text.

**TCHAR\* btnText**

The button text.

**TCHAR\* sourceFile**

The source file name.

**int sourceOffset**

The source offset.

**Return Value:**

TRUE if set; FALSE if the ID was not found.

**Prototype:**

```
virtual TCHAR* MakeNameValid(TCHAR* s)=0;
```

**Remarks:**

This method modifies the string in place to be a valid macro name (no punctuations other than spaces).

**Parameters:**

**TCHAR\* s**

The name string.

**Prototype:**

**virtual TCHAR\* MakeCategoryValid(TCHAR\* s)=0;**

**Remarks:**

This method modifies the string in place to be a valid category name (no punctuations other than spaces).

**Parameters:**

**TCHAR\* s**

The category string.

**Prototype:**

**virtual BOOL EditMacro(MacroID mid)=0;**

**Remarks:**

This methods brings up the editor for editing the specified macro script text.

**Parameters:**

**MacroID mid**

The ID of the macro script to edit.

**Prototype:**

**virtual Value\* Execute(MacroID mid)=0;**

**Remarks:**

Executes the macro script whose ID is passed.

**Parameters:**

**MacroID mid**

The ID of the macro to execute.

**Return Value:**

A pointer to the result of executing the macro. If a developer doesn't care about the result of executing a macroScript, which is usually the case, then the **Value\*** returned from this method can just be ignored. If a developer does care, then the necessary information about working with **Value\***'s is in the [MAXScript SDK](#) documentation.

**Prototype:**

**virtual void LoadMacroScripts(TCHAR\* path\_name = NULL, BOOL recurse = TRUE)=0;**

**Remarks:**

This method loads all the macro scripts found in the specified path and optionally its sub-directories. You can point this method at any directory and it will scan it for .mcr files and scan those for macroScript definitions. 3ds max uses this during startup to scan the UI directory (recursively) for .mcr files.

**Parameters:**

**TCHAR\* path\_name = NULL**

The path to check. If NULL the default path is used.

**BOOL recurse = TRUE**

If TRUE nested sub-directories are scanned and loaded as well.

**Prototype:**

**virtual void SetMacroScriptPath(TCHAR\* path\_name)=0;**

**Remarks:**

Sets the default path for storing / searching macro script files.

**Parameters:**

**TCHAR\* path\_name**

The path to set.

**Prototype:**

**virtual TCHAR\* GetMacroScriptPath()=0;**

**Remarks:**

Returns the default path for storing / searching macro script files.

## Class MacroEntry

See Also: [Class MacroDir](#), [Class MacroButtonData](#).

```
class MacroEntry : public BaseInterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This class provides access to a single macro entry. There are methods provided to access the macro ID, name, category, file name, tooltip, UI button text, and the UI button icon. MacroEntries are returned from methods of **class**

### MacroDir.

Macro scripts (or macros) are scripts that live in buttons and menus in the new customizable UI. The macro script manager keeps a directory of all known macros and provides an API for running and editing macros and for accessing and updating the directory.

All macro scripts have a name and a category. The category is used to organize the macros into groupings and is given to the macro script at definition time. If you look at the macro scripts list in the UI Customize dialog, you see a Category dropdown with things like 'Cameras', 'DragandDrop', 'LightTools', etc., which is derived from the all categories present in the currently-defined macroscripts.

Note that the normal way to define a macroScript is through MAXScript, the .mcr files in the UI\MacroScripts directory contain examples, and they all have category definitions.

Note: In order to use these methods you need to **#include** "**IMACROSCRIPT.H**" and link to "**MAXSCRIPT.LIB**".

### Methods:

public:

### Prototype:

```
virtual MacroID GetID()=0;
```

### Remarks:

Returns the ID for this macro script. Note: **typedef short MacroID;**

### Prototype:

```
virtual TSTR& GetName()=0;
```

**Remarks:**

Returns the name for this macro script.

**Prototype:**

**virtual TSTR& GetCategory()=0;**

**Remarks:**

Returns the category for this macro script.

**Prototype:**

**virtual TSTR& GetFileName()=0;**

**Remarks:**

Returns the file name of the script source text for this script.

**Prototype:**

**virtual void SetFileName(TCHAR\* fn)=0;**

**Remarks:**

Sets the file name for the script source.

**Parameters:**

**TCHAR\* fn**

The file name to set.

**Prototype:**

**virtual void SetOffset(int o)=0;**

**Remarks:**

Sets the offset for this macro script entry. There can be any number of macroScripts in a single source file and the offset keeps track of the beginning of its definition in the file.

**Parameters:**

**int o**

The offset to set.

**Prototype:**

**virtual long GetOffset()=0;**

**Remarks:**

Returns the offset for this macro script entry. See **SetOffset()** above.

**Prototype:**

**virtual Value\* GetCode()=0;**

**Remarks:**

When the macroScript is defined, only its source file and source offset are registered. When the user first runs it, the MAXScript compiler is used to compile the definition into executable code, which is then cached and used for any later executions and is what this method returns. If this returns NULL, the macro hasn't been compiled or run yet. Another way to run the macro is via the **MacroEntry::Execute()** and this causes the code to be cached as a side effect also. Normally, developers only ever need to use the **Execute()** method, but if they are using the MAXScript SDK, they can grab the code using **GetCode()** and work with it directly.

**Prototype:**

**virtual TSTR& GetToolTip()=0;**

**Remarks:**

Returns the tooltip text for the UI button.

**Prototype:**

**virtual void SetToolTip(TCHAR\* tt)=0;**

**Remarks:**

Sets the tooltip text for the UI button.

**Parameters:**

**TCHAR\* tt**

The tooltip text to set.

**Prototype:**

**virtual TSTR& GetButtonText()=0;**

**Remarks:**

Returns the UI button text (for label buttons).

**Prototype:**

**virtual void SetButtonText(TCHAR\* bt)=0;**

**Remarks:**

Sets the UI button text (for label buttons).

**Parameters:**

**TCHAR\* bt**

The button text.

**Prototype:**

**virtual void SetButtonIcon(TCHAR\* icnf, int indx)=0;**

**Remarks:**

Sets the UI button icon via a base icon file name and index into the specified BMP bitmap.

**Parameters:**

**TCHAR\* icnf**

The file name of the BMP file. See the remarks in [Class CUIFrameMgr](#) for details on the naming scheme.

**int indx**

The zero based index of the icon in the BMP file. The first icon is 0, the second is 1, etc.

**Prototype:**

**virtual TSTR& GetButtonIconFile()=0;**

**Remarks:**

Returns the file name of the icon file.

**Prototype:**

**virtual int GetButtonIconIndex()=0;**

**Remarks:**

Returns the zero based index of the icon in the icon file.

**Prototype:**

**virtual void SetFlags(short mask)=0;**

**Remarks:**

Sets the specified flags.

**Parameters:**

**short mask**

The flags to set. One or more of the following values:

**ME\_DROPPED\_SCRIPT**

Macro made from some drag-and-dropped text.

**ME\_SILENT\_ERRORS**

Macro won't report any runtime errors.

**Prototype:**

**virtual void ClearFlags(short mask)=0;**

**Remarks:**

Clears the specified flags.

**Parameters:**

**short mask**

The flags to clear. One or more of the following values:

**ME\_DROPPED\_SCRIPT**

Macro made from some drag-and-dropped text.

**ME\_SILENT\_ERRORS**

Macro won't report any runtime errors.

**Prototype:**

**virtual short GetFlags(short mask)=0;**

**Remarks:**

Returns the state of the specified flags.

**Parameters:**

**short mask**

The flags to get. One or more of the following values:

**ME\_DROPPED\_SCRIPT**

Macro made from some drag-and-dropped text.

**ME\_SILENT\_ERRORS**

Macro won't report any runtime errors.

**Prototype:**

**virtual Value\* Execute()=0;**

**Remarks:**

Executes this macro entry.

**Return Value:**

A pointer to the result of executing the macro. If a developer doesn't care about the result of executing a macro script, which is usually the case, then the

**Value\*** returned from this method can just be ignored. If a developer does care, then the necessary information about working with **Value\***'s is in the

[MAXScript SDK](#) documentation.

## Class ISliderControl

See Also: [Class ICustomControl](#), [Class ISpinnerControl](#).

```
class ISliderControl : public ICustomControl
```

### Description:

This class is available in release 3.0 and later only.

**Important Note: The slider control ensures that it only displays, and the user is only allowed to input, values within the specified ranges. However the slider is just a front end to a controller which actually controls the value. The user can thus circumvent the slider constraints by editing the controller directly (via function curves in track view, key info, etc.). Therefore, when a plug-in gets a value from a controller (or a parameter block, which may use a controller) it is its responsibility to clamp the value to a valid range.**



(Slider Control)



('Bracketed' Slider Control)

The custom slider control is functionality similar to the custom spinner control. It supports the following features:

- can link to custom edit box.
- right click reset of value.  
if not dragging, resets to default reset value.  
if dragging, resets to previous value.
- shift+right click sets an animation key.
- red highlight for animated key positions.

It also supports the following functionality:

- dynamically set tick marks segment the slider track.
- default reset value and last value are visually indicated.
- left click in slider track moves button to that position.
- ctrl key snaps to nearest tick mark.

Also Note: Developers should use the functions **Get/SetSpinnerPrecision()** for controlling precision of edit boxes linked to slider controls. Those functions affect both spinners and sliders.

To initialize the pointer to the control call:

**Function:****ISliderControl \*GetISlider(HWND hCtrl);**

To release the control call:

**Function:****void ReleaseISlider(ISliderControl \*isc);**

The value to use in the Class field of the Custom Control Properties dialog is:

**SliderControl**

The following messages may be sent by the slider control:

This message is sent when the value of a slider changes.

**CC\_SLIDER\_CHANGE**

**lParam** contains a pointer to the slider control. You can cast this pointer to a **ISliderControl** type and then call methods of the control.

**LOWORD(wParam)** contains the ID of the slider. This is the ID established in the ID field of the Custom Control Properties dialog.

**HIWORD(wParam)** is TRUE if the user is dragging the slider interactively.

This message is sent when the user presses down on the slider.

**CC\_SLIDER\_BUTTONDOWN**

**lParam** contains a pointer to the slider control. You can cast this pointer to a **ISliderControl** type and then call methods of the control.

**LOWORD(wParam)** contains the ID of the slider. This is the ID established in the ID field of the Custom Control Properties dialog.

This message is sent when the user releases a slider.

**CC\_SLIDER\_BUTTONUP**

**lParam** contains a pointer to the slider control. You can cast this pointer to a **ISliderControl** type and then call methods of the control.

**LOWORD(wParam)** contains the ID of the slider. This is the ID established in the ID field of the Custom Control Properties dialog.

**HIWORD(wParam)** is FALSE if the user canceled and TRUE otherwise.

**Methods:**

public:

**Prototype:**

**virtual float GetFVal()=0;**

**Remarks:**

Returns the floating point value of the control.

**Prototype:**

**virtual int GetIVal()=0;**

**Remarks:**

Returns the integer value of the control.

**Prototype:**

**virtual void SetNumSegs(int num)=0;**

**Remarks:**

Sets the number of segments (tick marks) used by the control.

**Parameters:**

**int num**

The number to set.

**Prototype:**

**virtual void SetValue(float v, int notify)=0;**

**Remarks:**

This method sets the value of the control to the specific floating point number passed. You may pass FALSE as the notify parameter so the control wont send a message when you set the value.

**Parameters:**

**float v**

The new value for the control.

**int notify**

If TRUE a message is sent indicating the control has changed; if FALSE no message is sent.

**Prototype:**

**virtual void SetValue(int v, int notify)=0;**

**Remarks:**

This method sets the value of the control to the specific integer number passed. You may pass FALSE as the notify parameter so the control wont send a message when you set the value.

**Parameters:**

**int v**

The new value for the control.

**int notify**

If TRUE a message is sent indicating the control has changed; if FALSE no message is sent.

**Prototype:**

**virtual void SetLimits(int min, int max, int limitCurValue = TRUE)=0;**

**Remarks:**

This method establishes the allowable limits for integer values entered.

**Parameters:**

**int min**

The minimum allowable value.

**int max**

The maximum allowable value.

**int limitCurValue = TRUE**

You may pass FALSE to the this parameter so the control will not send a spinner changed message when the limits are set.

**Prototype:**

**virtual void SetLimits(float min, float max, int limitCurValue = TRUE)=0;**

**Remarks:**

This method establishes the allowable limits for floating point values entered.

**Parameters:**

**float min**

The minimum allowable value.

**float max**

The maximum allowable value.

**int limitCurValue = TRUE**

You may pass FALSE to the this parameter so the control will not send a spinner changed message when the limits are set.

**Prototype:**

**virtual void LinkToEdit(HWND hEdit, EditSpinnerType type)=0;**

**Remarks:**

When an edit control is used in conjunction with the slider control, this method is used to link the two, so values entered using the slider are displayed in the edit control. This method is also used to set the type of value which may be entered.

**Parameters:****HWND hEdit**

The handle of the edit control to link.

**EditSpinnerType type**

The type of value that may be entered. One of the following values:

**EDITTYPE\_INT**

Any integer value.

**EDITTYPE\_FLOAT**

Any floating point value.

**EDITTYPE\_UNIVERSE**

This is a value in world space units. It respects the system's unit settings (for example feet and inches).

**EDITTYPE\_POS\_INT**

Any integer  $\geq 0$

**EDITTYPE\_POS\_FLOAT**

Any floating point value  $\geq 0.0$

**EDITTYPE\_POS\_UNIVERSE**

This is a positive value in world space units. It respects the system's unit

settings (for example feet and inches) .

### **EDITTYPE\_TIME**

This is a time value. It respects the system time settings (SMPTE for example).

#### **Prototype:**

**virtual void SetIndeterminate(BOOL i=TRUE)=0;**

#### **Remarks:**

This method is used to show commonality. When several different values are being reflected by the slider, the value is indeterminate. When TRUE, the value field of the slider appears empty.

#### **Parameters:**

**BOOL i=TRUE**

Pass TRUE to this method to set the value to indeterminate.

#### **Prototype:**

**virtual BOOL IsIndeterminate()=0;**

#### **Remarks:**

This method returns TRUE if the current state of the slider is indeterminate; otherwise FALSE. See **SetIndeterminate()** above.

#### **Prototype:**

**virtual void SetResetValue(float v)=0;**

#### **Remarks:**

A user may right click on a slider to reset it to its 'reset' value (after it has been changed). This method specifies the value used when the reset occurs.

#### **Parameters:**

**float v**

The reset value.

#### **Prototype:**

**virtual void SetResetValue(int v)=0;**

**Remarks:**

A user may right click on a slider to reset it to its 'reset' value (after it has been changed). This method specifies the value used when the reset occurs.

**Parameters:**

**int v**

The reset value.

**Prototype:**

**virtual void SetKeyBrackets(BOOL onOff)=0;**

**Remarks:**

Sets the display of the 'brackets' surrounding the slider control. This is used to indicate if a key exists for the parameter controlled by the slider at the current time. These brackets turned on and off automatically if you are using a parameter map and parameter block to handle the control. If not you'll need to use this method. For a slider, the 'brackets' appear as a colored dot in the position marker.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**The following functions are not members of this class but are available for use:**

**Function:**

**ISliderControl \*SetupIntSlider(HWND hwnd, int idSlider, int idEdit, int min, int max, int val, int numSegs);**

**Remarks:**

This global function is used for setting up integer sliders. It performs the equivalent of the **GetISlider()**, **SetLimits()**, **SetValue()**, and **LinkToEdit()**.

**Parameters:**

**HWND hwnd**

The handle of the dialog box in which the slider appears.

**int idSlider**

The ID of the slider.

**int idEdit**

The ID of the edit control.

**int min**

The minimum allowable value.

**int max**

The maximum allowable value.

**int val**

The initial value for the spinner.

**int numSegs**

The number of segments to use for the control.

**Return Value:**

A pointer to the slider control.

**Function:**

**ISliderControl \*SetupFloatSlider(HWND hwnd, int idSlider, int idEdit, float min, float max, float val, int numSegs);**

**Remarks:**

This global function is used for setting up floating point sliders. It performs the equivalent of the **GetISlider()**, **SetLimits()**, **SetValue()**, and **LinkToEdit()**.

**Parameters:****HWND hwnd**

The handle of the dialog box in which the slider appears.

**int idSlider**

The ID of the slider.

**int idEdit**

The ID of the edit control.

**float min**

The minimum allowable value.

**float max**

The maximum allowable value.

**float val**

The initial value for the spinner.

**int numSegs**

The number of segments to use for the control.

**Return Value:**

A pointer to the slider control.

**Function:**

**ISliderControl \*SetupUniverseSlider(HWND hwnd, int idSlider, int idEdit, float min, float max, float val, int numSegs);**

**Remarks:**

This global function is used for setting up 'universal' value sliders (**EDITTYPE\_UNIVERSE** -- these display world space units). It performs the equivalent of the **GetISlider()**, **SetLimits()**, **SetValue()**, and **LinkToEdit()**.

**Parameters:****HWND hwnd**

The handle of the dialog box in which the slider appears.

**int idSlider**

The ID of the slider.

**int idEdit**

The ID of the edit control.

**float min**

The minimum allowable value.

**float max**

The maximum allowable value.

**float val**

The initial value for the spinner.

**int numSegs**

The number of segments to use for the control.

**Return Value:**

A pointer to the slider control.

**Function:**

**void SetSliderDragNotify(BOOL onOff);**

**Remarks:**

This function controls whether or not sliders send **CC\_SLIDER\_CHANGE** notifications while the user adjusts them with the mouse.

**Parameters:**

**BOOL onOff**

TRUE to turn on; FALSE to turn off.

**Function:**

**BOOL GetSliderDragNotify();**

**Remarks:**

Returns TRUE if **CC\_SLIDER\_CHANGE** notifications are sent by sliders while the user adjusts them with the mouse; FALSE if they are not sent.

## Class ICurveCtl

See Also: [Class ReferenceTarget](#), [Class ICurve](#), [Class CurvePoint](#), [Class ResourceMakerCallback](#), [Class BitArray](#), [Class Interval](#).

```
class ICurveCtl : public ReferenceTarget
```

### Description:

This class is available in release 3.0 and later only.

This class is an interface to the curve custom control. An example of this control in the 3ds max user interface can be seen in the Color Map section of the Output rollup of a 2D Texture map. Sample code using these APIs is available in `\MAXSDK\SAMPLES\UTILITIES\CCUTIL\CCUTIL.CPP`.

### Methods:

public:

### Prototype:

```
virtual BOOL IsActive()=0;
```

### Remarks:

This method indicates if the dialog box for the control is up or not. Returns TRUE if active; otherwise FALSE.

### Prototype:

```
virtual void SetActive(BOOL sw)=0;
```

### Remarks:

This method is used to bring up or close the dialog box.

### Parameters:

**BOOL sw**

TRUE to open; FALSE to close.

### Prototype:

```
virtual HWND GetHWND()=0;
```

### Remarks:

Returns the window handle of the control.

**Prototype:**

**virtual void SetNumCurves(int iNum, BOOL doUndo=FALSE)=0;**

**Remarks:**

Sets the number of curves used in this control.

**Parameters:**

**int iNum**

The number of curves to use.

**BOOL doUndo=FALSE**

This parameter is available in release 4.0 and later only.

This will cause the function to register an Restore Object if set to TRUE.

**Prototype:**

**virtual int GetNumCurves()=0;**

**Remarks:**

Returns the numbers of curves used by the control.

**Prototype:**

**virtual void SetXRange(float min, float max, BOOL rescaleKeys = TRUE)=0;**

**Remarks:**

Sets the absolute position of the first and last CurvePoints.

**Parameters:**

**float min**

The minimum value.

**float max**

The maximum value.

**BOOL rescaleKeys = TRUE**

This parameter is available in release 4.0 and later only.

This parameter controls whether changing the X range will rescale the keys and key tangents or not. When this is TRUE, the default, the keys and tangents get rescaled to the total X range. Setting this to FALSE allow developers to have move control over the exact placement of keys and tangents when the X

range changes. This also allow developers to set ranges from within the points changed message handler without getting into a loop.

**Prototype:**

**virtual void SetYRange(float min, float max)=0;**

**Remarks:**

This determines the absolute upper and lower Y-constraint. This method only has an effect if the **CC\_CONSTRAIN\_Y** flag is set.

**Parameters:**

**float min**

The minimum value.

**float max**

The maximum value.

**Prototype:**

**virtual Point2 GetXRange()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns the X Range.

**Prototype:**

**virtual Point2 GetYRange()=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Returns the Y Range.

**Prototype:**

**virtual void RegisterResourceMaker(ReferenceMaker \*rmak)=0;**

**Remarks:**

This methods registers a callback object used to handle certain aspects of the control. The callback object will be used for updating of the display button image list and tool tip text for the curve control. It also gets called when the

user executes a Reset or creates a new control.

This registers a reference maker which implements the method **Animatable::GetInterface()** for the id **I\_RESMAKER\_INTERFACE** by returning an object derived from class **ResourceMakerCallback**.

**Parameters:**

**ReferenceMaker \*rmak**

Points to the reference maker which returns the callback object.

**Prototype:**

**virtual BOOL GetZoomValues(float \*h, float \*v)=0;**

**Remarks:**

Returns the current zoom values.

**Parameters:**

**float \*h**

Points to storage for the horizontal zoom value.

**float \*v**

Points to storage for the vertical zoom value.

**Return Value:**

TRUE if valid values were returned; otherwise FALSE.

**Prototype:**

**virtual void SetZoomValues(float h, float v)=0;**

**Remarks:**

Sets the zoom values. To determine the values to use developers should use the CCUtil plug-in, create a pop-up window as big as is appropriate for the final control, adjust the zoom and scroll values interactively, and then simply read out the values from the CCUtil user interface (GetZoom and GetScroll).

**Parameters:**

**float h**

The horizontal value to set.

**float v**

The vertical value to set.

**Prototype:**

**virtual BOOL GetScrollValues(int \*h, int \*v)=0;**

**Remarks:**

Returns the horizontal and vertical scroll values.

**Parameters:**

**int \*h**

Points to storage for the horizontal scroll value.

**int \*v**

Points to storage for the vertical scroll value.

**Return Value:**

TRUE if valid values were returned; otherwise FALSE.

**Prototype:**

**virtual void SetScrollValues(int h, int v)=0;**

**Remarks:**

Sets the scroll values. To determine the values to use developers should use the CCUtil plug-in, create a pop-up window as big as is appropriate for the final control, adjust the zoom and scroll values interactively, and then simply read out the values from the CCUtil user interface (GetZoom and GetScroll).

**Parameters:**

**int h**

The horizontal scroll value to set.

**int v**

The vertical scroll value to set.

**Prototype:**

**virtual void ZoomExtents()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Performs a zoom extents operation to the curve view.

**Prototype:**

**virtual void SetTitle(TCHAR \*str)=0;**

**Remarks:**

Sets the title of the dialog box to the specified string. This is only used if the **CC\_ASPOPUP** is used to create a pop-up dialog.

**Parameters:**

**TCHAR \*str**

The title string to display.

**Prototype:**

**virtual ICurve \*GetControlCurve(int numCurve)=0;**

**Remarks:**

Returns and interface to the specified curve. This interface allows you to set the color of the curve and retrieve the Y value of the curve given an X value.

**Parameters:**

**int numCurve**

The zero based index of the curve.

**Prototype:**

**virtual void SetDisplayMode(BitArray &mode)=0;**

**Remarks:**

Determines which curves are toggled on.

**Parameters:**

**BitArray &mode**

The BitArray to control curve visibility -- one bit for each curve. If the bit is set the curve is toggled on; otherwise it is toggled off.

**Prototype:**

**virtual BitArray GetDisplayMode()=0;**

**Remarks:**

Returns a BitArray which indicates which curves are toggled on or off.

**Prototype:**

**virtual void EnableDraw(BOOL enable)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to turn on/off the display code. It is useful when you are doing lots of changes and don't want the window to continually redraw.

**Parameters:**

**BOOL enable**

TRUE to enable; FALSE to disable.

**Prototype:**

**virtual void SetCCFlags(DWORD flags)=0;**

**Remarks:**

Sets the curve control flags to those passed.

**Parameters:**

**DWORD flags**

See [List of Custom Curve Control Flags](#).

**Prototype:**

**virtual DWORD GetCCFlags()=0;**

**Remarks:**

Returns the curve control flags. See [List of Custom Curve Control Flags](#).

**Prototype:**

**virtual void SetCustomParentWnd(HWND hParent)=0;**

**Remarks:**

Sets the parent window for the control if the control is not a popup window.

**Parameters:**

**HWND hParent**

The window handle of the parent.

**Prototype:**

**virtual void SetMessageSink(HWND hWnd)=0;**

**Remarks:**

Call this method and the following window messages will be sent to the window whose handle is passed. The contents of the **lParam** and **wParam** parameters passed to the window proc are shown.

**WM\_CC\_SEL\_CURVEPT**

Sent when a point is selected or deselected.

**lParam = ICurve \* , LOWORD(wParam)** = The number of points which are selected.

**WM\_CC\_CHANGE\_CURVEPT**

Sent when a point is changed.

**lParam = ICurve \* , LOWORD(wParam)** = The zero based index of the changed point.

**WM\_CC\_CHANGE\_CURVETANGENT**

Sent when a point's in or out tangent is changed.

**lParam = ICurve \* , LOWORD(wParam)** = The zero based index of the changed point,

**HIWORD(wParam)** contains a flag, that indicates if the changed tangent is the in, or out tangent. You can check these flags as **HIWORD(wParam) & IN\_CURVETANGENT\_CHANGED** and **HIWORD(wParam) & OUT\_CURVETANGENT\_CHANGED**, for the in and out tangent respectively.

**WM\_CC\_DEL\_CURVEPT**

Sent when a point is deleted.

**lParam = ICurve \* , LOWORD(wParam)** = The zero based index of the deleted point.

**WM\_CC\_INSERT\_CURVEPT**

This option is available in release 4.0 and later only.

**lParam = ICurve \* , LOWORD(wParam)** = The zero based index of the inserted point.

**Parameters:**

**HWND hWnd**

The handle of the window which will receive the messages.

**Prototype:****virtual void SetCommandMode(int ID)=0;****Remarks:**

Sets the command mode in use by the control.

**Parameters:****int ID**

One of the following values (which correspond directly to toolbar buttons in the UI. See the 2D texture map Output rollup for example):

**CID\_CC\_MOVE\_XY****CID\_CC\_MOVE\_X****CID\_CC\_MOVE\_Y****CID\_CC\_SCALE****CID\_CC\_INSERT\_CORNER****CID\_CC\_INSERT\_BEZIER****Prototype:****virtual int GetCommandMode()=0;****Remarks:**

Returns a value which indicates the current command mode. One of the following values (which correspond directly to toolbar buttons in the UI. See the 2D texture map Output rollup for example):

**CID\_CC\_MOVE\_XY****CID\_CC\_MOVE\_X****CID\_CC\_MOVE\_Y****CID\_CC\_SCALE****CID\_CC\_INSERT\_CORNER****CID\_CC\_INSERT\_BEZIER****Prototype:****virtual void Redraw()=0;****Remarks:**

This method redraws the custom curve control.

**Prototype:**

**virtual Interval GetValidity(TimeValue t)=0;**

**Remarks:**

Returns an **Interval** which reflects the validity of every point of every curve used by the curve control.

**Parameters:**

**TimeValue t**

The time about which the interval is computed.

**Prototype:**

**virtual void Update(TimeValue t, Interval& valid)=0;**

**Remarks:**

Updates the validity interval passed with the overall validity of the curves in the control. This simply does:

**valid &= GetValidity(t);**

**Parameters:**

**TimeValue t**

The time about which the interval is computed.

**Interval& valid**

The interval which is updated.

**Prototype:**

**virtual void SetCurrentXValue(float val)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This sets the position of the vertical line drawn over the graph showing the current X value. See flag **CC\_SHOW\_CURRENTXVAL** in [List of Custom Curve Control Flags](#).

**Parameters:**

**float val**

The value to set.

## Class RightClickMenuManager

See Also: [Class RightClickMenu](#), [Class Interface](#).

class RightClickMenuManager

### Description:

Methods of this class allow a developer to extend the menu presented when a user right clicks on an item in the viewports. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
void Register(RightClickMenu *menu);
```

#### Remarks:

This method is used to register an instance of a class derived from **RightClickMenu**. This allows its methods to be called when the user right clicks on an object in a viewport.

#### Parameters:

**RightClickMenu \*menu**

The menu to set as register.

#### Prototype:

```
void Unregister(RightClickMenu *menu);
```

#### Remarks:

This method is used to un-register a registered right click menu.

#### Parameters:

**RightClickMenu \*menu**

The menu to set as un-register.

#### Prototype:

```
int AddMenu(RightClickMenu *menu, UINT flags, UINT id,  
LPCTSTR data);
```

#### Remarks:

This method is called to add items to the right click menu passed. A string or separator may be added. Strings may be checked or unchecked. They may also be disabled and grayed.

**Parameters:**

**RightClickMenu \*menu**

The menu to add the item to.

**UINT flags**

Describes the item being added. One or more of the following values:

**MF\_CHECKED**

**MF\_UNCHECKED**

**MF\_STRING**

**MF\_DISABLED**

**MF\_GRAYED**

**MF\_SEPARATOR**

**UINT id**

The **id** of the selection. This is the **id** passed to the

**RightClickMenu::Selected()** method when the user makes a selection from the menu.

**LPCTSTR data**

The string to display in the menu (or NULL if adding a separator).

**Return Value:**

Nonzero if the item was added; otherwise 0.

**Prototype:**

**int BeginSubMenu(LPCTSTR name);**

**Remarks:**

This method is available in release 3.0 and later only.

This begins a new sub menu. Items added after this call will appear as sub choices of this one until **EndSubMenu()** is called.

**Parameters:**

**LPCTSTR name**

The name to appear for the sub menu item.

**Return Value:**

Always returns TRUE.

**Prototype:**

```
int EndSubMenu();
```

**Remarks:**

This method is available in release 3.0 and later only.

This ends a sub menu. Items added after this call will appear as they did prior to calling **BeginSubMenu()**.

**Return Value:**

Always returns TRUE.

**Prototype:**

```
void CleanUp();
```

**Remarks:**

This method is available in release 3.0 and later only.

This removes all sub menus from the right click menu.

**Prototype:**

```
void Init(HMENU menu, int startId, int limit, HWND hWnd,  
IPoint2 m);
```

**Remarks:**

This method is used internally.

**Prototype:**

```
void Process(UINT id);
```

**Remarks:**

This method is used internally.

## Class DataEntryMouseProc

See Also: [Class MouseCallBack](#), [Class ViewExp](#), [Class Object](#), [Class Interface](#), [Class Point3](#), [Class Matrix3](#), [Class CreateMouseCallBack](#).

```
class DataEntryMouseProc : public MouseCallBack
```

### Description:

This class is available in release 3.0 and later only.

This mouse proc allows drawing in multiple viewports, offsetting from the construction plane, and orthogonal and angle snapping. This allows developers to support orthogonal snapping and angle snapping on creation (as the Bezier line tool does). If the user presses Shift while dragging the mouse, the point is snapped to the nearest quadrant (ortho snapping). If the Alt key is held, the point is snapped using the setting of the angle snap system.

The typical control flow of this class is that the **OnPointSelected()** method is called every time the user clicks in the viewport, and **OnMouseAbort()** is called when the user right clicks to finish the curve. **RemoveLastPoint()** is called when backspace is pressed, and **OnMouseMove(Point3& p)** is called every time the mouse moves (this lets the developer update the curve continuously).

This class is a sub-class of MouseCallBack, but it can also be used as a CreateMouseCallBack to create curves from the creation panel. To do this you embed a DataEntryMouseProc in a CreateMouseCallBack as show below. Notice the implementation of the virtual member StartNewCreation(). This is a new virtual method on CreateMouseCallBack that tells the system whether the mouse proc is in a state ready to create a new object. This was required, because the "proc" function now always returns "CREATE\_STOP" in order to implement multi-viewport input.

```
class TopCVCurveCreateMouseProc : public  
Em3DDataEntryMouseProc {  
public:  
TopCVCurveCreateMouseProc() :  
Em3DDataEntryMouseProc() {}  
virtual BOOL OnPointSelected();  
virtual void OnMouseMove(Point3& p);  
virtual BOOL AllowAnyViewport();
```

```

virtual void RemoveLastPoint();
virtual int OnMouseAbort();
virtual BOOL PerformRedraw() { return FALSE; }
void SetObj(EditableCVCurve* o) { mpOb = o; }
virtual BOOL StartNewCreation() { return mMouseClicked == 0; }
friend class EditableCVCurve;
private:
EditableCVCurve * mpOb;
};

```

```

class EditableCVCurveCreateCallBack: public
CreateMouseCallBack {
public:
EditableCVCurveCreateCallBack() {}

```

```

virtual int proc( ViewExp* vpt,int msg, int point, int flags, IPoint2
m, Matrix3& mat );
friend class CVBackspaceUser;
friend class EditableCVCurve;
virtual BOOL StartNewCreation() { return
mMouseProc.StartNewCreation(); }
private:
void RemoveLastPoint();
TopCVCurveCreateMouseProc mMouseProc;
};

```

```

int EditableCVCurveCreateCallBack::proc(ViewExp* vpt,int msg,
int point, int flags, IPoint2 m, Matrix3& mat)
{
spTransformMat = &mat;
return mMouseProc.proc(vpt->GetHWnd(), msg, point, flags, m);
}
static EditableCVCurveCreateCallBack nsCreateCB;

```

```

CreateMouseCallBack*
EditableCVCurve::GetCreateMouseCallBack()
{
nsCreateCB.mMouseProc.SetObj(this);
nsCreateCB.mMouseProc.SetParams(hInstance, mpEM, 0);
return(&nsCreateCB);
}

```

#### **Data Members:**

protected:

**Object\* mpObject;**

This a pointer to the object that is using the mouse proc.

**int mMouseClicked;**

The number of clicks (i.e. selected points) the user has entered in creating this object. It is like the "point" parameter to "proc".

**Tab<Point3> mPoints;**

These are the 3D values of the points the user has selected.

**Tab<IPoint2> mClickPoints;**

These are the 2D viewport coordinates the user selected.

**BOOL mLiftOffCP;**

TRUE when in the mode where we lift off the construction plane

#### **Methods:**

public:

#### **Prototype:**

**DataEntryMouseProc(Object\* pObj, int cursor, HINSTANCE hInst);**

#### **Remarks:**

Constructor. The data members are initialized as follows:

**mpObject = pObj;**

**mCursor = cursor;**

**mInstance = hInst;**

**mMouseClicked = 0;**

```
mDoNotDouble = TRUE;  
mLiftOffCP = FALSE;  
mPreviousFlags = 0;
```

**Prototype:**

```
DataEntryMouseProc();
```

**Remarks:**

Constructor. The data members are initialized as follows:

```
mpObject = NULL;  
mpIp = NULL;  
mMouseClicked = 0;  
mDoNotDouble = TRUE;  
mCursor = 0;  
mLiftOffCP = FALSE;  
mPreviousFlags = 0;  
mInstance = 0;
```

**Prototype:**

```
virtual BOOL OnPointSelected();
```

**Remarks:**

Implemented by the Plug-In.

This method is called every time the user clicks in the viewport to enter data. This is the method in NURBS curves, for example, that adds a new CV or point to the curve. The method can query the **mMouseClicked** member to see which point this is in the sequence (like the "point" parameter to traditional MouseCallback classes), and the 3D value of the point can be determined from **mPoints[mMouseClicked]**. The data member **mPoints** contains all the 3D points selected, and **mClickPoints** is a table of the 2d points where the user clicked in the viewport.

**Return Value:**

The return value is used to determine whether the creation should continue or not. If it returns TRUE, more points are selected. If it returns FALSE, then the

creation is done. In the case of NURBS, this is used to implement the feature that asks users if they want to close a curve when they click on the same point where they started the curve. If the answer is yes, this method returns FALSE, otherwise it always return TRUE.

**Default Implementation:**

```
{return TRUE; }
```

**Prototype:**

```
virtual void OnMouseMove(Point3& p);
```

**Remarks:**

Implemented by the Plug-In.

This method is called on every mouse move event.

**Parameters:**

**Point3& p**

The current point in world space of the mouse position.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BOOL AllowAnyViewport();
```

**Remarks:**

Implemented by the Plug-In.

This method tells the system when to allow drawing in mutiple viewports.

**Return Value:**

TRUE to allow drawing between viewports; FALSE to not allow drawing between viewports.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual void RemoveLastPoint();
```

**Remarks:**

Implemented by the Plug-In.

This method is called when the backspace key is pressed. Typically this deletes the last point entered by the user so they may correct its entry.

**Default Implementation:**

```
{
```

**Prototype:**

```
virtual int OnMouseAbort();
```

**Remarks:**

Implemented by the Plug-In.

This method is called when the creation is finished.

**Return Value:**

Return one of the following value to indicate the state of the creation process:

**CREATE\_CONTINUE**

The creation process should continue.

**CREATE\_STOP**

The creation process has terminated normally.

**CREATE\_ABORT**

The creation process has been aborted. The system will delete the created object and node.

**Default Implementation:**

```
{ return CREATE_ABORT; }
```

**Prototype:**

```
virtual BOOL PerformRedraw();
```

**Remarks:**

Implemented by the Plug-In.

This method indicates whether the mouse proc should perform redraws. When used in a **CreateMouseCallBack**, this should return FALSE.

**Return Value:**

TRUE to have the mouse proc perform redraws; otherwise FALSE.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual void SetUseConstructionLine(BOOL useLine) = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called to tell the object to draw offset lines. This is called passing TRUE when the system enters the mode where points are lifted off the construction plane. It is telling the object that it needs to draw a line between the points supplied by **SetConstructionLine(int i, Point3 p)**. It is called passing FALSE when the offset procedure is complete.

To see an example of how this is used, create a NURBS Point curve, and press the Control key while laying down a point. It enters a mode that lets you lift the point off the construction plane, and draws a red dotted line back to the CP to give some visual feedback.

**Parameters:**

**BOOL useLine**

TRUE if the mode is beginning; FALSE if it is ending.

**Prototype:**

```
virtual void SetConstructionLine(int i, Point3 p) = 0;
```

**Remarks:**

These methods need to be implemented to get the offset line drawn

This method is called with  $i==0$  for the start point and with  $i==1$  for the end point.

**Parameters:**

**int i**

The point index: 0 for the start or 1 for the end.

**Point3 p**

The point to draw to or from.

**Prototype:**

**int proc(HWND hwnd, int msg, int point, int flags, IPoint2 m);**

**Remarks:**

Implemented by the System.

This is the method where the developer defines the user / mouse interaction that takes place.

**Parameters:**

**HWND hwnd**

The window handle of the window the user clicked in. This is one of the viewports.

**int msg**

This message describes the type of event that occurred. See [List of Mouse Callback Messages](#).

**int point**

The point number. This is 0 for the first click, 1 for the second, etc.

**int flags**

These flags describe the state of the mouse buttons. See [List of Mouse Callback Flags](#).

**IPoint2 m**

The 2D screen point that the user clicked on.

**Return Value:**

**CREATE\_STOP** is returned.

Note: Notice the implementation of the virtual member

**StartNewCreation()**. This is a virtual method on **CreateMouseButton**

that tells the system whether the mouse proc is in a state ready to create a new object. This was required, because this method now always returns

**CREATE\_STOP** in order to implement the multi-viewport input

**Prototype:**

**void ClearCreationParams();**

**Remarks:**

Implemented by the System.

This method clears the creation parameters. The data members are reset as follows:

```
mMouseClicked = 0;  
mPoints.SetCount(0);  
mClickPoints.SetCount(0);  
mLiftOffCP = FALSE;  
mPreviousFlags = 0;
```

**Prototype:**

```
void SetParams(HINSTANCE hInst, Object* pObj, int cursor);
```

**Remarks:**

Implemented by the System.

This method sets the parameters as follows:

```
mpObject = pObj;  
mCursor = cursor;  
mInstance = hInst;
```

## Class IGraphObjectManager

See Also: [Class IGraphNode](#), [Class IGraphRef](#), [Class Animatable](#).

class IGraphObjectManager

### Description:

This class is available in release 3.0 and later only.

This class essentially represents an instance of a schematic view window and provides methods for adding nodes and node pointers, refreshing the schematic view, accessing filter bits and updating and controlling the various editors within 3ds max in ways that are not surfaced in the general interface.

### Methods:

public:

### Prototype:

```
virtual SvGraphNodeReference AddAnimatable(Animatable  
*anim, Animatable *owner, int id, DWORD flags = 0)=0;
```

### Remarks:

Adds an Animatable to the schematic view. Note that “owner” and “id” are actually arbitrary – they are used to provide context for this Animatable. This means that the plug-in developer can set them to any value. They are not used internally by the schematic view except in the "Sv\*" methods (which the developer can override). So, when you add an Animatable to the schematic view, you would typically add the owner (parent) Animatable as well as the sub-anim id. This allows you to, for example, easily return the name of the object when queried by the SvGetName(...) call (whose default implementation is shown below):

```
TSTR Animatable::SvGetName(IGraphObjectManager *gom,  
IGraphNode *gNode, bool isBeingEdited)  
{  
    Animatable *owner;  
    int subNum;  
    TSTR name;
```

```
owner = gNode->GetOwner();
subNum = gNode->GetID();
name = owner->SubAnimName(subNum);

return name;
}
```

**Parameters:**

**Animatable \*anim**

Points to the animatable to add.

**Animatable \*owner**

Points to the owner of **anim** above (typically).

**int id**

When nodes are added to the schematic view via this method this integer is provided. This value is not used internally by the schematic view. Rather, it is available to implementers of the **Animatable::Sv\*()** methods to aid in identifying the node.

**DWORD flags = 0**

This flag is some combination of the following bit flags. See [List of Schematic View AddAnimatable Flags](#).

**Return Value:**

A SvGraphNodeReference object.

**Prototype:**

```
virtual void PushLevel(Animatable *anim, int id = SV_NO_ID)=0;
```

**Remarks:**

During traversal of the Animatable graph via **SvTraverseAnimGraph(...)**, this method (and **PopLevel()** below) should be called appropriately to maintain an ownership stack. This is required by the schematic view when nodes are added to the graph with the **SV\_DUPLICATE\_INSTANCES** flag set. Note: In 3ds max 3.0, **SV\_DUPLICATE\_INSTANCES** is always on (i.e., the flag is ignored). Because of this, **PushLevel()** and **PopLevel()** should always be called in **SvTraverseAnimGraph(...)**.

See the sample code in **Animatable::SvStdTraverseAnimGraph()** for an example of this.

**Parameters:**

**Animatable \*anim**

This is the Animatable that you are, in all likelihood, about to add to the graph via the **AddAnimatable()** call.

**int id = SV\_NO\_ID**

This is also the same "id" you'd pass into **AddAnimatable()**. The "id" is only required in cases where it's impossible for the schematic view to distinguish between two (or more) children in the tree that have the same Animatable but represent different sub-anim. For example, a box has both its width and height set to the same controller (instanced). In the schematic view, this is still shown in tree form so we need the ID to distinguish between the "width" and "height" children.

**Prototype:**

**virtual void PopLevel()=0;**

**Remarks:**

Pops a level off the animatable ownership stack. See **PushLevel()** above for details.

**Prototype:**

**virtual IGraphRef \*AddReference(IGraphNode \*maker, IGraphNode \*target, SvReferenceType type)=0;**

**Remarks:**

This method adds a reference from the specified "maker" node to the specified "target" node.

**Parameters:**

**IGraphNode \*maker**

Points to the 'maker' node in schematic view.

**IGraphNode \*target**

Points to the 'target' node in schematic view.

**SvReferenceType type**

One of the following enum values:

**REFTYPE\_CHILD**

**REFTYPE\_SUBANIM**

**REFTYPE\_PLUGIN**

**Return Value:**

A pointer to an IGraphRef object.

**Sample Code:**

**SvGraphNodeReference**

**Control::SvTraverseAnimGraph(IGraphObjectManager \*gom,  
Animatable \*owner, int id, DWORD flags)**

**{**

**int i;**

**SvGraphNodeReference nodeRef;**

**SvGraphNodeReference childNodeRef;**

**//**

**// Test filter to see if "Controllers" are active.**

**// Bail out if they're off (being filtered out)...**

**//**

**if (!gom->TestFilter(SV\_FILTER\_CONTROLLERS))**

**return SvGraphNodeReference();**

**//**

**// Push this level in the tree. Note that the sub-anim id is passed**

**// in here because it's possible that the same instance of this control**

**// may exist in multiple tracks of "owner".**

**//**

**gom->PushLevel(this, id);**

**//**

**// Some flags are set here pertaining to the control being added.**

**// Note that the flags are also propagated down the tree**

**// by passing them on to SubAnim(i)-**

**>SvTraverseAnimGraph(gom, this, i, flags);**

```
// SV_DUPLICATE_INSTANCES tells the schematic view not to  
// represent multiple instances with a single node. Instead they  
// are represented by multiple nodes in the schematic view  
// with the "triangle thingy" attached to the side to indicate  
// shared instances. This flag is ignored in R3 because  
// this mode of operation is globally enabled  
// SV_INITIALLY_HIDDEN tells the schematic view that this  
// control's node is to be initially displayed in the closed state.  
// Note that this has no effect if the node already exists  
// in the graph -- it only applies to newly added nodes.  
//
```

```
flags |= SV_DUPLICATE_INSTANCES |  
SV_INITIALLY_HIDDEN;
```

```
//
```

```
// The control is added to the schematic view...
```

```
//
```

```
nodeRef = gom->AddAnimatable(this, owner, id, flags);
```

```
if (nodeRef.stat == SVT_PROCEED)
```

```
{
```

```
//
```

```
// This control's sub-anim's are iterated over...
```

```
//
```

```
for (i = 0; i < NumSubs(); i++)
```

```
{
```

```
if (SubAnim(i))
```

```
{
```

```
//
```

```
// SvTraverseAnimGraph(...) is recursively called to add this sub-  
anim (and all its descendents) to the graph...
```

```
//
```

```
childNodeRef = SubAnim(i)->SvTraverseAnimGraph(gom, this,  
i, flags);
```

```

//
// Now a link (node pointer) is created in the schematic between
// the control (nodeRef.gNode) and its child sub-anim
// (childNodeRef.gNode)....
//
    if (childNodeRef.stat != SVT_DO_NOT_PROCEED)
        gom->AddReference(nodeRef.gNode, childNodeRef.gNode,
REFTYPE_SUBANIM);
    }
}
}

//
// The tree level is popped. Note: a PopLevel() call must always be
// paired with a PushLevel() call!
//
    gom->PopLevel();

    return nodeRef;
}

```

**Prototype:**

**virtual void SvEditSelectedNodeProperties()=0;**

**Remarks:**

Pops up the property editor dialog on the selected nodes in the schematic view.

**Prototype:**

**virtual void SvSelectInMaterialEditor(IGraphNode \*gNode)=0;**

**Remarks:**

Selects the given node in the material editor. Does nothing if "gNode" does not represent a material or map.

**Parameters:**

## **IGraphNode \*gNode**

Points to the node in schematic view.

### **Prototype:**

```
virtual void SvSetCurEditObject(IGraphNode *gNode)=0;
```

### **Remarks:**

Selects the given node in the modifier panel. Does nothing if "gNode" does not represent an object.

### **Parameters:**

## **IGraphNode \*gNode**

Points to the node in schematic view.

### **Prototype:**

```
virtual void SvInvalidateView()=0;
```

### **Remarks:**

Invalidates the schematic view window.

### **Prototype:**

```
virtual void SvInvalidateNode(IGraphNode *gNode)=0;
```

### **Remarks:**

Invalidates a node in the schematic view window.

### **Parameters:**

## **IGraphNode \*gNode**

Points to the node in schematic view.

### **Prototype:**

```
virtual void SvUpdateMaterialEditor()=0;
```

### **Remarks:**

Forces the material editor to update.

### **Prototype:**

```
virtual void SvUpdateModifierPanel()=0;
```

**Remarks:**

Forces the modifier panel to update.

**Prototype:**

**virtual void SetFilter(DWORD mask)=0;**

**Remarks:**

Sets the specified filter bits.

**Parameters:**

**DWORD mask**

See [List of IGraphObjectManager Filter Bits](#).

**Prototype:**

**virtual void ClearFilter(DWORD mask)=0;**

**Remarks:**

Clears the specified filter bits.

**Parameters:**

**DWORD mask**

See [List of IGraphObjectManager Filter Bits](#).

**Prototype:**

**virtual bool TestFilter(DWORD mask)=0;**

**Remarks:**

Tests the specified filter bits. Returns true if set; otherwise false.

**Parameters:**

**DWORD mask**

See [List of IGraphObjectManager Filter Bits](#).

## Class IGraphNode

See Also: [Class IGraphObjectManager](#), [Class Object](#), [Class Animatable](#).

class IGraphNode

### Description:

This class is available in release 3.0 and later only.

This class represents a node in the schematic view graph and provides a few methods for querying information about the node.

### Methods:

public:

### Prototype:

**virtual Animatable \*GetAnim()=0;**

### Remarks:

Returns the Animatable associated with this node.

### Prototype:

**virtual IGraphNode \*GetParentNode()=0;**

### Remarks:

Returns the "primary parent" of this node. Nodes can have multiple parents (objects referencing this node) so this function is not strictly accurate. That said, many nodes have the concept of an owner node, which is what this function returns.

### Prototype:

**virtual bool IsObjectOrModifier()=0;**

### Remarks:

Returns true if this node represents a 3ds max object or modifier; otherwise false.

### Prototype:

**virtual bool IsMaterial()=0;**

**Remarks:**

Returns true if this node represents a 3ds max material or texmap; otherwise false.

**Prototype:**

**virtual Animatable \*GetOwner()=0;**

**Remarks:**

Returns the "owner" of this node. Some nodes have multiple owners. When this is the case, this function returns the "first" owner (the object that first added this node to the schematic view).

**Prototype:**

**virtual int GetID()=0;**

**Remarks:**

Return the ID of this node. When nodes are added to the schematic view (via the **IGraphObjectManager::AddAnimatable(...)** method), an integer is provided. This value is not used internally by the schematic view. Rather, it is available to implementers of the **Animatable::Sv\*()** methods to aid in identifying the node.

## Class IGraphRef

[Class IGraphObjectManager](#), [Class IGraphNode](#), [Class Object](#).

class IGraphRef

### **Description:**

This class is available in release 3.0 and later only.

This class represents a node pointer and, currently, has no methods.

## Class SubClassList

See Also: [Class DllDir](#), [Class ClassDirectory](#), [Class ClassEntry](#), [Class ClassDesc](#), [Class SClassUIInfo](#).

class SubClassList

### Description:

This class is available in release 2.0 and later only.

A sub class list is a table of **ClassEntry** objects that provide information on plug-in classes as well as usage counts for these classes within 3ds max. These sub class lists are organized by super class ID by the **ClassDirectory** class.

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**int FindClass(Class\_ID subClassID);**

#### Remarks:

Returns the index in the list of sub-classes of the class whose Class\_ID is passed.

#### Parameters:

**Class\_ID subClassID**

Specifies which class to return the index of.

#### Prototype:

**ulong SuperID();**

#### Remarks:

Returns the Super class ID corresponding to this sub-class list.

#### Prototype:

**int Count(int accType);**

#### Remarks:

Returns the number of sub-classes that match the specified access type.

#### Parameters:

## **int accType**

One of the following values:

**ACC\_PUBLIC** - public classes

**ACC\_PRIVATE** - non-public classes

**ACC\_ALL** - both of the above (**ACC\_PUBLIC|ACC\_PRIVATE**).

### **Prototype:**

**int FindClass(const TCHAR \*name);**

### **Remarks:**

Returns the index in the list of sub-classes of the class whose **ClassName()** is passed.

### **Parameters:**

**const TCHAR \*name**

Specifies which class to return the index of.

### **Prototype:**

**int GetFirst(int accType);**

### **Remarks:**

Returns the index of the first **ClassDesc** of the specified type in the list of sub-classes.

### **Parameters:**

**int accType**

One of the following values:

**ACC\_PUBLIC** - public classes

**ACC\_PRIVATE** - non-public classes

### **Prototype:**

**int GetNext(int accType);**

### **Remarks:**

Returns the index of the next **ClassDesc** of the specified type (or -1 at the end).

### **Parameters:**

## **int accType**

One of the following values:

**ACC\_PUBLIC** - public classes

**ACC\_PRIVATE** - non-public classes

### **Prototype:**

```
void AddClass(ClassDesc *cld, int dllNum, int index);
```

### **Remarks:**

This method is used internally.

### **Prototype:**

```
void SetUIInfo(SClassUIInfo *uiInfo);
```

### **Remarks:**

This method is available in release 3.0 and later only.

Allows developer to provide some additional information on a superclass.

Currently this includes a color, and a method which draws a representative image in a Windows DC.

### **Parameters:**

**SClassUIInfo \*uiInfo**

Points to the information to set.

### **Prototype:**

```
SClassUIInfo *GetUIInfo();
```

### **Remarks:**

This method is available in release 3.0 and later only.

This method returns additional user interface related information on a given superclass. Returns NULL if no superclass information was assigned.

### **Prototype:**

```
void ReplaceClass(int idx, ClassDesc *cld, int dllNum, int index,  
bool load);
```

### **Remarks:**

This method is available in release 3.0 and later only.

This method is for internal use only.

**Operators:**

**Prototype:**

**int operator==(const SubClassList& lst) const;**

**Remarks:**

Equality operator.

**Prototype:**

**int operator==(const SubClassList &sl);**

**Remarks:**

Equality operator.

**Prototype:**

**ClassEntry& operator[](int i);**

**Remarks:**

Returns a reference to the 'i-th' **ClassEntry** for this super class.

**Parameters:**

**int i**

The index of the entry to return. Valid values begin at an index of 1.

# Class Random

See Also: [Class Interface](#).

class Random

## Description:

This class is available in release 3.0 and later only.

This class defines a Pseudo-random number generator that precisely matches the behavior of the MSVCRT 6.0 random routines. That is to say, for equivalent calls to `::srand()` and `Random::srand()`, both `::rand()` and `Random::rand()` will produce the same results.

The benefit, however, in having this class is that each instantiation is independent, permitting several uncoupled random number generators to be present in the system at once. Moreover, each instantiation is automatically "pre-seeded", making calls to `Random::srand` unnecessary in most uses. Even arrays of `Random` items will operate independently.

In addition to providing the analogues to the "stdlib" functions, this class also provides two useful member functions which can be used to get a random number bounded in either a float or int interval.

Note: To use this class be sure to link to **MAXUTIL.LIB**.

## Sample Code:

```
#include "random.h"
...
Random r;
r.srand(1); // generally unnecessary, seeds generator like stdlib.h's
srand()
r.rand(); // returns a random number a la stdlib.h's rand()
r.get(); // ditto
r.get(16); // returns 1 of 16 possible random numbers from 0 to 15
inclusive
r.get(5,2); // returns 1 of 3 possible random numbers from 2 to 4
inclusive
r.get(1.0f); // returns 1 of "Random::s_rand_max+1" floats, 0 <=
value < 1
```

**r.get(1.0f,0.5f); // as above, but limits the result to 0.5 <= value < 1**  
**Random::s\_rand\_max; // similar to stdlib.h's RAND\_MAX**

...

Note in all "get" cases that contain limits they are specified (max, min). Also be aware that the min value can be attained, but the max cannot. That is to say min <= value < max.

### **Data Members:**

public:

**static const int s\_rand\_max;**

This is akin to the Windows API global **RAND\_MAX**. The constant **RAND\_MAX** is the maximum value that can be returned by the **rand** function. **RAND\_MAX** is defined as the value 0x7fff.

### **Methods:**

public:

### **Prototype:**

**Random();**

### **Remarks:**

The constructor will automatically initialize the seed.

### **Prototype:**

**void srand(unsigned int seed = 1);**

### **Remarks:**

This method is akin to the global **srand()** function. From the Windows API documentation:

The **srand** function sets the starting point for generating a series of pseudorandom integers.

### **Parameters:**

**unsigned int seed = 1**

To reinitialize the generator, use 1 as the seed argument. Any other value for seed sets the generator to a random starting point. **rand** retrieves the pseudorandom numbers that are generated. Calling **rand** before any call to

**srand** generates the same sequence as calling **srand** with seed passed as 1.

**Prototype:**

```
int rand();
```

**Remarks:**

This method is akin to the global **rand()** function. From the Windows API documentation:

The **rand** function returns a pseudorandom integer in the range 0 to **RAND\_MAX**. Use the **srand** function to seed the pseudorandom-number generator before calling **rand**.

**Prototype:**

```
inline int get(int max_exclusive = s_rand_max+1, int min_inclusive = 0);
```

**Remarks:**

Returns a random number in the half-open interval [min, max) such that  $r = \text{get}(\text{max}, \text{min}) := \text{min} \leq r < \text{max}$ . Note that max is the first arg, and min is the second, permitting one to select, for example, an int in  $[0,5) = [0,4]$  with "get(5)". With no arguments, **Random::get()** is equivalent to **Random::rand()**.

**Parameters:**

```
int max_exclusive = s_rand_max+1
```

The maximum value.

```
int min_inclusive = 0
```

The minimum value.

**Prototype:**

```
inline float getf(float max_exclusive = 1.0f, float min_inclusive = 0.0f);
```

**Remarks:**

Returns a random number in the half-open interval [min, max) such that  $r = \text{get}(\text{max}, \text{min}) := \text{min} \leq r < \text{max}$ . Note that max is the first arg, and min is the second, permitting one to select, for example, a float in  $[0.0, 5.0)$  with

"getf(5)". With no arguments, **Random::getf()** returns a float in [0.0, 1.0).

**Parameters:**

**float max\_exclusive = 1.0f**

The maximum value.

**float min\_inclusive = 0.0f**

The minimum value.

# Class MacroRecorder

See Also: [Class ClassDesc](#), [Class INode](#), [Class IParamBlock](#), [Class IParamBlock2](#), [Class Matrix3](#), [Class ReferenceTarget](#).

```
class MacroRecorder : public BaseInterfaceServer
```

## Description:

This class is available in release 3.0 and later only.

This class provides various methods to emit pieces of script to the Macro Recorder. There are also methods to specify the nature of the scripts generated by the user operating 3ds max. Additionally there are methods to enable or disable the recording of scripts.

This class may be used by plug-ins but is also used internally as part of the 3ds max Macro Recorder. Inside key areas of 3ds max macro scripts are emitted corresponding to the operation being performed. For example when a call goes to a parameter block to set a value 3ds max internally call this classes method **ParamBlock2SetValue(...)** to emit script to record the change. Thus, many operations a plug-in performs are recorded automatically. There are however operations a plug-in can perform which won't be recorded automatically. In these cases methods of this class may be used to emit script to record these operations. Several of the functions in this class use an ellipsis argument (...). This ellipsis is the var-args style of passing information to a method. This allows a developer to pass a variable number of values to the method. These are typically a set of tag values followed by some C++ types. The sample code shown with many of the methods shows how this is done. For a starting point for more general information on variable argument lists see **va\_arg** in the Window help file.

Developers use the following global instance of this class to call these methods:

```
MacroRecorder *macroRecorder;
```

## Methods:

```
public:
```

## Prototype:

```
virtual void SetProperty(ReferenceTarget* targ, TCHAR*  
prop_name, BYTE type, ...)=0;
```

## Remarks:

This provides a simple way to emit a properly assignment script. An example of this type of script is shown below:

```
$sphere01.radius = 50
```

**Parameters:**

**ReferenceTarget\* targ**

Points to the object whose property is changing.

**TCHAR\* prop\_name**

The string which is the name of the property. This is the fixed machine-parsable name.

**BYTE type**

One of the type tags from [List of Macro Recorder Value Types](#).

...

This ellipsis is the var-args style of passing information to a method. This allows a developer to pass a variable number of values to a method. These are typically a set of 'tags' followed by some C++ data types.

**Prototype:**

```
virtual void SetSelProperty(TCHAR* prop_name, BYTE type,  
...)=0;
```

**Remarks:**

This provides a simple way to emit a properly assignment script for the current selection set.

**Parameters:**

**TCHAR\* prop\_name**

The name of the property to set.

**BYTE type**

One of the type tags from [List of Macro Recorder Value Types](#).

...

This ellipsis is the var-args style of passing information to a method. This allows a developer to pass a variable number of values to a method. These are typically a set of 'tags' followed by some C++ types.

**Sample Code:**

```
macroRecorder->SetSelProperty(_T("material"), mr_reftarg,
```

**(Mtl\*)dropThis);**

**Prototype:**

**virtual void FunctionCall(TCHAR\* op\_name, int arg\_count, int keyarg\_count, ...)=0;**

**Remarks:**

This method is used to build a MAXScript function call. In the general case, such a call may have positional arguments followed by keyword arguments.

**Parameters:**

**TCHAR\* op\_name**

The name of the function to call.

**int arg\_count**

The number of positional arguments in the varargs section.

**int keyarg\_count**

The number of keyword arguments in the varargs section.

...

This ellipsis is the var-args style of passing information to a method. This allows a developer to pass a variable number of values to a method. These are typically a set of 'tags' followed by some C++ types.

See [List of Macro Recorder Value Types](#) for a list of potential tags and arguments.

**Sample Code:**

```
macroRecorder->FunctionCall(_T("addModifier"), 2, 1,  
mr_reftarg, this, mr_create, mod->ClassID(), mod->  
SuperClassID(), 0, _T("before"), mr_int, before);
```

This generates an addModifier() function call, such as:

```
addModifier $foo (bend()) before:3
```

The call has 2 positional arguments and 1 keyword argument (hence the **2, 1**). The first positional is given as **mr\_reftarg, this** which refers to the current node, the second positional is given as **mr\_create, mod->ClassID(), mod->SuperClassID(), 0** which causes a 0-argument constructor to be emitted for the modifier, and finally the single keyword argument is given as **\_T("before"), mr\_int, before** which is the keyword name followed by the

arg value.

**Prototype:**

**virtual void ScriptString(TCHAR\* s)=0;**

**Remarks:**

Emits a piece of macro script as a literal string. To understand when this is used consider the following example. Say you have a button in your user interface which does a certain thing but there is no way using the other macro recorder calls of constructing the piece of script that you need emitted. For instance the button may invoke a for loop. In such a case you can use this method to emit a macro script string which does a for loop. This is a string, just as you would type it into the MAXScript Listener.

**Parameters:**

**TCHAR\* s**

The string to emit.

**Prototype:**

**virtual void Assign(TCHAR\* var\_name, BYTE type, ...)=0;**

**Remarks:**

This method is called to emit script to record the assignment to a MAXScript variable.

**Parameters:**

**TCHAR\* var\_name**

The variable name.

**BYTE type**

One of the type tags from [List of Macro Recorder Value Types](#).

...

This ellipsis is the var-args style of passing information to a method. This allows a developer to pass a variable number of values to a method. These are typically a set of 'tags' followed by some C++ types.

**Prototype:**

**virtual void SetCopy(ReferenceTarget\* to\_copy)=0;**

**Remarks:**

This method is used to signal that an `mr_reftarg` argument in the currently accumulating script should be emitted as a copy. For example, when maps or materials are dragged onto sub-map/mtl buttons in the material editor, an instance/copy requester dialog is presented and depending on the choice, either a direct assignment or assignment of a copy is appropriate:

```
meditMaterials[4].materialList[2] = meditMaterials[11] --  
instance
```

```
meditMaterials[4].materialList[2] = copy meditMaterials[11] --  
copy
```

The actual assignment script is set up using a **macroRecorder-> SetProperty()** call with the dropped map/material supplied as an **mr\_reftarg** argument. In situations where the copy/instance status is known, you can emit directly an **mr\_funcall** argument for the copy, but there may be situations in which this choice is decided in some piece of remote code or control flow and so you can use this method to condition the emitted script to apply a 'copy' call.

**Parameters:**

**ReferenceTarget\* to\_copy**

The ReferenceTarget\* object which should be copied.

**Prototype:**

```
virtual void Cancel()=0;
```

**Remarks:**

This cancels and clears the currently accumulating script. This would be used for example, if the operation that is being accumulated can be canceled by the user, such as right-clicking out of a transform or a geometry node create.

There are calls to **macroRecorder()->Cancel()** in the **MOUSE\_ABORT** processing in the default creation manager.

**Prototype:**

```
virtual void EmitScript()=0;
```

**Remarks:**

This signals the completion of an accumulating script, causing it to be frozen in the recorder pane and any new calls that might have been folded into the current script will cause a new one to be started. For example, when you drag the time slider, the sliderTime assignment script accumulates the changes, but when you let go of the mouse button, an EmitScript() is called, so that subsequent drags will start a separate script fragment. Same with interactive transforms and node creation.

**Prototype:**

**virtual TSTR GetSubMtlPropName(Mtl\* m, int i)=0;**

**Remarks:**

Returns the property name of the 'i-th' sub-material of the specified material.

**Parameters:**

**Mtl\* m**

The material whose 'i-th' sub-material property name is returned.

**int i**

The zero based index of the sub-material.

**Prototype:**

**virtual TSTR GetSubTexmapPropName(MtlBase\* m, int i)=0;**

**Remarks:**

Returns the property name of the 'i-th' sub-texmap of the specified material.

**Parameters:**

**MtlBase\* m**

The material or texmap whose 'i-th' sub-texmap property name is returned.

**int i**

The zero based index of the sub-texmap.

**Prototype:**

**virtual void Enable()=0;**

**Remarks:**

Enables the Macro Recorder. This call is 'nestable', i.e. it uses a use counter internally so recursive or nested code can manage local enables and disable

states.

**Prototype:**

**virtual void Disable()=0;**

**Remarks:**

Disables the Macro Recorder. This allows the developer to disable the automatic macro recording.

**Prototype:**

**virtual BOOL Enabled()=0;**

**Remarks:**

Returns TRUE if the Macro Recorder is enabled (via **Enable()** above); otherwise FALSE.

**Prototype:**

**virtual BOOL MasterEnable()=0;**

**Remarks:**

In the MAXScript Listener Window Macro Recorder pulldown menu choice is an option to enable or disable the Macro Recorder. This method corresponds to that state. It returns TRUE if enabled; FALSE if disabled.

**Prototype:**

**virtual void MasterEnable(BOOL onOff)=0;**

**Remarks:**

In the MAXScript Listener Window Macro Recorder pulldown menu choice is an option to enable or disable the Macro Recorder. This method sets this state.

**Parameters:**

**BOOL onOff**

TRUE for enabled; FALSE for disabled.

**Prototype:**

**virtual BOOL ShowCommandPanelSwitch()=0;**

**Remarks:**

Returns TRUE if code is emitted when command panels are changed; FALSE if code is not emitted.

**Prototype:**

**virtual void ShowCommandPanelSwitch(BOOL onOff)=0;**

**Remarks:**

Determines if the macro recorder will emit script for command panel mode changes.

**Parameters:**

**BOOL onOff**

TRUE to record command panel changes; FALSE to ignore them.

**Prototype:**

**virtual BOOL ShowToolSelections()=0;**

**Remarks:**

Returns TRUE if the macro recorder will emit script for 3ds max toolbar tool selections; otherwise FALSE.

**Prototype:**

**virtual void ShowToolSelections(BOOL onOff)=0;**

**Remarks:**

Determines if the macro recorder will emit script for 3ds max toolbar selections.

**Parameters:**

**BOOL onOff**

TRUE to record toolbar selections; FALSE to ignore them.

**Prototype:**

**virtual BOOL ShowMenuSelections()=0;**

**Remarks:**

Returns TRUE if the macro recorder will emit script for 3ds max menu

selections; otherwise FALSE.

**Prototype:**

**virtual void ShowMenuSelections(BOOL onOff)=0;**

**Remarks:**

Determines if the macro recorder will emit script for 3ds max menu selections.

**Parameters:**

**BOOL onOff**

TRUE to record menu selections; FALSE to ignore them.

**Prototype:**

**virtual BOOL EmitAbsoluteSceneNames()=0;**

**Remarks:**

Returns TRUE if specific node names are used in the generated code; FALSE if the current selection is used.

**Prototype:**

**virtual void EmitAbsoluteSceneNames(BOOL onOff)=0;**

**Remarks:**

This controls whether the code generated refers to the exact node names being operated on or simply the current selection.

**Parameters:**

**BOOL onOff**

TRUE to record absolute scene names; FALSE to use the selection set.

**Prototype:**

**virtual BOOL EmitAbsoluteSubObjects()=0;**

**Remarks:**

Returns TRUE if recording absolute sub-object numbers; FALSE if using the selection set.

**Prototype:**

**virtual void EmitAbsoluteSubObjects(BOOL onOff)=0;**

**Remarks:**

Determines if code generated is relative to the current sub-object selection state or if explicit sub-object numbers are generated.

**Parameters:**

**BOOL onOff**

TRUE to record explicit, absolute sub-object numbers; FALSE to use the selection set.

**Prototype:**

**virtual BOOL EmitAbsoluteTransforms()=0;**

**Remarks:**

Returns TRUE if code is generated using absolute transform assignments; FALSE if relative transforms operations are generated.

**Prototype:**

**virtual void EmitAbsoluteTransforms(BOOL onOff)=0;**

**Remarks:**

Sets if code is generated using absolute transform assignments.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL EmitExplicitCoordinates()=0;**

**Remarks:**

Returns TRUE if the macro recorder emits explicit coordinate contexts; otherwise FALSE. This determines whether explicit coordinate contexts are emitted in the generated script. For example, when you do a move() call or a .pos assign in MAXScript, the coordinates are interpreted with respect to the current active coordinate system, in a way similar to interactive transforms in the 3ds max UI. You set up the current system using a coordsys context, like this:

**coordsys parent move \$foo [10,0,0]**

which would move the object 'foo' [10,0,0] in its parent's coordinate system. If you just say

**move \$foo [10,0,0]**

it will work relative to the current active coordinate system. So, when this method returns TRUE 3ds max always sticks the coordsys context prefix onto a generated transform script, as set by the user in the coordinate system dropdown.

**Prototype:**

**virtual void EmitExplicitCoordinates(BOOL onOff)=0;**

**Remarks:**

Sets if the code is generated using explicit coordinate contexts. See **EmitExplicitCoordinates()** above.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL BeginCreate(ClassDesc\* cd, int flags=0)=0;**

**Remarks:**

You would use this method if implementing a custom creation manager. This method starts a special accumulation 'mode' in which certain other Macro Recorder calls are treated specially until the next **EmitScript()**.

**BeginCreate()** effectively signals the start of a scene node creation and enters a mode in which calls to **SetNodeTM()**, **SetProperty()**,

**ParamBlockXSetValue()** and **SetSelProperty()** all generate keyword parameters to the current constructor, rather than emitting stand-alone property assignment scripts. Outside the 'create' mode, a call to **SetNodeTM()** would generate something like:

**move \$foo [10,0,0]**

but when in the mode would add a pos: argument to the constructor:

**sphere radius:20 pos:[10,0,0]**

**Parameters:****ClassDesc\* cd**

Points to the class descriptor for the plug-in.

**Return Value:**

This tells you whether MAXScript successfully entered the BeginCreate mode. It will fail if MAXScript can't create the object described by the ClassDesc, so you might use it in some generic situations to decided whether to call the closing EmitScript().

**Prototype:****virtual void SetNodeTM(INode\* n, Matrix3 m)=0;****Remarks:**

This method is for internal use only.

**Prototype:****virtual void ParamBlockSetValue(ParamBlock\* pb, int i, BYTE type, ...)=0;****Remarks:**

This method is for internal use only.

**Prototype:****virtual void ParamBlock2SetValue(ParamBlock2\* pb, int i, int tabIndex, ...)=0;****Remarks:**

This method is for internal use only.

**Prototype:****virtual void ParamBlock2SetCount(IParamBlock2\* pb, int i, int n)=0;****Remarks:**

This method is for internal use only.

**Prototype:**

**virtual BOOL BeginSelectNode()=0;**

**Remarks:**

This method is for internal use only.

**Prototype:**

**virtual void Select(INode\*)=0;**

**Remarks:**

This method is for internal use only.

**Prototype:**

**virtual void DeSelect(INode\*)=0;**

**Remarks:**

This method is for internal use only.

**Prototype:**

**virtual void MAXCommand(int com)=0;**

**Remarks:**

This method is for internal use only.

**The following global function is not a part of this class:**

**Function:**

**void InitMacroRecorder();**

**Remarks:**

This global function is used internally to initialize the macro recorder and should not be called by plug-in developers.

## Class ILayerManager

See Also: [Class ReferenceTarget](#), [Class ILayer](#).

class ILayerManager : public ReferenceTarget

### Description:

This class is available in release 3.0 and later only.

This class is an interface to the layer manager. Note that some methods of this class are not functional in 3ds max (only in 3D Studio VIZ).

### Data Members:

public:

**static const SClass\_ID kLayerManagerSuperClassID;**

The super class ID of the layer manager interface.

### Methods:

public:

### Prototype:

**virtual bool AddLayer(ILayer \*layer)=0;**

### Remarks:

Adds the specified layer.

### Parameters:

**ILayer \*layer**

Points to the layer to add.

### Return Value:

Returns true if the layer was added; false if not.

### Prototype:

**virtual ILayer \* CreateLayer(void) = 0;**

### Remarks:

Creates a layer. The name is based on the incremented layer count.

### Prototype:

**virtual BOOL DeleteLayer(const TSTR & name) = 0;**

**Remarks:**

Deletes the layer whose name is passed. Note: This method does nothing in 3ds max.

**Parameters:**

**const TSTR &name**

The name for the layer.

**Return Value:**

TRUE if the layer was deleted, otherwise FALSE.

**Prototype:**

**virtual void SetCurrentLayer(const TSTR &name)=0;**

**Remarks:**

Sets the layer whose name is passed as current.

**Parameters:**

**const TSTR &name**

The name for the new current layer.

**Prototype:**

**virtual void SetCurrentLayer()=0;**

**Remarks:**

Sets the current layer based on the selection set (the common layer).

**Prototype:**

**virtual ILayer \*GetCurrentLayer() const=0;**

**Remarks:**

Returns an interface to the current layer.

**Prototype:**

**virtual void EditLayer(const TSTR &name)=0;**

**Remarks:**

Edits the layer whose name is passed. Note: This method does nothing in 3ds max.

**Parameters:**

**const TSTR &name**

The name of the layer to edit.

**Prototype:**

**virtual void DoLayerPropDialog(HWND hWnd)=0;**

**Remarks:**

Brings up the layer property dialog. Note: This method does nothing in 3ds max.

**Parameters:**

**HWND hWnd**

The parent window handle.

**Prototype:**

**virtual LayerIterator \*MakeIterator()=0;**

**Remarks:**

This method is for internal use in VIZ.

**Prototype:**

**virtual ConstLayerIterator \*MakeConstIterator() const=0;**

**Remarks:**

This method is for internal use in VIZ.

**Prototype:**

**virtual int GetLayerCount()=0;**

**Remarks:**

Returns the number of layers.

**Prototype:**

**virtual ILayer \*GetLayer(const TSTR &name) const=0;**

**Remarks:**

Returns a pointer to a layer interface for the named layer.

**Parameters:**

**const TSTR &name**

The name of the layer to get.

**Prototype:**

**virtual void DoLayerSelDialog(HWND hWnd)=0;**

**Remarks:**

Brings up the select layer dialog. Note: This method does nothing in 3ds max.

**Parameters:**

**HWND hWnd**

The parent window handle.

**Prototype:**

**virtual void SetupToolList(HWND hWnd)=0;**

**Remarks:**

Sets up the toolbar list. Note: This method does nothing in 3ds max.

**Parameters:**

**HWND hWnd**

The parent window handle.

**Prototype:**

**virtual void ExtendMenu(HMENU hMenu, bool geometry = true, bool grid = false)=0;**

**Remarks:**

Extends the right click menu. Note: This method does nothing in 3ds max.

**Parameters:**

**HMENU hMenu**

The handle of the menu to append to.

**bool geometry = true**

Use true to add the geometry commands; false to not add them.

**bool grid = false**

Use true to add the grid commands; false to not add them.

**Prototype:**

**virtual ILayer \*GetRootLayer() const=0;**

**Remarks:**

Returns an interface to the 0 layer.

**Prototype:**

**virtual void Reset(BOOL fileReset = FALSE)=0;**

**Remarks:**

Resets the layer manager.

**Parameters:**

**BOOL fileReset = FALSE**

This parameter is ignored.

**Prototype:**

**virtual void SelectObjectsByLayer(HWND hWnd) = 0;**

**Remarks:**

This method will bring up the select objects by layer dialog.

**Parameters:**

**HWND hWnd**

The handle to the parent window.

# Class IKMasterControl

See Also: [Class ReferenceTarget](#), [Class IKSlaveControl](#), [Class Control](#).

```
class IKMasterControl : public ReferenceTarget
```

## Description:

This class is available in release 2.0 and later only.

The IK Controller requires that you use the Bones system. When you create the bones, a slave IK controller is assigned to each bone. All of the slave IK controllers in a single hierarchy are, in turn, controlled by a master IK controller. This class provides two methods to work with the master controller. To get an interface to this class call **GetInterface(I\_MASTER);** on the controller in question. If the return value is non-NULL you can cast the pointer to an instance of this class.

For an example the use of this class see

**\\MAXSDK\\SAMPLES\\OBJECTS\\BONES.CPP.**

## Methods:

### Prototype:

```
virtual void AddSlaveNode(INode *node)=0;
```

### Remarks:

Adds the specified node to the list of slave nodes maintained.

### Parameters:

**INode \*node**

The node to add.

### Prototype:

```
virtual void *GetMasterBase()=0;
```

### Remarks:

Returns a pointer to the IK master object.

### Prototype:

```
virtual void SetPosThresh(float t)=0;
```

### Remarks:

This method is available in release 3.0 and later only.

Sets the position threshold. If the UI for the master controller is not visible while this method is called, the screen is not redrawn to reflect the changes. After calling this method you should therefore call

**Interface::RedrawViews()**.

**Parameters:**

**float t**

The value to set.

**Prototype:**

**virtual void SetRotThresh(float t)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the rotation threshold. If the UI for the master controller is not visible while this method is called, the screen is not redrawn to reflect the changes. After calling this method you should therefore call

**Interface::RedrawViews()**.

**Parameters:**

**float t**

The value to set.

**Prototype:**

**virtual void SetIterations(int i)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the iterations value. If the UI for the master controller is not visible while this method is called, the screen is not redrawn to reflect the changes. After calling this method you should therefore call **Interface::RedrawViews()**.

**Parameters:**

**int i**

The value to set.

**Prototype:**

**virtual void SetStartTime(TimeValue s)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the start time. If the UI for the master controller is not visible while this method is called, the screen is not redrawn to reflect the changes. After calling this method you should therefore call **Interface::RedrawViews()**.

**Parameters:**

**TimeValue s**

The time to set.

**Prototype:**

**virtual void SetEndTime(TimeValue e)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the end time. If the UI for the master controller is not visible while this method is called, the screen is not redrawn to reflect the changes. After calling this method you should therefore call **Interface::RedrawViews()**.

**Parameters:**

**TimeValue e**

The time to set.

**Prototype:**

**virtual float GetPosThresh()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the position threshold.

**Prototype:**

**virtual float GetRotThresh()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the rotation threshold.

**Prototype:**

**virtual int GetIterations()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the iterations setting.

**Prototype:**

**virtual TimeValue GetStartTime()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the start time.

**Prototype:**

**virtual TimeValue GetEndTime()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the end time.

**The following function is not part of this class but is available for use:**

**Function:**

**IKMasterControl \*CreateIKMasterControl();**

**Remarks:**

This global function creates a new IK master controller.

## Class MeshDelta

See Also: [Class Mesh](#), [Class FaceChange](#), [Class FaceRemap](#), [Class FaceSmooth](#), [Class VertMove](#), [Class UVVertSet](#), [Class MapDelta](#), [Class VDataDelta](#), [Class AdjEdgeList](#), [Class AdjFaceList](#), [Class MeshChamferData](#), [Template Class Tab](#), [Class BitArray](#).

```
class MeshDelta : public BaseInterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This class represents the notion of a mesh edit.

This is an SDK class that represent some kind of change to a mesh. This "delta" can include topological, geometric, map, and/or selection changes. Most standard mesh "edits" available in the Editable Mesh or Edit Mesh interface are available through the MeshDelta SDK, giving developers a powerful way to manipulate meshes while not having to "sweat the details" of maintaining maps to match the mesh changes, updating edge selections, etc.

The MeshDelta members and methods make use of a number of mesh-related classes, including [Class FaceChange](#), [Class FaceRemap](#), [Class FaceSmooth](#), [Class VertMove](#), [Class UVVertSet](#), [Class MapDelta](#), [Class VDataDelta](#), [Class AdjEdgeList](#), [Class AdjFaceList](#), [Class MeshChamferData](#).

While we often talk about the characteristics of the "input mesh" that a MeshDelta is based on, all MeshDeltas should be able to cope with any mesh.

Note: You must **#include "MESHDLIB.H"** to use this class as it's not included by default by **MAX.H**.

### Methods Groups:

The hyperlinks below take you to the start of groups of related methods within the class:

[Initialization & Cleanup](#)

[Mesh Interaction](#)

[Composition and operators](#)

[Characteristics](#)

[Lookup Table Methods](#)

[Basic Operations](#)

[Advanced additive operations](#)

[Advanced operations](#)  
[I/O, Debugging](#)

**Data Members:**

public:

**DWORD vnum**

The expected number of vertices in the input mesh.

**DWORD fnum;**

The expected number of faces in the input mesh.

**Tab<VertMove> vMove;**

This data member stores movements of input vertices. Each VertMove consists of a vertex ID indicating which vertex should be moved and a Point3 offset in object space. VertMoves are stored in vertex ID order, and there is never more than one VertMove per original vertex.

**Tab<VertMove> vClone;**

If the vClone[i] record has a vid of UNDEFINED, it's considered a "create", and the coordinates of vClone[i].p are considered to be in object space. If vClone[i].vid is not undefined, it's the index of a vertex "original" in the input mesh, and vClone[i].p is treated as an offset from that vertex. If the vertex is not present in the input mesh, i.e. vClone[i].vid >= mesh::numVerts, the clone will not be created in the output. All creates and clones are stored in the order created..

**BitArray vDelete;**

This data member stores deletions of vertices in the input mesh. vDelete's size is vnum.

**Tab<FaceCreate> fCreate;**

This data member stores faces newly created as part of the MeshDelta. These are stored in the order created.

**Tab<FaceRemap> fRemap;**

This data member stores changes in which vertices are used by existing faces. See class FaceRemap for more information. These are stored in original face order, and there is never more than one per original face.

**Tab<FaceChange> fChange;**

This data member stores changes in input face characteristics, such as material ID, edge visibility, and face hiding. See class FaceChange for more

information. These are stored in original face order, and there is never more than one per original face.

**Tab<FaceSmooth> fSmooth;**

This data member stores changes in input face smoothing groups. See class FaceSmooth for more information. These are stored in original face order, and there is never more than one per original face.

**BitArray fDelete;**

This data member stores deletions of faces in the input mesh. The size of this BitArray is fnum.

**BitArray vsel;**

This data member stores the vertex selection of the output mesh.

**BitArray esel;**

This data member stores the edge selection of the output mesh. As with class Mesh's edgeSel data member, this information is indexed by side of face: esel[ff\*3+k] is the edge selection for face ff, side k.

**BitArray fsel;**

This data member stores the face selection of the output mesh.

**BitArray vhide;**

This data member stores the vertex hide information of the output mesh.

**MapDelta \*map;**

Points to an array of MapDeltas which maintain any relevant changes to the various map channels. The size of this array is always equal to mapSupport.GetSize().

**BitArray mapSupport;**

Indicates which maps are supported by this MeshDelta.

**VDataDelta \*vd;**

Points to an array of VDataDeltas which maintain any relevant changes to the various vertex data channels. The size of this array is always equal to vdSupport.GetSize().

**BitArray vdSupport;**

Indicates which vertex data channels are supported by this MeshDelta.

**Methods:**

public:

## Initialization & Cleanup

### Prototype:

**MeshDelta();**

### Remarks:

Constructor. Initializes the MeshDelta with NULL pointers and 0's for input mesh size.

### Prototype:

**MeshDelta(const Mesh &m);**

### Remarks:

Constructor. Initializes the MeshDelta to be based on the mesh given. MapDeltas and VDataDeltas are allocated as appropriate, and vnum and fnum are set.

### Prototype:

**~MeshDelta();**

### Remarks:

Destructor. Frees all allocated memory, including the MapDeltas.

### Prototype:

**void InitToMesh(const Mesh &m);**

### Remarks:

Initializes the MeshDelta to the mesh given, setting map and vdata support as appropriate. Does NOT clear out existing changes.

### Parameters:

**const Mesh &m**

The mesh to init from.

### Prototype:

**void ClearAllOps();**

### Remarks:

Clears out all existing mesh changes. Zeroes all the vCreate, vMove, etc

arrays, as well as those in the active MapDeltas. Does not clear memory.

**Prototype:**

**void SetInFNum(int nface);**

**Remarks:**

Sets the number of faces in the input mesh. NOTE that if nface is less than the current fnum, the data relating to the extra faces will be lost. (That is, if one of your face remaps is applied to face 32, and you SetInFNum to 30, that face remap will be lost, and will not be recovered if you later SetInFNum to 35.) It is NOT necessary to call this method before applying this MeshDelta to a smaller than expected Mesh.

**Parameters:**

**int nface**

The number of faces expected from the input mesh.

**Prototype:**

**void SetInVNum(int nv);**

**Remarks:**

Sets the number of vertices in the input mesh. NOTE that if nv is less than the current vnum, the data relating to the extra vertices will be lost. (That is, if one of your vertex moves is applied to vertex 32, and you SetInVNum to 30, that vertex move will be lost, and will not be recovered if you later SetInVNum to 35.) It is NOT necessary to call this method before applying this MeshDelta to a smaller than expected Mesh.

**Parameters:**

**int nv**

The number of vertices expected from the input mesh.

**Prototype:**

**void SetMapNum(int num, bool keep=TRUE);**

**Remarks:**

Sets the number of map channels used by the MeshDelta – allocates the "map" array.

**Parameters:****int num**

The number of maps to allocate.

**bool keep=TRUE**

If TRUE any previous maps are kept; otherwise they are discarded.

**Prototype:****int GetMapNum();****Remarks:**

Gets the number of map channels in the MeshDelta – equivalent to mapSupport.GetSize().

**Prototype:****MapDelta & Map(int mp);****Remarks:**

Data accessor - gets the MapDelta for the specified map channel. Since in 4.0 we now have "hidden map channels" which are accessed by negative indices (-1 for MAP\_SHADING, for example), data accessor methods like this one should be used instead of the actual arrays. (Hidden map channels are stored in a new private data member, not as part of the public map array.)

**Prototype:****bool getMapSupport (int mp);****Remarks:**

Indicates whether the specified map channel is supported by this MeshDelta. Since in 4.0 we now have "hidden map channels" which are accessed by negative indices (-1 for MAP\_SHADING, for example), data accessor methods like this one should be used instead of the actual data members. (Hidden map channel support information is stored in a new private data member, not as part of the public mapSupport BitArray.)

**Prototype:****void setMapSupport (int mp, bool val=true);**

**Remarks:**

Sets map support in this MeshDelta for the specified map channel. Since in 4.0 we now have "hidden map channels" which are accessed by negative indices (-1 for MAP\_SHADING, for example), data accessor methods like this one should be used instead of the actual data members. (Hidden map channel support information is stored in a new private data member, not as part of the public mapSupport BitArray.)

**Prototype:**

```
bool hasMapSupport ();
```

**Remarks:**

Indicates whether any map channel is supported by this MeshDelta.

**Prototype:**

```
void SetVDataNum(int num, bool keep=TRUE);
```

**Remarks:**

Sets the number of vertex data channels used by the MeshDelta – allocates the "vd" array.

**Parameters:**

**int num**

The number of vertex data channels to allocate.

**bool keep=TRUE**

If TRUE any previous vertex data channels are kept; otherwise they are discarded.

**Prototype:**

```
int GetVDataNum();
```

**Remarks:**

Gets the number of vertex data channels in the MeshDelta – equivalent to vdSupport.GetSize().

**Prototype:**

```
void AddVertexColors();
```

**Remarks:**

Activates the vertex color MapDelta, adding whatever new map verts or faces are needed to match the current MeshDelta. After calling this method, applying this MeshDelta to a mesh without vertex colors will result in a vertex color map with all white vertices and a topology identical to the mesh.

**Prototype:**

**void AddMap(int mapID);**

**Remarks:**

Activates the specified MapDelta, adding whatever new map verts or faces are needed to match the current MeshDelta. If mapID is 0, the standard vertex colors (white) will be applied (see AddVertexColors). Otherwise, after calling this method, applying this MeshDelta to a mesh without the specified map active will result in a map with the same topology as the mesh and UVVerts that are copies of the mesh vertices.

**Parameters:**

**int mapID**

The map channel to add. 0 represents vertex colors, 1 is the original map channel (referred to in class Mesh by tVerts and tvFaces), and 2-99 are the new map channels (stored in meshes in the MeshMap class).

**Prototype:**

**void AddVertexData(int vdChan, Mesh \*m=NULL);**

**Remarks:**

Activates the specified vertex data channel, creating a default set of vertex data to match the output of the current MeshDelta. ("Default" values of vertex data depend on the channel, and are given by VDataDefault (vdChan).)

**Parameters:**

**int vdChan**

The vertex data channel

**Mesh \*m=NULL**

A pointer to the Mesh object.

## Mesh Interaction

### Prototype:

**void FillInFaces(Mesh & m);**

### Remarks:

To make things easy for developers, it's possible to create or remap mapping faces to use UNDEFINED mapping verts. This routine, which is called by Apply below, fills in those UNDEFINED mapping values with the mapping vertices used by neighboring faces, or, if necessary, by new mapping vertices. (These new mapping vertices are always (.5,.5,0) for regular map channels or (1,1,1) for the vertex color channel.)

This process does not produce very good maps, but it allows a sort of minimal mapping support that prevents maps from being lost before the user can make their manual corrections.

### Prototype:

**void Apply(Mesh & m);**

### Remarks:

Changes the given mesh by this MeshDelta, in the following manner:

First, any maps that are supported by the MeshDelta but not by the mesh are assigned to the mesh in their default form. (Vertex color channels are white, and other maps are copies of the mesh vertices. All have the same topology as the mesh.)

Next, any UNDEFINED mapping verts in the MeshDelta are filled in by FillInFaces.

Then the new vertices are added, creates first, followed by clones. The original vertices are then moved.

The faces are then modified, by applying all the FaceRemaps, FaceChanges, and FaceSmooths to the appropriate faces. New faces (in fCreate) are appended to the end of the face list.

Map changes are applied to all active maps, and map channels not supported by this MeshDelta are removed.

After all that is done, the vertices and faces marked in the vDelete and fDelete arrays are deleted.

Finally, the vertex data, vertex hide, and selections kept in the MeshDelta are

applied to the result.

**Prototype:**

**DWORD PartsChanged();**

**Remarks:**

Indicates what data channels of a mesh would be changed by this MeshDelta. For instance, a MeshDelta with vertex moves but no other changes would return PART\_GEOM|PART\_SELECT. PART\_GEOM represents the moves, and PART\_SELECT represents the fact that MeshDeltas always overwrite selection info. Most of the changes in a MeshDelta will alter PART\_TOPO. PART\_VERTCOLOR and PART\_TEXMAP may also be returned.

This is especially useful for knowing what parts of a mesh to back up in a restore object for an undo/redo. (See the SDK implementation of Editable Mesh for an example of this.) Also, it can be used for invalidating temporary data, as in both Edit and Editable Mesh.

## **Composition & Operators**

MeshDeltas can be multiplied together like Transforms before being applied to a mesh. This is especially useful in Edit Mesh, where a single MeshDelta is used to store the current "state" of the Edit Mesh, and extra MeshDeltas are created for each operation and added on to the main MeshDelta.

An example of this is as follows: given a Mesh m that we want to divide a face on, then we want to break the vertex created in the face divide. These operations are adequately handled by the DivideFace and BreakVerts methods independently, but to combine the two into one operation, we must compose two MeshDeltas.

```
MeshDelta DivideThenBreak (const Mesh & m, int ff) {  
MeshDelta md1, md2;  
Mesh mcopy = m;  
md1.DivideFace (mcopy, ff);  
int nvid = md1.vnum; // since DivideFace creates exactly one vertex,  
this must be its index in the result.  
md1.Apply (mcopy);  
md2.InitToMesh (mcopy); // second MeshDelta must be based on
```

```

first MeshDelta result.
BitArray vset;
vset.SetSize (mcopy.numVerts);
vset.Set (nvid);
md2.BreakVerts (mcopy, vset); // breaks the new vertex into a
separate vert for each face.
md1.Compose (md2); // Adds the second MeshDelta into the first.
return md1;
}

```

If all we wanted was to perform these operations on the mesh given, we could have done so without making the composition. The point here is to be able to construct complex MeshDeltas representing a series of user operations while remaining flexible enough to respond to changes in the actual mesh given as input. This is a central feature of the Edit Mesh modifier, completely rewritten for 3.0, found in **MAXSDK\SAMPLES\MODIFIERS**.

**Prototype:**

**DWORD Compose(MeshDelta & td);**

**Remarks:**

Appends the given MeshDelta to the current one.

**Parameters:**

**MeshDelta & td**

The MeshDelta to append. This MeshDelta may be modified to make it suitable, ie the vnum and fnum values will be set to the expected output of the current MeshDelta if they don't already match. (This may result in the loss of some data – see "SetInVNum" and "SetInFNum" for more information.)

**Prototype:**

**MeshDelta & operator\*=(MeshDelta & td);**

**Remarks:**

Appends the given MeshDelta to the current one.

**Parameters:**

**MeshDelta & td**

The MeshDelta to append. This MeshDelta may be modified to make it suitable, ie the vnum and fnum values will be set to the expected output of the current MeshDelta if they don't already match. (This may result in the loss of some data – see "SetInVNum" and "SetInFNum" for more information.)

**Prototype:**

**MeshDelta & operator=(MeshDelta & td);**

**Remarks:**

Assignment operator – makes this MeshDelta just like the one given.

**Prototype:**

**DWORD ChangeFlags(Tab<DWORD> \*mChannels=NULL);**

**Remarks:**

Indicates what parts of a MeshDelta could be changed if this MeshDelta were appended to it. This is useful when backing up MeshDelta for Restore Objects. For instance, if you had a MeshDelta with lots of face smoothing changes, and you wanted to compose it with one that only moved vertices, there would be no reason to back up the smoothing changes for an undo.

**Parameters:**

**Tab<DWORD> \*mChannels=NULL**

If non-NULL, this points to a table that should be filled with change flags for the various map channels. The table is set to the number of map channels, and each DWORD in it is filled in by calling MapDelta::ChangeFlags on the appropriate map channel (or left at zero if the map channel is inactive.)

**Return Value:**

Returns some combination of the following flags, corresponding to the data members that would be changed:

**MDELTA\_VMOVE**

**MDELTA\_VCREATE**

**MDELTA\_VCLONE**

**MDELTA\_VDELETE**

**MDELTA\_VDATA**

**MDELTA\_FREMAP**

**MDELTA\_FCHANGE**  
**MDELTA\_FCREATE**  
**MDELTA\_FDELETE**  
**MDELTA\_FDATA**  
**MDELTA\_NUMBERS**  
**MDELTA\_FSMOOTH**

Note that in 4.0 and thereafter, the **MDELTA\_VCREATE** and **MDELTA\_VCLONE** flags are identical and represent the same information. (This was not true in 3.0 or 3.1.)

**Prototype:**

**void CopyMDChannels(MeshDelta & from, DWORD channels, Tab<DWORD> \*mChannels=NULL);**

**Remarks:**

Copies the specified parts of the MeshDelta. (Useful in combination with ChangeFlags to create efficient Restore objects.)

**Parameters:**

**MeshDelta & from**

The MeshDelta to copy into this.

**DWORD channels**

Indicates the parts to copy – some combination of the following flags:

**MDELTA\_VMOVE**  
**MDELTA\_VCREATE**  
**MDELTA\_VCLONE**  
**MDELTA\_VDELETE**  
**MDELTA\_VDATA**  
**MDELTA\_FREMAP**  
**MDELTA\_FCHANGE**  
**MDELTA\_FCREATE**  
**MDELTA\_FDELETE**  
**MDELTA\_FDATA**

## **MDELTA\_NUMBERS**

## **MDELTA\_FSMOOTH**

Note that in 4.0 and thereafter, the **MDELTA\_VCREATE** and **MDELTA\_VCLONE** flags are identical and represent the same information. (This was not true in 3.0 or 3.1.)

### **Tab<DWORD> \*mChannels=NULL**

If non-NULL, this points to a table that contains channels to copy in the various map channels. The table should be of the size of the number of map channels. For each active map channel in from, the corresponding DWORD in this table is passed in MapDelta::CopyMDChannels to copy the relevant parts of the map.

## **Characteristics**

The following methods give useful information about the MeshDelta.

### **Prototype:**

**DWORD NumVMove(DWORD inVNum);**

### **Remarks:**

Returns the number of vertex moves that would be applied to a mesh with the specified number of vertices. If that number equals this MeshDelta's vnum, this is simply vMove.Count().

### **Parameters:**

**DWORD inVNum**

The number of vertices in the input mesh we're inquiring about.

### **Prototype:**

**DWORD NumVClone(DWORD inVNum);**

### **Remarks:**

Returns the number of vertex clones & creates that would be applied to a mesh with the specified number of vertices. If that number equals this MeshDelta's vnum, this is simply vClone.Count(). If, however, inVNum is lower than the expected vnum, some of the clones might be eliminated, reducing this number.

### **Parameters:**

**DWORD inVNum**

The number of vertices in the input mesh we're inquiring about.

**Prototype:****DWORD NumVDelete(DWORD inVNum);****Remarks:**

Returns the number of vertex deletes that would be applied to a mesh with the specified number of vertices. If that number equals this MeshDelta's vnum, this is simply vDelete.NumberSet ().

**Parameters:****DWORD inVNum**

The number of vertices in the input mesh we're inquiring about.

**Prototype:****DWORD NumFDelete(DWORD inFNum);****Remarks:**

Returns the number of face deletes that would be applied to a mesh with the specified number of faces. If inFNum equals this MeshDelta's fnum, this is simply fDelete.NumberSet ().

**Parameters:****DWORD inFNum**

The number of faces in the input mesh we're inquiring about.

**Prototype:****int NumFCreate ();****Remarks:**

Returns the number of face creates in this MeshDelta.

**Prototype:****DWORD outVNum();****Remarks:**

Returns the number of vertices in the output mesh, assuming that the input

mesh is of the expected (vnum) size.

**Prototype:**

**DWORD outVNum(int inVNum);**

**Remarks:**

Returns the number of vertices in the output mesh, assuming that the input mesh has the specified number of vertices.

**Parameters:**

**DWORD inVNum**

The number of vertices expected in the input mesh.

**Prototype:**

**DWORD outFNum();**

**Remarks:**

Returns the number of faces in the output mesh, assuming that the input mesh is of the expected (fnum) size.

**Prototype:**

**DWORD outFNum(int inFNum);**

**Remarks:**

Returns the number of faces in the output mesh, assuming that the input mesh has the specified number of faces.

**Parameters:**

**DWORD inFNum**

The number of faces expected in the input mesh.

**Prototype:**

**Point3 outVert(Mesh & m, DWORD v);**

**Remarks:**

Returns the expected location in the output mesh of the specified vertex.

**Parameters:**

**Mesh & m**

The input mesh.

**DWORD v**

The vertex you want the output location of. This index is input-based – the vertex index in **m**, not in the output mesh.

**Prototype:**

**Face outFace(Mesh & m, DWORD f);**

**Remarks:**

Returns the specified face as it would appear in the MeshDelta output. Face Changes, Smooths, and Remaps are applied.

**Parameters:**

**Mesh & m**

The input mesh.

**DWORD f**

The index of the face you want the output version of. This index is input-based – the face index in **m**, not in the output mesh.

**Prototype:**

**DWORD RemapID(DWORD ff);**

**Remarks:**

Obtains the index of the fRemap entry that relates to this face.

**Parameters:**

**DWORD ff**

The input-based face index.

**Return Value:**

If there is such an entry, the index is returned, so fRemap[RemapID(ff)].fid == ff. If there is no remap record for this face, the method returns UNDEFINED.

**Prototype:**

**DWORD IsRemapped(DWORD ff, DWORD vid);**

**Remarks:**

Tells whether the specified corner of the specified face has been remapped in

this MeshDelta.

**Parameters:**

**DWORD ff**

The input-based face index.

**DWORD vid**

The corner of the face – 0, 1, or 2.

**Return Value:**

If this corner has been remapped, it returns the vertex it's been remapped to. Otherwise, it returns UNDEFINED.

**Prototype:**

**DWORD MoveID(DWORD i);**

**Remarks:**

Obtains the index of the vMove entry that relates to this vertex.

**Parameters:**

**DWORD i**

The input-based vertex index.

**Return Value:**

The index in the vMove array of the vertex move corresponding to this vertex, or UNDEFINED if this vertex has no move associated with it.

**Prototype:**

**bool IsVCreate(DWORD i);**

**Remarks:**

No longer used.

This method was used in 3.0 and 3.1 to indicate whether the specified vertex was created as a create, not a clone in this MeshDelta. In 4.0, we integrated the clone and create records together, and this method is now set to always return FALSE.

**Parameters:**

**DWORD i**

The output-based vertex index.

**Prototype:****bool IsVClone(DWORD i);****Remarks:**

Indicates whether or not the specified vertex is created in this MeshDelta.

**Parameters:****DWORD i**

The output-based vertex index.

**Prototype:****DWORD VCloneOf(DWORD i);****Remarks:**

Tells you what input vertex the specified output vertex is a clone of.

**Parameters:****DWORD i**

The output-based vertex index.

**Return Value:**

The index in the input mesh of the original vertex this one's a clone of. If this vertex is not a clone, UNDEFINED is returned.

**Prototype:****bool IsFCreate(DWORD i);****Remarks:**

Indicated whether the specified face was created in this MeshDelta.

**Parameters:****DWORD i**

The face index in the output mesh.

**Lookup Table Methods****Prototype:****void UpdateLUTs(int extraV=0, int extraF=0);****Remarks:**

Updates the MeshDelta's internal lookup tables, which make use of all delete and create records to set up a correspondence between output and input vertices and faces.

**Parameters:**

**int extraV**

If nonzero, this indicates the number of extra spaces that should be added to the lookup table. This is useful for example if you want the lookup table to still be valid after the next <extraV> vertex creates or clones.

**int extraF**

If nonzero, this indicates the number of extra spaces that should be added to the lookup table. This is useful for example if you want the lookup table to still be valid after the next <extraF> face creates.

**Prototype:**

**void ClearLUTs();**

**Remarks:**

Invalidates and clears the lookup tables. This usually only needs to be called internally.

**Prototype:**

**DWORD VLut(DWORD i);**

**Remarks:**

Finds the input mesh index of the vertex with the specified output mesh index. Note that these indices are the same if there are no vertex deletes.

**Parameters:**

**DWORD i**

The output mesh index.

**Return Value:**

The input mesh index of the same vertex. If the vertex specified is actually created by this MeshDelta, the return value would be vnum+i for vClone[i].

**Prototype:**

**DWORD FLut(DWORD i);**

**Remarks:**

Finds the input mesh index of the face with the specified output mesh index.  
Note that these indices are the same if there are no face deletes.

**Parameters:****DWORD i**

The output mesh index.

**Return Value:**

The input mesh index of the same face. If the face specified is actually created by this MeshDelta, the return value would be fnum+i for fCreate[i].

**Prototype:**

**DWORD PostVIndex(DWORD i);**

**Remarks:**

Returns the index in the output mesh of the specified input vertex.  
Note that these indices are the same if there are no vertex deletes.

**Parameters:****DWORD i**

The index of the vertex in the input mesh.

**Return Value:**

The output mesh index of the same vertex. If the vertex was deleted in this MeshDelta, UNDEFINED is returned.

**Prototype:**

**DWORD PostFIndex(DWORD i);**

**Remarks:**

Returns the index in the output mesh of the specified input face.  
Note that these indices are the same if there are no face deletes.

**Parameters:****DWORD i**

The index of the face in the input mesh.

**Return Value:**

The output mesh index of the same face. If the face was deleted in this

MeshDelta, UNDEFINED is returned.

## Basic Operations

These operations are the "building blocks" of MeshDeltas. All of them may be safely performed on MeshDeltas that are already quite complex. Those that accept DWORD indices require **output** mesh indices, as all operations are appended to the end of the existing delta.

### Prototype:

```
void Move(int i, const Point3 & p);
```

### Remarks:

Moves a single vertex. (Note that if the same vertex is moved twice, the new move is simply added to the old one – there is never more than one VertMove in the vMove array for a single input vertex.)

### Parameters:

**int i**

The index of the vertex in the output mesh.

**const Point3 & p**

The vector to move the vertex by.

### Prototype:

```
void Move(BitArray & sel, const Point3 & p);
```

### Remarks:

Moves the specified vertices. (Note that if the same vertex is moved twice, the new move is simply added to the old one – there is never more than one VertMove in the vMove array for a single input vertex.)

### Parameters:

**BitArray & sel**

Indicates which vertices should be moved. Vertices are indexed based on the output mesh.

**const Point3 & p**

The vector to move the vertices by.

**Prototype:**

```
void Move(VertMove *vm, int num);
```

**Remarks:**

Adds in the specified vertex moves. (Note that if the same vertex is moved twice, the new move is simply added to the old one – there is never more than one VertMove in the vMove array for a single input vertex.)

**Parameters:**

**VertMove \*vm**

A pointer to an array of VertMoves to apply to this MeshDelta. Vertices are indexed based on the output mesh.

**int num**

The size of the VertMove array.

**Prototype:**

```
DWORD VCreate(Point3 *p, int num=1, BitArray *sel=NULL,  
BitArray *hide=NULL);
```

**Remarks:**

Creates new vertices.

**Parameters:**

**Point3 \*p**

A pointer to an array of points representing the new vertices.

**int num**

The size of the point array.

**BitArray \*sel=NULL**

If non-NULL, this points to a BitArray of size num that indicates which of these new vertices should be selected. (If NULL, none of the new vertices are selected.)

**BitArray \*hide=NULL**

If non-NULL, this points to a BitArray of size num that indicates which of these new vertices should be hidden. (If NULL, none of the new vertices are hidden.)

**Return Value:**

The index (in the output mesh) of the first of these new vertices.

**Prototype:**

**DWORD VClone(DWORD \*v, int num=1);**

**Remarks:**

Clones some vertices.

**Parameters:**

**DWORD \*v**

A pointer to an array of indices of verts that should be cloned.

**int num**

The size of the array.

**Return Value:**

The index (in the output mesh) of the first of the clones.

**Prototype:**

**DWORD VClone(DWORD \*v, Point3 \*off, int num=1);**

**Remarks:**

Clones some vertices.

**Parameters:**

**DWORD \*v**

A pointer to an array of ids of vertices that should be cloned.

**Point3 \*off**

A pointer to an array of offsets for the clones.

**int num**

The size of the arrays.

**Return Value:**

The index (in the output mesh) of the first of the clones.

**Prototype:**

**DWORD VClone(VertMove \*vm, int num=1);**

**Remarks:**

Clones some vertices.

**Parameters:**

**VertMove \*vm**

A pointer to an array of VertMoves indicating which vertices should be cloned and what offsets the clones should use.

**int num**

The size of the vm array.

**Return Value:**

The index (in the output mesh) of the first of the clones.

**Prototype:**

**DWORD VClone(DWORD v);**

**Remarks:**

Clones a single vertex.

**Parameters:**

**DWORD v**

The index (in the output mesh) of the vertex you wish to clone.

**Return Value:**

The index (in the output mesh) of the clone.

**Prototype:**

**DWORD VClone(DWORD v, Point3 off);**

**Remarks:**

Clones and offsets a single vertex.

**Parameters:**

**DWORD v**

The index (in the output mesh) of the vertex you wish to clone.

**Point3 off**

The desired offset from the original vertex.

**Return Value:**

The index (in the output mesh) of the clone.

**Prototype:**

**void VDelete(DWORD \*v, int num=1);**

**Remarks:**

Deletes the specified vertices.

**Parameters:****DWORD \*v**

A pointer to an array of (output-based) ids of the vertices that should be deleted.

**int num**

The number of vertices to delete (the size of the v array).

**Prototype:**

```
void VDelete(BitArray & vdel);
```

**Remarks:**

Deletes the specified vertices.

**Parameters:****BitArray & vdel**

A BitArray, of size OutVNum, indicating which of the vertices should be deleted. Vertices in this array are indexed by output mesh order.

**Prototype:**

```
DWORD FCreate(Face *f, int num=1);
```

**Remarks:**

Creates new faces.

Note: MapDeltas must be kept up to date with all new face creations – see CreateDefaultMapFaces.

**Parameters:****Face \*f**

A pointer to an array of faces to be added to the MeshDelta.

**int num**

The size of the face array.

**Return Value:**

The index (in the output mesh) of the first of these new faces.

**Prototype:**

**DWORD FCreate(FaceCreate \*f, int num=1);**

**Remarks:**

Creates new faces. This is what's used to add in face creations in the all-important Compose method. It assumes that the "originals" in the array of FaceCreates are post-indexed, and uses FLut and extracts originals for creates-of-creates as appropriate.

Note: MapDeltas must be kept up to date with all new face creations – see CreateDefaultMapFaces.

**Parameters:**

**FaceCreate \*f**

A pointer to an array of face create records to be added to the MeshDelta.

**int num**

The size of the face create array.

**Return Value:**

The index (in the output mesh) of the first of these new faces.

**Prototype:**

**DWORD FCreateQuad(DWORD \*v, DWORD smG=0, MtlID matID=0, int orig=UNDEFINED);**

**Remarks:**

Creates 2 new faces, forming a quad.

Note: MapDeltas must be kept up to date with all new face creations – see CreateDefaultMapFaces.

**Parameters:**

**DWORD \*v**

A pointer to an array of 4 vertices to be used as corners of the quad.

**DWORD smG=0**

The smoothing group desired for the new faces.

**MtlID matID=0**

The material ID desired for the new faces.

**int orig=UNDEFINED**

This optional parameter sets the map faces to undefined.

**Return Value:**

The index (in the output mesh) of the first of these 2 new faces.

**Prototype:**

```
DWORD FClone(Face & f, DWORD ff, DWORD remapFlags=0,  
DWORD *v=NULL);
```

**Remarks:**

Creates a new face by copying an existing face. The result is put into the fCreate array and treated thereafter like a face create – this is different from vertex clones, which are maintained separately from vertex creates.

Note: MapDeltas must be kept up to date with all new face creations – see CreateDefaultMapFaces.

**Parameters:****Face & f**

The face we wish to clone. (This is typically generated by the outFace method.)

**DWORD ff**

The (output-based) index of the face we're cloning. (This is used to copy face and edge selection.)

**DWORD remapFlags=0****DWORD \*v=NULL**

If we wish to remap any of the corners of this face while cloning, the appropriate flags and vertices should be passed in these last two arguments. v should point to an array of 3 vertex (output) indices, although the ones not marked as used by the remapFlags need not be set to anything in particular. See class FaceRemap for more information about face remapping.

**Return Value:**

The index (in the output mesh) of the new face.

**Prototype:**

```
DWORD CreateDefaultMapFaces(int num=1);
```

**Remarks:**

MapDeltas must **always** keep their faces in sync with the parent MeshDelta. If

the developer creates new faces, but doesn't want to go through the bother of figuring out exactly how the related map faces should look, this method may be used to create map faces with UNDEFINED verts. These can then be filled in automatically later. (See the MeshDelta FillInFaces method for details.)

For **every** face created in the MeshDelta, either CreateDefaultMapFaces should be called, or map faces should be created in every active map channel, using MapDelta::FCreate and related methods.

**Parameters:**

**int num=1**

The number of default faces we wish to create in each active map channel.

**Prototype:**

**void FRemap(FaceRemap \*f, int num=1);**

**Remarks:**

Adds face remaps to this MeshDelta. If the face specified in each FaceRemap already has a remap record, the two are combined. If the face specified is a face created by this MeshDelta, the remap is applied directly to the fCreate entry instead of being stored in fRemap.

**Parameters:**

**FaceRemap \*f**

A pointer to an array of FaceRemap that should be appended to this MeshDelta.

Note that the faces and vertices in each FaceRemap must be indexed by their positions after all of the current MeshDelta's creates, clones, etc, but before any vertex or face deletes. Vertex index values of 0 through vnum-1 are considered to be the original mesh's vertices; values above this are cloned or created vertices. Likewise, face index values of 0 through fnum-1 are considered to be the original mesh faces, while fnum through fnum+fCreate.Count()-1 are this MeshDelta's face creates.

**int num=1**

The number of elements in the FaceRemap array.

**Prototype:**

**void FRemap(DWORD f, DWORD flags, DWORD \*v);**

**Remarks:**

Adds a face remap to this MeshDelta. If the face specified already has a remap record, the two are combined. If the face specified is a face created by this MeshDelta, the remap is applied directly to the fCreate entry instead of being stored in fRemap.

**Parameters:****DWORD f**

The face to remap.

Note that this face must be indexed by its position after all of the current MeshDelta's face creates, but before any face deletes. Face index values of 0 through fnum-1 are considered to be the original mesh faces, while fnum through fnum+fCreate.Count()-1 are this MeshDelta's face creates.

**DWORD flags**

Face Remap flags – these indicate which vertices should be remapped. The possibilities are FR\_V0 (1), FR\_V1 (2), and FR\_V2 (4). (See class FaceRemap for more information.)

**DWORD \*v**

A pointer to the vertices to remap the face to use. Only the positions indicated in the remap flags need contain meaningful data.

Note that the vertices indicated here must be indexed by their positions after all of the current MeshDelta's creates and clones, but before any vertex deletes – essentially input-based indexing. Vertex index values of 0 through vnum-1 are considered to be the original mesh's vertices; values of vnum through vnum+vCreate.Count()-1 are considered to be this MeshDelta's newly created vertices; and values above this are cloned vertices.

**Prototype:**

```
void FChange(FaceChange *f, int num=1);
```

**Remarks:**

Appends some face changes to the current MeshDelta. Face changes can encompass changes to face material IDs, edge visibility, or face hiding. See class FaceChange for more information.

**Parameters:**

**FaceChange \*f**

A pointer to an array of new face changes for this MeshDelta.

**int num=1**

The number of elements in the FaceChange array.

**Prototype:**

**void FChange(DWORD f, DWORD flags, DWORD dat);**

**Remarks:**

Changes the characteristics of one face.

**Parameters:**

**DWORD f**

The output-indexed face to change.

**DWORD flags**

**DWORD dat**

These two parameters describe the change desired. See class FaceChange for a description of these flags. flags indicates which characteristics should be set, and dat includes the on-or-off state of each flag we're setting. So for example FChange (26, ATTRIB\_EDGE\_A| ATTRIB\_EDGE\_B, ATTRIB\_EDGE\_A) would set face 26 to have the first edge visible and the second invisible, without changing the existing visibility for the third edge.

**Prototype:**

**void SetMatID(DWORD f, MtlID mt);**

**Remarks:**

Sets the material ID for the specified face. If the face is created by this MeshDelta, the fCreate record is amended. If it's an input face, a FaceChange record is created or amended.

**Parameters:**

**DWORD f**

The output-indexed face to change.

**MtlID mt**

The desired material ID.

**Prototype:**

**void SetEdgeVis(DWORD f, DWORD ed, BOOL vis=TRUE);**

**Remarks:**

Sets the edge visibility for the specified side of the specified face. If the face is created by this MeshDelta, the fCreate record is amended. If it's an input face, a FaceChange record is created or amended.

**Parameters:**

**DWORD f**

The output-indexed face to change.

**DWORD ed**

The side of the face to change (0, 1, or 2).

**BOOL vis=TRUE**

The desired visibility.

**Prototype:**

**void FSmooth(FaceSmooth \*f, int num=1);**

**Remarks:**

Adds smoothing group changes to this MeshDelta. See class FaceSmooth for more information.

**Parameters:**

**FaceSmooth \*f**

A pointer to an array of smoothing change records. The face IDs in these records should be output-indexed.

**int num=1**

The number of elements in the FaceSmooth array.

**Prototype:**

**void FSmooth(DWORD f, DWORD mask, DWORD val);**

**Remarks:**

Changes the smoothing groups on the specified face.

**Parameters:**

**DWORD f**

The output-based index of the face to change.

### **DWORD mask**

The smoothing groups to change.

### **DWORD val**

The smoothing group values. For instance, FSmooth (32, 7, 2) would set smoothing group 2 and clear groups 1 and 3 for face 32, since mask has bits 0, 1, and 2 set but val only has bit 1 set.

### **Prototype:**

```
void SetSmGroup(DWORD f, DWORD smG);
```

### **Remarks:**

Sets the smoothing groups on the specified face.

### **Parameters:**

#### **DWORD f**

The output-based index of the face to change.

#### **DWORD smG**

The smoothing groups to set. All bits not set in this parameter are cleared.

### **Prototype:**

```
void FDelete(DWORD *f, int num=1);
```

### **Remarks:**

Deletes the specified faces.

### **Parameters:**

#### **DWORD \*f**

A pointer to an array of output-based indices of faces we wish to delete. Note that all the elements should be based on the output before any deletions occur. So if you wanted to delete what are currently faces 3 and 5, you could pass an array with 3 and 5, you would **not** have to think, "Ah, 3 will be deleted, so I should use 4 instead of 5."

#### **int num=1**

The size of the array.

### **Prototype:**

```
void FDelete(BitArray & fdel);
```

**Remarks:**

Deletes the specified faces.

**Parameters:****BitArray & fdel**

The faces to delete. The faces are indexed by their output mesh positions.

**Advanced additive operations**

The following are more complex operations, built out of the simple operations above. The mesh given is expected to be result of the current MeshDelta. They should all work no matter how complex the current MeshDelta is, but they have not been extensively tested on complex MeshDeltas. To see how most of these work, look at Editable Mesh or Edit Mesh, both of which have source in **MAXSDK\SAMPLES**.

**Prototype:**

```
void AutoSmooth(Mesh & m, BitArray sel, float angle,  
AdjFaceList *af=NULL, AdjEdgeList *ae=NULL);
```

**Remarks:**

Automatically generates smoothing groups for the selected faces. Existing smoothing groups are ignored. See the AutoSmooth feature in Edit Mesh for an example.

**Parameters:****Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**BitArray sel**

The faces to AutoSmooth.

**float angle**

The maximum angle between faces that should be smoothed together.

**AdjFaceList \*af=NULL**

A pointer to the adjacent face list for this mesh. If NULL, the method constructs its own AdjFaceList.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method

constructs its own AdjEdgeList.

**Prototype:**

```
void Bevel(Mesh & m, BitArray vset, float outline, Tab<Point3>  
*odir, float height, Tab<Point3> *hdir);
```

**Remarks:**

Moves the selected vertices along the directions given to produce the movement corresponding to the mouse drags of a Bevel operation in Editable Mesh. Note that this method only changes geometry.

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**BitArray vset**

The set of vertices to move.

**float outline**

The amount of outlining to do in this Bevel move.

**Tab<Point3> \*odir**

The outline direction for each vertex. This should be given by MeshTempData::OutlineDir. See the Edit Mesh source for details.

**float height**

The amount of extrusion to do in this Bevel move.

**Tab<Point3> \*hdir**

The extrusion direction for each vertex. This should be given by MeshTempData::EdgeExtDir or FaceExtDir. See the Edit Mesh source for examples.

**Prototype:**

```
DWORD CreatePolygon(Mesh & m, int deg, int *v, DWORD  
smG=0, MtlID matID=0);
```

**Remarks:**

Create a polygon of any size. The polygon may be nonconvex, but should be (roughly) coplanar.

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**int deg**

The number of vertices used by this polygon (its degree).

**int \*v**

The indices of the vertices in **Mesh m** that this polygon should use.

**DWORD smG=0**

The desired smoothing group for the new polygon.

**MtlID matID=0**

The desired material ID for the new polygon.

**Prototype:**

**void DeleteVertSet(Mesh & m, BitArray sel);**

**Remarks:**

Deletes the specified vertices, along with any faces that used them.

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**BitArray sel**

The vertices to delete. Any face that uses any of the vertices selected here will also be deleted.

**Prototype:**

**void DeleteEdgeSet(Mesh & m, BitArray sel);**

**Remarks:**

Deletes all faces using the specified edges. (Doesn't delete any verts.)

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that

should be affected.

**BitArray sel**

The edges to delete the faces of. Edges are indexed by  $\text{face} \times 3 + \text{side}$ .

**Prototype:**

**void DeleteFaceSet(Mesh & m, BitArray sel);**

**Remarks:**

Deletes the specified faces. (Doesn't delete any verts.)

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**BitArray sel**

The faces to delete.

**Prototype:**

**void DeleteSelected(Mesh & m);**

**Remarks:**

Deletes the current subobject selection. If  $\text{m.selLevel}$  is `MESH_OBJECT`, nothing is deleted. If it's `MESH_VERTEX`, the faces using the selected vertices are also deleted.

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**Prototype:**

**void DeleteIsoVerts(Mesh & m);**

**Remarks:**

Deletes the vertices not in use by any faces.

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**Prototype:**

**void FlipNormal(Mesh & m, DWORD face);**

**Remarks:**

Flips the normal of the specified face (by switching the face's v[0] and v[1]). (Related map faces are also flipped.)

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**DWORD face**

The face to flip.

**Prototype:**

**void MakeSelFacesPlanar(Mesh & m, BitArray sel);**

**Remarks:**

Flattens the faces indicated into the same plane. The target plane is determined by the average of all the face centers and the average of all the face normals.

**Parameters:**

**Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**BitArray sel**

The faces to make coplanar.

**Prototype:**

**void MakeSelVertsPlanar(Mesh & m, BitArray sel);**

**Remarks:**

Flattens the vertices indicated into the same plane. The target plane is determined by the average position and normal of the vertices.

**Parameters:****Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**BitArray sel**

The vertices to make coplanar.

**Prototype:**

```
void MoveVertsToPlane(Mesh & m, BitArray sel, Point3 & N,  
float offset);
```

**Remarks:**

Moves the vertices indicated into the specified plane. (The target plane is defined as all points which, when DotProd'd with N, return offset.) All vertices are moved along the normal vector N.

**Parameters:****Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

**BitArray sel**

The vertices to move into the plane.

**Point3 & N**

The unit normal to the plane.

**float offset**

The offset of the plane (also its distance from the origin).

**Prototype:**

```
void RestrictMatIDs(Mesh & m, int numMats);
```

**Remarks:**

Like the old standalone method "FitMeshIDsToMaterial", this method limits the material IDs to values between 0 and numMats-1. This is useful eg in matching the number of material Ids to the number of materials used on this node.

**Parameters:**

## **Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

## **int numMats**

The number of material ids allowed.

## **Prototype:**

```
void SelectFacesByFlags(Mesh & m, BOOL onoff, DWORD  
flagmask, DWORD flags);
```

## **Remarks:**

Sets or clears face selection depending on whether they match a pattern of flags. This is pretty much only useful for selecting or deselecting hidden faces:

```
mdelta.SelectFacesByFlags(*mesh, FALSE, FACE_HIDDEN,  
FACE_HIDDEN); // deselects hidden faces.
```

## **Parameters:**

### **Mesh & m**

The mesh, which should match the output of the current MeshDelta, that should be affected.

### **BOOL onoff**

Indicates whether faces should be selected or deselected if they match the flag pattern.

### **DWORD flagmask**

Indicates whether faces should be selected or deselected if they match the flag pattern.

### **DWORD flags**

Indicates whether faces should be selected or deselected if they match the flag pattern.

## **Advanced operations**

The following will initialize to the mesh given: they can't be used to "add" ops to an existing MeshDelta. (To add these ops, make a new MeshDelta, call one of the following, and append it to your previous one with Compose.) To see how most of these work, look at Editable Mesh or Edit Mesh, both of which have

source in **MAXSDK\SAMPLES**.

**Prototype:**

**void AttachMesh(Mesh & m, Mesh & attachment, Matrix3 & relativeTransform, int matOffset);**

**Remarks:**

Attaches another mesh to this one.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**Mesh & attachment**

The mesh this MeshDelta should attach.

**Matrix3 & relativeTransform**

The transform taking the attachment mesh from its object space to ours.

**int matOffset**

The offset that should be applied to all the material IDs in the attachment.

**Prototype:**

**void BreakVerts(Mesh & m, BitArray vset);**

**Remarks:**

Splits the selected vertices into a separate vertex for every face that uses them.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray vset**

The vertices that should be broken.

**Prototype:**

**void ChamferEdges(Mesh & m, BitArray eset, MeshChamferData & mcd, AdjEdgeList \*ae=NULL);**

**Remarks:**

Performs the topological changes needed for an edge chamfer, and creates the information necessary to do the geometric changes.

**Parameters:****Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray eset**

The edges to chamfer.

**MeshChamferData & mcd**

A class in which the directions and limits of movement for all the vertices involved in the chamfer should be stored. See class MeshChamferData.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**Prototype:**

```
void ChamferMove(Mesh & m, MeshChamferData & mcd, float amount, AdjEdgeList *ae=NULL);
```

**Remarks:**

Moves the relevant vertices to a specified chamfer value.

**Parameters:****Mesh & m**

The mesh this MeshDelta should be based on.

**MeshChamferData & mcd**

A class in which the directions and limits of movement for all the vertices involved in the chamfer have been stored by a previous ChamferVertices or ChamferEdges call. See class MeshChamferData.

**float amount**

The amount (in object space units) of chamfering to do. For vertex chamfers, this indicates how far along the edges each point should move. For edge chamfers, it represents how far along each face each edge should move.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**Prototype:**

**void ChamferVertices(Mesh & m, BitArray vset,  
MeshChamferData & mcd, AdjEdgeList \*ae=NULL);**

**Remarks:**

Performs the topological changes needed for a vertex chamfer, and creates the information necessary to do the geometric changes.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray vset**

The vertices that should be chamfered.

**MeshChamferData & mcd**

A class in which the directions and limits of movement for all the vertices involved in the chamfer should be stored. See class MeshChamferData.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**Prototype:**

**void CloneFaces(Mesh & m, BitArray fset);**

**Remarks:**

Clones the specified faces, along with the vertices and mapping vertices they use.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray fset**

The faces that should be cloned.

**Prototype:**

**void CloneVerts(Mesh & m, BitArray vset);**

**Remarks:**

Clones the specified vertices. More efficient on an initialized MeshDelta than

VClone, which has to be able to cope with existing complex MeshDeltas.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray vset**

The vertices that should be cloned.

**Prototype:**

```
void CollapseEdges(Mesh & m, BitArray ecol, AdjEdgeList  
*ae=NULL);
```

**Remarks:**

Collapses the edges indicated down to a point.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray ecol**

The edges to collapse.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**Prototype:**

```
DWORD Cut(Mesh & m, DWORD ed1, float prop1, DWORD ed2,  
float prop2, Point3 &norm, bool fixNeighbors=TRUE, bool  
split=FALSE);
```

**Remarks:**

Cuts the mesh from a point on one edge to a point on another, along a line drawn by looking at the mesh from a particular viewpoint. (See Edit Mesh's Cut feature for an illustration.)

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**DWORD ed1**

The edge that the cut starts on. The edge is indexed by  $\text{face} \cdot 3 + \text{side}$ , so the start vertex is  $\text{m.faces}[\text{ed1}/3].\text{v}[\text{ed1}\%3]$ , and the end vertex is  $\text{m.faces}[\text{ed1}/3].\text{v}[(\text{ed1}+1)\%3]$ .

**float prop1**

The position on the edge to start the cut from. 0 means the start vertex of the edge, and 1 means the end vertex.

**DWORD ed2**

The edge that the cut should end on.

**float prop2**

The position on the edge to finish the cut on. 0 means the start vertex of the edge, and 1 means the end vertex.

**Point3 &norm**

The direction of view. The cut will take place on this "side" of the mesh, in the plane formed by this vector and the direction from the start to the end.

**bool fixNeighbors=TRUE**

Indicates whether the faces on the other side of each end of the cut should be split to prevent splits at the ends.

**bool split=FALSE**

Indicates whether the cut should actually split the mesh apart or just refine it by adding geometry.

**Prototype:**

```
void Detach(Mesh & m, Mesh *out, BitArray fset, BOOL faces,  
            BOOL del, BOOL elem);
```

**Remarks:**

Detaches a subset of the geometry from the mesh given, either separating it as a new element or creating a new mesh with it.

**Parameters:****Mesh & m**

The mesh this MeshDelta should be based on.

**Mesh \*out**

The new mesh into which the detached portion can be put. (If elem is TRUE, this is not used and may be NULL.)

### **BitArray fset**

The selection to detach. If faces is TRUE, this is a face selection. Otherwise, it's a vertex selection.

### **BOOL faces**

If TRUE, we should detach the selected faces; otherwise, we should detach the selected vertices.

### **BOOL del**

Indicates whether the detached portion should be deleted from the original mesh.

### **BOOL elem**

If TRUE, we're actually just detaching to an element, and the out mesh will not be used.

### **Prototype:**

```
void DivideEdge(Mesh & m, DWORD ed, float prop=.5f,  
AdjEdgeList *el=NULL, bool visDiag1=FALSE, bool  
fixNeighbors=TRUE, bool visDiag2=FALSE, bool split=FALSE);
```

### **Remarks:**

Divides the specified edge, adding a point and dividing faces to match.

### **Parameters:**

#### **Mesh & m**

The mesh this MeshDelta should be based on.

#### **DWORD ed**

The edge to divide, indexed as face\*3+side.

#### **float prop=.5f**

The proportion along the edge where the division should occur. 0 is the start vertex, m.faces[ed/3].v[ed%3], and 1 is the end vertex, m.faces[ed/3].v[(ed+1)%3].

#### **AdjEdgeList \*el=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList if needed.

#### **bool visDiag1=FALSE**

Indicates whether the diagonal connecting the new point on this edge with the far corner of face ed/3 should be visible.

**bool fixNeighbors=TRUE**

Indicates whether the face on the other side of this edge, that is, the face using this edge that isn't ed/3, should be divided as well to prevent the introduction of a seam.

**bool visDiag2=FALSE**

Indicates whether the diagonal connecting the new point on this edge with the far corner of the face on the other side of the edge should be visible. (Not used if fixNeighbors is FALSE.)

**bool split=FALSE**

Indicates whether the method should create separate vertices for the two halves of the edge, splitting the mesh open along the diagonal(s).

**Prototype:**

```
void DivideEdges(Mesh & m, BitArray eset, AdjEdgeList  
*el=NULL);
```

**Remarks:**

Divides all the selected edges in half, creating new points and subdividing faces.

**Parameters:****Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray eset**

The edges to divide.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**Prototype:**

```
void DivideFace(Mesh & m, DWORD f, float *bary=NULL);
```

**Remarks:**

Divides the selected face into 3, by introducing a new point on the face and splitting the original face along lines from the corners to the new point.

**Parameters:**

## **Mesh & m**

The mesh this MeshDelta should be based on.

## **DWORD f**

The face to divide.

## **float \*bary=NULL**

A pointer to the barycentric coordinates of the new point on the face. If NULL, the center of the face is used.

Barycentric coordinates on a triangle are a set of three numbers between 0 and 1 that add up to 1. Any point on a triangle can be uniquely described by a set of these. The point corresponding to barycentric coordinates (a,b,c) on a face with corners A,B,C is  $a*A + b*B + c*C$ .

## **Prototype:**

```
void DivideFaces(Mesh & m, BitArray fset, MeshOpProgress *mop=NULL);
```

## **Remarks:**

Divides the selected faces into 3, by creating their center points and splitting the original faces along lines from the corners to the center.

## **Parameters:**

### **Mesh & m**

The mesh this MeshDelta should be based on.

### **BitArray fset**

The faces to divide.

### **MeshOpProgress \*mop=NULL**

If non-NULL, this points to an implementation of class MeshOpProgress which can be used to interrupt the algorithm if it's taking too long. See class MeshOpProgress for details.

## **Prototype:**

```
void EdgeTessellate(Mesh & m, BitArray fset, float tens, AdjEdgeList *ae=NULL, AdjFaceList *af=NULL, MeshOpProgress *mop=NULL);
```

## **Remarks:**

Tessellates the mesh. This algorithm is exactly the one used in the Tessellate modifier, when operating on "Faces" (triangle icon) and in "Edge" type.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray fset**

The faces to tessellate.

**float tens**

The tension for the edge tessellation. This value should be fairly small, between 0 and .5, and corresponds to the value in the Tessellate, Edit Mesh, or Editable Mesh UI's divided by 400.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**AdjFaceList \*af=NULL**

A pointer to the adjacent face list for this mesh. If NULL, the method constructs its own AdjFaceList.

**MeshOpProgress \*mop=NULL**

If non-NULL, this points to an implementation of class MeshOpProgress which can be used to interrupt the algorithm if it's taking too long. See class MeshOpProgress for details.

**Prototype:**

**void ExplodeFaces(Mesh & m, float thresh, bool useFaceSel=FALSE, AdjFaceList \*af=NULL);**

**Remarks:**

"Explodes" the mesh into separate elements.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**float thresh**

The threshold angle between faces that indicates whether they should be in the same or different element.

**bool useFaceSel=FALSE**

Indicates whether the mesh's current face selection should be used or if the whole mesh should be exploded.

**AdjFaceList \*af=NULL**

A pointer to the adjacent face list for this mesh. If NULL, the method constructs its own AdjFaceList.

**Prototype:**

```
void ExtrudeEdges(Mesh & m, BitArray eset, Tab<Point3>  
*edir=NULL);
```

**Remarks:**

Performs the topological changes necessary to extrude the indicated edges. (The geometric component is handled later by the Bevel method.)

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray eset**

The edges to extrude.

**Tab<Point3> \*edir=NULL**

Fills in the directions for moving all the relevant vertices to handle the geometric part of the extrusion. See the Edit Mesh or Editable Mesh source in **MAXSDK\SAMPLES\MODIFIERS** and **MAXSDK\SAMPLES\MESH\EDITABLEMESH** to see how this is used.

**Prototype:**

```
void ExtrudeFaces(Mesh & m, BitArray fset, AdjEdgeList  
*el=NULL);
```

**Remarks:**

Performs the topological changes necessary to extrude the indicated faces. (The geometric component is handled later by the Bevel method.)

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray fset**

The faces that should be extruded.

**AdjEdgeList \*el=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**Prototype:**

**void ResetVertWeights(Mesh & m);**

**Remarks:**

Resets all the vertex weights to 1.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**Prototype:**

**void SetFaceColors(Mesh & m, BitArray fset, VertColor vc , int mp=0);**

**Remarks:**

Sets the indicated faces to have vertex colors all equal to the color value specified. (This often involves creating new vertex color map vertices, so faces that neighbor the indicated faces are not affected.)

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray fset**

The faces that should be affected.

**VertColor vc**

The desired color.

**int mp=0**

The map channel. Use 0 for normal vertex colors, MAP\_SHADING for the illumination channel, and MAP\_ALPHA for the alpha channel.

**Prototype:**

```
void SetVertColors(Mesh & m, BitArray vset, VertColor vc , int mp=0);
```

**Remarks:**

Sets all vertex color map vertices associated with the indicated vertices to the specified color.

**Parameters:****Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray vset**

The vertices that should be affected. If more than one map vertex is used at this vertex, all of them have their colors set.

**VertColor vc**

The desired color.

**int mp=0**

The map channel. Use 0 for normal vertex colors, MAP\_SHADING for the illumination channel, and MAP\_ALPHA for the alpha channel.

**Prototype:**

```
void SetFaceAlpha (Mesh &m, BitArray fset, float alpha, int mp=MAP_ALPHA);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the indicated faces to have face alpha.

**Parameters:****Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray fset**

The faces that should be affected.

**float alpha**

The amount of alpha

**int mp=MAP\_ALPHA**

The map channel. Use 0 for normal vertex colors, MAP\_SHADING for the

illumination channel, and MAP\_ALPHA for the alpha channel.

**Prototype:**

```
void SetVertAlpha (Mesh &m, BitArray vset, float alpha, int  
mp=MAP_ALPHA);
```

**Remarks:**

This method is available in release 4.0 and later only.

Sets the indicated vertices to have face alpha.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray vset**

The vertices that should be affected.

**float alpha**

The amount of alpha

**int mp=MAP\_ALPHA**

The map channel. Use 0 for normal vertex colors, MAP\_SHADING for the illumination channel, and MAP\_ALPHA for the alpha channel.

**Prototype:**

```
void ResetVertCorners (Mesh & m);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method resets the vertex corners.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**Prototype:**

```
void SetVertWeights(Mesh & m, BitArray vset, float weight);
```

**Remarks:**

Sets the weights of the specified vertices. (These weight values are only used

in MeshSmooth NURMS mode as of 3ds max 3.0.)

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray vset**

The vertices that should have their weights set.

**float weight**

The weight to set.

**Prototype:**

```
void Slice(Mesh & m, Point3 N, float off, bool sep=FALSE, bool  
remove=FALSE, BitArray *fslice=NULL, AdjEdgeList  
*ae=NULL);
```

**Remarks:**

Slices the mesh along the specified slicing plane.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**Point3 N**

The normal of the slice plane.

**float off**

These parameters define the slicing plane as all points  $p$  satisfying the equation  $\text{DotProd}(p,N) = \text{off}$ .  $N$  should be normalized.

**bool sep=FALSE**

Indicates whether the slice should separate the mesh into two separate elements (if TRUE) or just refine the existing mesh by splitting faces (if FALSE).

**bool remove=FALSE**

Indicates whether the slice should remove the portion of the mesh "below" the slicing plane, where "below" is defined as the area where  $\text{DotProd}(p,N) - \text{off} < 0$ . If remove is TRUE, sep is ignored.

**BitArray \*fslice=NULL**

A bit array containing the list of faces to slice.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**Prototype:**

**void TurnEdge(Mesh & m, DWORD ed, AdjEdgeList \*el=NULL);**

**Remarks:**

"Turns" the specified edge. Only works on edges that have a face on both sides. These two faces are considered as a quad, where this edge is the diagonal, and remapped so that the diagonal flows the other way, between the vertices that were opposite this edge on each face.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**DWORD ed**

The edge to turn, indexed as face\*3+side.

**AdjEdgeList \*ae=NULL**

A pointer to the adjacent edge list for this mesh. If NULL, the method constructs its own AdjEdgeList.

**Prototype:**

**BOOL WeldByThreshold(Mesh & m, BitArray vset, float thresh);**

**Remarks:**

Welds all vertices that are sufficiently close together.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray vset**

The vertices that are candidates for being welded.

**float thresh**

The maximum distance (in object space units) between two vertices that will allow them to be welded.

**Return Value:**

Returns TRUE if any vertices were welded, FALSE if none were within threshold.

**Prototype:**

```
void WeldVertSet(Mesh & m, BitArray vset, Point3  
*weldPoint=NULL);
```

**Remarks:**

Welds the specified vertices together into one vertex, no matter how far apart they are.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray vset**

The vertices that should be welded.

**Point3 \*weldPoint=NULL**

If non-NULL, this points to the location we'd like to put the weld result. (If NULL, the result is put at the average location of the selected vertices.)

**Prototype:**

```
void PropagateFacing(Mesh & m, BitArray & fset, int face,  
AdjFaceList & af, BitArray & done, BOOL bias=1);
```

**Remarks:**

This method is designed for internal use, in UnifyNormals, but may also be called directly.

**Parameters:**

**Mesh & m**

The mesh this MeshDelta should be based on.

**BitArray fset**

The faces that should be affected.

**int face**

The starting face to propagate normal directions from.

**AdjFaceList & af**

The adjacent face list corresponding to the mesh. Required, can't be made

locally, for efficiency's sake.

### **BitArray & done**

Keeps track of which faces have had their normals unified. Faces that are set here when the call is made will not be processed, and will not be crossed to reach other faces. Faces that are still clear upon completion were not processed, probably because they were on a separate element from "face".

### **BOOL bias=1**

Used to keep track of whether the current face has been oriented correctly. For example, if the starting face was not selected in fset, and therefore doesn't have the right orientation, but you still want selected faces in the same element to be corrected, you would submit FALSE here.

### **Prototype:**

```
void UnifyNormals(Mesh & m, BitArray fset, AdjFaceList  
*af=NULL);
```

### **Remarks:**

Unifies normals on selected faces, making the normals consistent from face to face.

### **Parameters:**

#### **Mesh & m**

The mesh this MeshDelta should be based on.

#### **BitArray fset**

The faces that should be affected. Nonselected faces can be traversed by the algorithm, but they will not be corrected if their normals are pointing the "wrong" way.

#### **AdjFaceList \*af=NULL**

A pointer to the adjacent face list corresponding to the mesh. If NULL, an adjacent face list is computed by the method.

## **I/O, Debugging**

### **Prototype:**

```
IOResult Save(ISave *save);
```

### **Remarks:**

Saves the MeshDelta to a 3ds max file.

**Prototype:**

**IOResult Load(ILoad \*load);**

**Remarks:**

Loads the MeshDelta from a 3ds max file.

**Prototype:**

**void MyDebugPrint(bool lut=FALSE, bool mp=FALSE);**

**Remarks:**

Prints out the MeshDelta to the DebugPrint window in Developer Studio.

**Parameters:**

**bool lut=FALSE**

If TRUE, the vertex and face lookup tables are printed out.

**bool mp=FALSE**

Active map deltas are also printed out if this is TRUE.

## Class VertMove

See Also: [Class Mesh](#), [Class Point3](#).

class VertMove

### Description:

This class is available in release 3.0 and later only.

This class represents the notion of a mesh edit vertex move. The public data members provide the index of the vertex moved as well as the amount of the move in X, Y, Z.

### Data Members:

public:

**DWORD vid;**

The id of the vertex moved.

**Point3 dv;**

The amount of the move.

### Methods:

public:

### Prototype:

**VertMove();**

### Remarks:

Constructor.

### Prototype:

**VertMove(DWORD i, Point3 p);**

### Remarks:

Constructor. The data members are initialized to the values passed.

### Prototype:

**~VertMove();**

### Remarks:

Destructor.

**Prototype:**

**VertMove &operator=(VertMove & from);**

**Remarks:**

Assignment operator.

## Class UVVertSet

See Also: [Class Mesh](#), [Class Point3](#).

class UVVertSet

### Description:

This class is available in release 3.0 and later only.

This class represents the notion of a mesh edit UVW vertex assignment. The public data members provide the index of the vertex as well as the UVWVert.

### Data Members:

public:

**DWORD vid;**

The index of the vertex.

**UVVert v;**

The UVW vertex.

### Methods:

public:

### Prototype:

**UVVertSet();**

### Remarks:

Constructor.

### Prototype:

**UVVertSet(DWORD i, UVVert p);**

### Remarks:

Constructor. The data members are initialized to the values passed.

### Prototype:

**~UVVertSet();**

### Remarks:

Destructor.

**Prototype:**

**UVVertSet & operator=(UVVertSet & from);**

**Remarks:**

Assignment operator.

## Class FaceRemap

See Also: [Class Mesh](#), [Class Face](#), [Class Point3](#).

class FaceRemap

### Description:

This class is available in release 3.0 and later only.

This class represents the notion of a mesh edit Face Remap, which changes one or more of the vertices a face uses. It can also alter the visibility of the face's edge, its hidden state and its material ID.

### Data Members:

public:

**DWORD f,**

Face being remapped. This is a zero based index into the Mesh's **faces** array.

**DWORD flags**

Specifies which vertices to remap. One or more of the following values:

**FR\_V0** - Remap the 0th vertex.

**FR\_V1** - Remap the 1st vertex.

**FR\_V2** - Remap the 2nd vertex.

**FR\_ALL** - Remap all the vertices.

**DWORD v[3];**

Array of vertex indices. These indicate which vertex is used by each specified corner of the face being remapped.

### Methods:

public:

### Prototype:

**FaceRemap();**

### Remarks:

Constructor. The **flags** and **f** are set to 0.

### Prototype:

**FaceRemap(DWORD ff, DWORD fl, DWORD \*vv);**

**Remarks:**

Constructor.

**Parameters:**

**DWORD ff**

The face to remap.

**DWORD fl**

The flags to set.

**DWORD \*vv**

The array of vertex indicies.

**Prototype:**

**void Apply(Face &ff);**

**Remarks:**

Applies the vertex remapping to the given face based on the flags of this FaceRemap object.

**Parameters:**

**Face &ff**

The face whose vertices are remapped.

**Prototype:**

**void Apply(TVFace &tf);**

**Remarks:**

Applies the vertex remapping to the given map face based on the flags of this FaceRemap object.

**Parameters:**

**TVFace &tf**

The texture face whose tVerts are remapped by the verts of this FaceRemap object. The **v** data member used contains indices into the mesh object's **tVerts** array.

**Prototype:**

**void Apply(FaceRemap &fr);**

**Remarks:**

Assigns the flags and verts of this FaceRemap object to the FaceRemap passed.

**Parameters:****FaceRemap &fr**

The FaceRemap whose flags and verts are assigned.

**Prototype:**

**Face operator\*(Face &ff);**

**Remarks:**

Returns a new Face with the FaceRemap applied.

**Prototype:**

**TVFace operator\*(TVFace & ff);**

**Remarks:**

Returns a new TVFace with the FaceRemap applied.

# Class FaceChange

See Also: [Class Mesh](#), [Class Face](#), [Class Point3](#).

class FaceChange

## Description:

This class is available in release 3.0 and later only.

This class represents the notion of a mesh edit Face Change, which changes the visibility of the face's edges, its hidden state and/or its material ID.

## Data Members:

public:

### DWORD f

The face to change. This is a zero based index into the Mesh's **faces** array.

### DWORD flags

Specifies which attributes to change. One or more of the following values:

**ATTRIB\_EDGE\_A** - Alter edge between v0 and v1

**ATTRIB\_EDGE\_B** - Alter edge between v1 and v2

**ATTRIB\_EDGE\_C** - Alter edge between v2 and v0

**ATTRIB\_EDGE\_ALL** - Alter all the edges.

**ATTRIB\_HIDE\_FACE** - Alter the face hidden state.

**ATTRIB\_MATID** - Alter the material ID.

### DWORD val;

The value containing the face change information. Bits 0, 1, 2 hold the edge visibility, bit 3 holds the hidden state, and bits 5-21 hold the material ID.

## Methods:

public:

## Prototype:

**FaceChange();**

## Remarks:

Constructor. The **flags** and **f** are set to 0.

## Prototype:

**FaceChange(DWORD ff, DWORD fl, DWORD v);**

**Remarks:**

Constructor. The data members are initialized to the values passed.

**Parameters:**

**DWORD ff**

The index of the face to change. This is a zero based index into the Mesh's **faces** array.

**DWORD fl**

The flags to set.

**DWORD v**

The value to set.

**Prototype:**

**void Apply(Face &ff);**

**Remarks:**

Applies the face change to the Face passed using the flags of this FaceChange object to control what's altered.

**Parameters:**

**Face &ff**

The face to change.

**Prototype:**

**void Apply(FaceChange &fa);**

**Remarks:**

Updates the flags and val to the FaceChange passed using this FaceChange and its flags.

**Parameters:**

**FaceChange &fa**

The FaceChange object to alter.

## Class FaceSmooth

See Also: [Class Mesh](#), [Class Face](#), [Class Point3](#).

class FaceSmooth

### Description:

This class is available in release 3.0 and later only.

This class represents the notion of the edit mesh Face Smooth operation. This updates the smoothing group information in the face.

### Data Members:

public:

#### **DWORD f**

The face to change. This is a zero based index into the Mesh's **faces** array.

#### **DWORD mask**

The mask into the face of the smoothing groups.

#### **DWORD val;**

The smoothing group information to store.

### Methods:

public:

### Prototype:

**FaceSmooth();**

### Remarks:

Constructor. The **flags** and **f** are set to 0.

### Prototype:

**FaceSmooth(DWORD ff, DWORD mk, DWORD vl);**

### Remarks:

Constructor. The data members are initialized to the values passed.

### Prototype:

**void Apply(Face &ff);**

### Remarks:

Applies this smoothing change to the given face.

**Parameters:**

**Face &ff**

The face to update.

**Prototype:**

**void Apply(FaceSmooth &fs);**

**Remarks:**

Assigns the flags and val to the given FaceSmooth object from this one.

**Parameters:**

**FaceSmooth &fs**

The FaceSmooth object to alter.

## Class MeshDeltaUser

See Also: [Class Mesh](#), [Class MeshDelta](#).

class MeshDeltaUser : public InterfaceServer

### Description:

This class is available in release 3.0 and later only.

This is a virtual class which both Editable Mesh (class EditTriObject) and Edit Mesh (class EditMeshMod) subclass off of. It contains interface methods for editable meshes, much like the IMeshSelect class contains interface methods for modifiers and objects that can select sub-object parts of meshes.

### Methods:

public:

### Prototype:

**virtual void LocalDataChanged(DWORD parts)=0;**

### Remarks:

This method is used to notify the **MeshDeltaUser** that at least one of its associated **MeshDeltaUserDatas** has changed.

### Parameters:

#### **DWORD parts**

This represents the parts of the local data that have changed. One or more of the following values:

**PART\_TOPO, PART\_GEOM, PART\_TEXMAP, PART\_MTL,  
PART\_SELECT, PART\_SUBSEL\_TYPE, PART\_DISPLAY,  
PART\_VERTCOLOR, PART\_GFX\_DATA**

### Prototype:

**virtual void ToggleCommandMode(meshCommandMode mode)=0;**

### Remarks:

This method is used to start up one of the interactive command modes of the editable mesh.

### Parameters:

## **meshCommandMode mode**

The mode to start. One of the following values:

**McmCreate, McmAttach, McmExtrude, McmBevel,  
McmChamfer, McmSlicePlane, McmCut, McmWeldTarget,  
McmFlipNormalMode, McmDivide, McmTurnEdge**

### **Prototype:**

**virtual void ButtonOp(meshButtonOp opcode)=0;**

### **Remarks:**

This method performs the equivalent operation as a button press in the editable mesh UI.

### **Parameters:**

**meshButtonOp opcode**

One of the following values:

**MopHide, MopUnhideAll, MopDelete, MopDetach, MopBreak,  
MopViewAlign, MopGridAlign, MopMakePlanar, MopCollapse,  
MopTessellate, MopExplode, MopSlice, MopWeld,  
MopShowNormal, MopAutoSmooth, MopRemoveIsolatedVerts,  
MopSelectOpenEdges, MopCreateShapeFromEdges,  
MopFlipNormal, MopUnifyNormal, MopVisibleEdge,  
MopInvisibleEdge, MopAutoEdge, MopAttachList,  
MopSelectByID, MopSelectBySG, MopClearAllSG,  
MopSelectByColor**

### **Prototype:**

**virtual void GetUIParam(meshUIParam uiCode, int &ret);**

### **Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit spline parameters from the command panel. Currently not in use.

### **Parameters:**

**meshUIParam uiCode**

One of the following values;

**MuiSelByVert, MuiIgBack, MuiIgnoreVis, MuiSoftSel,  
MuiSSUseEDist, MuiSSEDist, MuiSSBack, MuiWeldBoxSize,  
MuiExtrudeType, MuiShowVNormals, MuiShowFNormals,  
MuiPolyThresh, MuiFalloff, MuiPinch, MuiBubble,  
MuiWeldDist, MuiNormalSize**

**int &ret**

The returned value.

**Default Implementation:**

**{ }**

**Prototype:**

**virtual void SetUIParam(meshUIParam uiCode, int val);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit spline parameters from the command panel. Currently not in use.

**Parameters:**

**meshUIParam uiCode**

One of the following values;

**MuiSelByVert, MuiIgBack, MuiIgnoreVis, MuiSoftSel,  
MuiSSUseEDist, MuiSSEDist, MuiSSBack, MuiWeldBoxSize,  
MuiExtrudeType, MuiShowVNormals, MuiShowFNormals,  
MuiPolyThresh, MuiFalloff, MuiPinch, MuiBubble,  
MuiWeldDist, MuiNormalSize**

**int val**

The value to set.

**Default Implementation:**

**{ }**

**Prototype:**

**virtual void GetUIParam(meshUIParam uiCode, float &ret);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to get the edit spline parameters from the command panel. Currently not in use.

**Parameters:****meshUIParam uiCode**

One of the following values;

**MuiSelByVert, MuiIgBack, MuiIgnoreVis, MuiSoftSel,  
MuiSSUseEDist, MuiSSEDist, MuiSSBack, MuiWeldBoxSize,  
MuiExtrudeType, MuiShowVNormals, MuiShowFNormals,  
MuiPolyThresh, MuiFalloff, MuiPinch, MuiBubble,  
MuiWeldDist, MuiNormalSize**

**float &ret**

The returned value.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void SetUIParam(meshUIParam uiCode, float val);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the edit spline parameters from the command panel. Currently not in use.

**Parameters:****meshUIParam uiCode**

One of the following values;

**MuiSelByVert, MuiIgBack, MuiIgnoreVis, MuiSoftSel,  
MuiSSUseEDist, MuiSSEDist, MuiSSBack, MuiWeldBoxSize,  
MuiExtrudeType, MuiShowVNormals, MuiShowFNormals,  
MuiPolyThresh, MuiFalloff, MuiPinch, MuiBubble,  
MuiWeldDist, MuiNormalSize**

**float val**

The value to set.

**Default Implementation:**

**{ }**

## Class MeshDeltaUserData

See Also: [Class Mesh](#), [Class MeshDelta](#), [Class MeshDeltaUser](#), [Class LocalModData](#), [Class Matrix3](#), [Class Point3](#), [Class Quat](#), [Class BitArray](#).

class MeshDeltaUserData

### Description:

This class is available in release 3.0 and later only.

This class provides a standard interface for modifiers and objects that use mesh deltas -- specifically Edit Mesh and Editable Mesh.

Both Edit Mesh and Editable Mesh have a current "state", which can be modified by MeshDeltas. In Editable Mesh, this "state" is an actual mesh, while in Edit Mesh, this is one **MeshDelta** per **LocalModData**. This class provides a standard interface to these: "MeshDeltaUser" and "MeshDeltaUserData".

### Methods:

public:

### Prototype:

```
virtual void ApplyMeshDelta(MeshDelta &md, MeshDeltaUser *mdu, TimeValue t)=0;
```

### Remarks:

This method applies the MeshDelta. Everything that happens in Edit Mesh and Editable Mesh goes through this method. Note that in Edit Mesh, the MeshDeltaUserData (EditMeshData) is separate from the MeshDeltaUser (EditMeshMod), though in Editable Mesh, EditTriObject subclasses from both of them.

There's essentially one mesh that can be edited per **MeshDeltaUserData**, so **ApplyMeshDelta** is the way to edit that mesh. ApplyMeshDelta typically handles adding Restore objects (if theHold.Holding()), clearing out any temporary local caches that are invalidated, and notifying the pipeline that the mesh has changed.

### Parameters:

**MeshDelta &md**

The mesh delta to apply.

**MeshDeltaUser \*mdu**

Points to the mesh delta user.

**TimeValue t**

The time to apply the mesh delta.

**Prototype:**

**virtual MeshDelta \*GetCurrentMDState();**

**Remarks:**

Returns a pointer to the MeshDelta object for this application of the Edit Mesh modifier. This is only non-NULL in Edit Mesh.

**Default Implementation:**

**{ return NULL; }**

**The following global function is not part of this class but is available for use:**

**Function:**

**void FindTriangulation(Mesh & m, int deg, int \*vv, int \*tri);**

**Remarks:**

This global function is available in release 3.0 and later only.

Finds a triangulation of an n-sided polygon using vertices in the specified mesh. As long as the vertices are coplanar, this algorithm will find a proper triangulation, even for nonconvex polygons.

**Parameters:**

**Mesh &m**

The mesh containing the vertices used in the polygon.

**int deg**

The size of the polygon.

**int \*vv**

The vertex indices of the polygon, in order around the perimeter. For instance, if deg is 5 and w points to an array containing (3, 6, 8, 0, 7), the polygon is presumed to have the outline described by m.verts[3], m.verts[6], m.verts[8], m.verts[0], and m.verts[7].

**int \*tri**

This is where the output is placed. Note that this should point to an array of size at least  $(deg-2)*3$ , to hold all the triangles. The values placed in this array are indices into the  $w$  array -- that is, given a 5-sided polygon, one triangle in this list might be (0,2,3), indicating you should use the 0th, 2nd, and 3rd elements of  $w$  to form the triangle. Put another way, to make a face from the  $n$ 'th triangle given by this array, you would set:

**$f.v[0] = w[tri[n*3+0]];$**

**$f.v[1] = w[tri[n*3+1]];$**

**$f.v[2] = w[tri[n*3+2]];$**

## Class VDataDelta

See Also: [Class Mesh](#), [Class PerData](#), [Class MeshDelta](#).

class VDataDelta

### Description:

This class is available in release 3.0 and later only.

VDataDelta is a way for a MeshDelta to keep track of per-vertex information. Like vertex selection, this information is explicitly set after all topological changes performed by the MeshDelta, so any active vertex data channels are always set to the size **outVNum()** for the related **MeshDelta**.

An example of this being used is in Edit Mesh, where you can assign or modify vertex weights. All these changes and assignments are stored in the Edit Mesh's main MeshDelta.

### Data Members:

public:

**PerData \*out;**

This is where the output vertex data is kept, if the given vertex data channel has been activated.

### Methods:

public:

### Prototype:

**VDataDelta();**

### Remarks:

Constructor. The data member **out** is set to NULL.

### Prototype:

**~VDataDelta();**

### Remarks:

Destructor. If **out** is allocated it is deleted.

### Prototype:

**void SetVNum(int nv, BOOL keep=FALSE);**

**Remarks:**

Sets the size of the output vertex data to the value specified.

**Parameters:**

**int nv**

The number of elements to allocate.

**BOOL keep=FALSE**

If TRUE previous values are kept (copied to the new storage); otherwise they are discarded.

**Prototype:**

**void Activate(int vnum, int vdID);**

**Remarks:**

If the output vertex data has not been allocated this method allocates it with the specified number of elements.

**Parameters:**

**int vnum**

The number of elements to allocate.

**int vdID**

The channel to allocate. One of the following values:

**VDATA\_SELECT**

**VDATA\_WEIGHT**

**Prototype:**

**void Set(int where, void \*data, int num=1);**

**Remarks:**

Sets the specified number of elements of vertex data at the location passed.

**Parameters:**

**int where**

The zero based index of the destination in the **out** array.

**void \*data**

The source data.

**int num=1**

The number of elements to set.

# Class MapDelta

See Also: [Class Mesh](#), [Class MeshDelta](#), [Template Class Tab](#).

class MapDelta

## Description:

This class is available in release 3.0 and later only.

This is an SDK class that represents some kind of change to a mesh map. This "delta" can include changes in map vertices and/or faces. It is always a subordinate part of a MeshDelta. Most of the time, the programmer does not need to worry about this class directly, but can let the parent MeshDelta do most of the work.

Note: You must **#include "MESHDLIB.H"** to use this class as it's not included by default by **MAX.H**.

## Methods Groups:

The hyperlinks below take you to the start of groups of related methods within the class:

[Initialization & Cleanup](#)

[Mesh Map Interaction](#)

[Composition and Operators](#)

[Characteristics](#)

[Basic Operations](#)

[Debugging](#)

## Data Members:

public:

**DWORD vnum**

The expected number of vertices in the input mesh

**DWORD fnum;**

The expected number of faces in the input mesh

**Tab<UVVertSet> vSet;**

This data member stores changes in the mapping vertices given as input. See class UVVertSet for more information. UVVertSets are stored in original map vertex ID order, and there is never more than one UVVertSet per original map vertex.

**Tab<Point3> vCreate;**

This data member stores mapping vertices newly created as part of the MapDelta. These are stored in the order created.

**Tab<TVFace> fCreate;**

This data member stores map faces newly created as part of the MapDelta. These are stored in the order created.

**Tab<FaceRemap> fRemap;**

This data member stores changes in which map vertices are used by existing map faces. See [Class FaceRemap](#) for more information. These are stored in original face order, and there is never more than one per original face.

**Methods:**

public:

**Initialization & Cleanup****Prototype:****MapDelta();****Remarks:**

Constructor. Initializes the MapDelta with empty tables and 0's for input mesh map size.

**Prototype:****void ClearAllOps(const Mesh & m);****Remarks:**

Clears out all existing map changes. Zeroes all the vCreate, vSet, etc tables.

**Prototype:****void SetInFNum(DWORD n);****Remarks:**

Sets the number of map faces in the input map. NOTE that if n is less than the current fnum, the data relating to the extra faces will be lost. (That is, if one of your face remaps is applied to face 32, and you SetInFNum to 30, that face remap will be lost, and will not be recovered if you later SetInFNum to 35.) It

is NOT necessary to call this method before applying this MapDelta to a smaller than expected Map.

**Parameters:**

**DWORD n**

The number of faces expected from the input mesh.

**Prototype:**

**void SetInVNum(DWORD n);**

**Remarks:**

Sets the number of map vertices in the input map. NOTE that if n is less than the current vnum, the data relating to the extra vertices will be lost. (That is, if one of your map vertex sets is applied to map vertex 32, and you SetInVNum to 30, that vertex set will be lost, and will not be recovered if you later SetInVNum to 35.) It is NOT necessary to call this method before applying this MapDelta to a smaller than expected Map.

**Parameters:**

**DWORD n**

The number of vertices expected from the input mesh.

## **Mesh Map Interaction**

**Prototype:**

**void Apply(UVVert \*tv, TVFace \*tf, DWORD inVNum, DWORD inFNum);**

**Remarks:**

Changes the given map by this MapDelta, in the following manner:

First, any maps that are supported by the MeshDelta but not by the mesh are assigned to the mesh in their default form. (Vertex colors are white, other maps are copies of the mesh vertices, and all have the same topology as the mesh.)

Next, any UNDEFINED mapping verts in the MeshDelta are filled in by FillInFaces.

Then the new vertices are added, creates first, followed by clones. The original vertices are then moved.

The faces are then modified, by applying all the FaceRemaps, FaceChanges, and FaceSmooths to the appropriate faces. New faces (in fCreate) are appended to the end of the face list.

Map changes are applied to all active maps, and map channels not supported by this MeshDelta are removed.

After all that is done, the vertices and faces marked in the vDelete and fDelete arrays are deleted.

Finally, the vertex data, vertex hide, and selections kept in the MeshDelta are applied to the result.

**Parameters:**

**UVVert \*tv**

The map vertex array to change. This should be allocated to handle all the new map vertices in the vCreate array.

**TVFace \*tf**

The map face array to change. This should be allocated to handle all the new map faces in the fCreate array.

**DWORD inVNum**

The actual number of map vertices (which doesn't have to match this MapDelta's vnum) in the input map.

**DWORD inFNum**

The actual number of map faces (which doesn't have to match this MapDelta's fnum) in the input map.

## **Composition & Operators**

See the class MeshDelta section on composition and operators for more information.

**Prototype:**

**MapDelta & operator\*=(MapDelta & from);**

**Remarks:**

Appends the given MapDelta to the current one.

**Parameters:**

**MapDelta & td**

The MapDelta to append. This MapDelta may be modified to make it suitable,

ie the vnum and fnum values will be set to the expected output of the current MapDelta if they don't already match. (This may result in the loss of some data – see "SetInVNum" and "SetInFNum" for more information.)

**Prototype:**

**MapDelta & operator=(MapDelta & td);**

**Remarks:**

Equality operator – makes this MapDelta just like the one given.

**Prototype:**

**DWORD ChangeFlags();**

**Remarks:**

Indicates what parts of a MapDelta could be changed if this MapDelta were appended to it. This is useful when backing up the MapDelta for Restore Objects. For instance, if you had a MapDelta with lots of face remaps, and you wanted to compose it with one that only added map vertices, there would be no reason to back up the remaps for an undo.

**Return Value:**

Returns some combination of the following flags, corresponding to the data members that would be changed:

**MDELTA\_VMOVE:** Indicates that the vSet array will be altered by this MapDelta.

**MDELTA\_VCREATE:** Indicates that the vCreate array will be altered by this MapDelta.

**MDELTA\_FREMAP:** Indicates that the fRemap array will be altered by this MapDelta.

**MDELTA\_FCREATE:** Indicates that the fCreate array will be altered by this MapDelta.

**Prototype:**

**void CopyMDChannels(MapDelta & from, DWORD channels);**

**Remarks:**

Copies the specified parts of the MapDelta. (Useful in combination with

ChangeFlags to create efficient Restore objects.)

**Parameters:**

**MapDelta & from**

The MapDelta to copy into this.

**DWORD channels**

Indicates the parts to copy – some combination of the following flags:

**MDELTA\_VMOVE:** Copy the vSet array.

**MDELTA\_VCREATE:** Copy the vCreate array.

**MDELTA\_FREMAP:** Copy the fRemap array.

**MDELTA\_FCREATE:** Copy the fCreate array.

**Characteristics**

The following methods give useful information about the MeshDelta.

**Prototype:**

**DWORD NumVSet(DWORD inVNum);**

**Remarks:**

Returns the number of map vertex sets that would be applied to a map with the specified number of map vertices. (If that number equals this MapDelta's vnum, this is simply vSet.Count().)

**Parameters:**

**DWORD inVNum**

The number of vertices in the input map we're inquiring about.

**Prototype:**

**DWORD outVNum();**

**Remarks:**

Returns the number of vertices in the output map, assuming that the input map is of the expected (vnum) size.

**Prototype:**

**DWORD outVNum(int inVNum);**

**Remarks:**

Returns the number of vertices in the output map, assuming that the input map has the specified number of map vertices.

**Parameters:**

**DWORD inVNum**

The number of map vertices expected in the input map.

**Prototype:**

**TVFace OutFace(TVFace \*mf, DWORD ff);**

**Remarks:**

Returns the specified map face as it would appear in the MapDelta output, taking into account any remaps.

**Parameters:**

**TVFace \*mf**

The input map face array.

**DWORD f**

The index of the face you want the output version of.

**Prototype:**

**DWORD RemapID(DWORD ff);**

**Remarks:**

Obtains the index of the fRemap entry that relates to this face.

**Parameters:**

**DWORD ff**

The map face index.

**Return Value:**

If there is such an entry, the index is returned, so fRemap[RemapID(ff)].fid == ff. If there is no remap record for this map face, the method returns UNDEFINED.

**Prototype:**

**DWORD IsRemapped(DWORD ff, DWORD vid);**

**Remarks:**

Tells whether the specified corner of the specified face has been remapped in this MapDelta.

**Parameters:****DWORD ff**

The map face index.

**DWORD vid**

The corner of the face – 0, 1, or 2.

**Return Value:**

If this corner has been remapped, it returns the vertex it's been remapped to. Otherwise, it returns UNDEFINED.

**Prototype:**

**DWORD SetID(DWORD i);**

**Remarks:**

Obtains the index of the vSet entry that relates to this vertex.

**Parameters:****DWORD i**

The map vertex index.

**Return Value:**

The index in the vSet array of the map vertex set corresponding to this vertex, or UNDEFINED if this map vertex has not been modified.

**Prototype:**

**bool IsCreate(DWORD i);**

**Remarks:**

Indicates whether the specified map vertex was created in this MapDelta.

**Parameters:****DWORD i**

The index in the output of the map vertex.

**Basic Operations**

These operations are the "building blocks" of MeshDeltas. All of them may be safely performed on MeshDeltas that are already quite complex. Those that accept DWORD indices require **output** mesh indices, as all operations are appended to the end of the existing delta.

**Prototype:**

**void Set(DWORD i, const UVVert & p);**

**Remarks:**

Sets an existing map vertex to the value given. (Note that if the same map vertex is set twice, the new set simply replaces the old one – there is never more than one UVVertSet in the vSet array for a single input map vertex.)

**Parameters:**

**int i**

The index of the map vertex to set.

**const UVVert & p**

The value to set the map vertex to.

**Prototype:**

**void Move(BitArray & sel, const UVVert & p);**

**Remarks:**

Sets the specified vertices to the value given. (Note that if the same map vertex is set twice, the new set simply replaces the old one – there is never more than one UVVertSet in the vSet array for a single input map vertex.)

**Parameters:**

**BitArray & sel**

Indicates which map vertices should be set.

**const UVVert & p**

The value to set the map vertex to.

**Prototype:**

**DWORD VCreate(UVVert \*v, int num=1);**

**Remarks:**

Creates new map vertices.

**Parameters:****UVVert \*v**

A pointer to an array of UVVerts representing the new map vertices.

**int num**

The size of the UVVert array.

**Return Value:**

The index (in the output map) of the first of these new map vertices.

**Prototype:****void FCreate(TVFace \*f, int num=1);****Remarks:**

Creates new map faces.

Note: MapDeltas must be kept up to date with the parent MeshDelta in all new face creations. See the MeshDelta method CreateDefaultMapFaces for details.

**Parameters:****TVFace \*f**

A pointer to an array of map faces to be added to the MapDelta.

**int num**

The size of the map face array.

**Prototype:****void FCreateDefault(int num=1);****Remarks:**

Creates new "default" map faces, where all the corners are UNDEFINED. (These are later filled in by a call to the parent MeshDelta's FillInFaces method.)

Note: MapDeltas must be kept up to date with the parent MeshDelta in all new face creations. See the MeshDelta method CreateDefaultMapFaces for details.

**Parameters:****int num**

The number of default faces to create.

**Prototype:**

```
void FCreateQuad(DWORD *t);
```

**Remarks:**

Creates 2 new faces, forming a quad.

Note: MapDeltas must be kept up to date with the parent MeshDelta in all new face creations. See the MeshDelta method CreateDefaultMapFaces for details.

**Parameters:**

```
DWORD *t
```

A pointer to an array of 4 map vertices to be used as corners of the quad.

**Prototype:**

```
DWORD FClone(TVFace & tf, DWORD remapFlags=0, DWORD *v=NULL);
```

**Remarks:**

Creates a new map face by copying (and optionally remapping) the face given.

Note: MapDeltas must be kept up to date with the parent MeshDelta in all new face creations. See the MeshDelta method CreateDefaultMapFaces for details.

**Parameters:**

```
TVFace & tf
```

The map face we wish to clone. (This is typically generated by the OutFace method.)

```
DWORD remapFlags=0
```

```
DWORD *v=NULL
```

If we wish to remap any of the corners of this map face while cloning, the appropriate flags and vertices should be passed in these last two arguments. v should point to an array of 3 map vertex indices, although the ones not marked as used by the remapFlags need not be set to anything in particular. See class FaceRemap for more information about face remapping.

**Prototype:**

```
void FRemap(FaceRemap & fr);
```

**Remarks:**

Adds a face remap to this MapDelta. If the face specified in the FaceRemap

already has a remap record, the two are combined. If the face specified is a face created by this MapDelta, the remap is applied directly to the fCreate entry instead of being stored in fRemap.

**Parameters:**

**FaceRemap & fr**

A FaceRemap that should be appended to this MapDelta.

**Prototype:**

**void FRemap(DWORD f, DWORD flags, DWORD \*v);**

**Remarks:**

Adds a face remap to this MapDelta. If the face specified already has a remap record, the two are combined. If the face specified is a face created by this MapDelta, the remap is applied directly to the fCreate entry instead of being stored in fRemap.

**Parameters:**

**DWORD f**

The face to remap.

**DWORD flags**

Face Remap flags – these indicate which vertices should be remapped. The possibilities are FR\_V0 (1), FR\_V1 (2), and FR\_V2 (4). (See class FaceRemap for more information.)

**DWORD \*v**

A pointer to the vertices to remap the face to use. Only the positions indicated in the remap flags need contain meaningful data.

Note that the vertices indicated here must be indexed by their positions after all of the current MeshDelta's creates and clones, but before any vertex deletes – essentially input-based indexing. Vertex index values of 0 through vnum-1 are considered to be the original mesh's vertices; values of vnum through vnum+vCreate.Count()-1 are considered to be this MeshDelta's newly created vertices; and values above this are cloned vertices.

**Prototype:**

**void FDelete(int offset, BitArray & fdel);**

**Remarks:**

Deletes the specified faces. This only affects the fCreate array, and should generally only be called by the parent MeshDelta's FDelete method to keep the face create arrays in sync.

**Parameters:****int offset**

Indicates what position in the fdel array corresponds to the first created face. (This is necessary since the BitArrays handed to MeshDelta::FDelete generally are based on the indexing after the previous MeshDelta::fDelete is applied. So this value is less than fnum if there was some previous deletion of original faces in the MeshDelta.)

**BitArray & fdel**

The faces to delete. The faces are indexed by their output mesh positions.

**Debugging****Prototype:**

```
void MyDebugPrint();
```

**Remarks:**

Prints out all the changes in this MapDelta to the DebugPrint window in Developer Studio.

## Class MeshTempData

See Also: [Class Mesh](#), [Class AdjEdgeList](#), [Class FaceClusterList](#), [Template Class Tab](#).

```
class MeshTempData : public BaseInterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This is a class for caching winged edge lists, face adjacency lists, face and edge clusters, vertex normals, and other derived data about a mesh.

There is a **SetMesh()** method to set the current mesh that the TempData is based on, then there's a series of methods to update the cache and return some sort of derived data. All of these methods follow the form:

### **DerivedData \*MeshTempData::DData (parameters);**

DerivedData is the container for the derived data requested (often a simple table, though there are some specialized classes returned from some methods). If the data has already been computed, the parameters are ignored and the cached data is returned. Otherwise, the data is computed from the parameters and the current mesh.

There are no procedures in place to detect changes in parameters or the mesh since the last time a method was called, so it's the calling routine's responsibility to free invalid structures. If you know that only certain pipeline channel, such as **GEOM\_CHANNEL**, have changed, you can use the **Invalidate(DWORD partsChanged)** method. (**GEOM\_CHANNEL** would free the distances-to-selected-vertices, for example, but not the **Adjacent Edge List**.)

In particular, there is no way for the **MeshTempData** to know when its mesh pointer is no longer valid, so it's vital that the calling routine clear the mesh (with **SetMesh(NULL)**) or stop using the **MeshTempData** when this happens.

All data members are private. They basically consist of a series of pointers which are initialized to NULL and then filled with allocated derived data as requested. There is also a NULL-initialized, private mesh pointer which is set with **SetMesh()**.

Editable Mesh and Edit Mesh both use this class to hold all the varieties of temporary, cached data they create -- examples are vertex normals and face clusters. This is called "ETTempData" in Editable Mesh and "EMTempData" in

Edit Mesh.

To use MeshTempData, just set it to your mesh and start asking for stuff:

```
MyAlgorithm (Mesh *m) {  
  MeshTempData mtd(m);  
  // Get Adjacent Edge List.  
  AdjEdgeList ae = mtd.AdjEList ();  
}
```

### **Methods Groups:**

The hyperlinks below take you to the start of groups of related methods within the class:

[Initialization and Class methods](#)

[Methods to Get Data](#)

[Data invalidation methods](#)

### **Data Members:**

public:

### **Initialization and Class methods**

#### **Prototype:**

```
MeshTempData();
```

#### **Remarks:**

Constructor. Sets all data members to NULL.

#### **Prototype:**

```
MeshTempData(Mesh *m);
```

#### **Remarks:**

Constructor. Sets the internal mesh pointer to the mesh passed.

#### **Parameters:**

```
Mesh *m
```

The mesh to set.

#### **Prototype:**

**~MeshTempData();**

**Remarks:**

Destructor. Frees all cached data.

**Prototype:**

**void SetMesh(Mesh \*m);**

**Remarks:**

Sets the internal mesh pointer to **m**.

**Parameters:**

**Mesh \*m**

Points to the mesh to set.

## **Methods to Get Data**

**Prototype:**

**AdjEdgeList \*AdjEList();**

**Remarks:**

Returns an adjacent edge list. See class **AdjEdgeList** for more information. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh.

**Prototype:**

**AdjFaceList \*AdjFList();**

**Remarks:**

Returns an adjacent face list. See class **AdjFaceList** for more information. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh.

**Prototype:**

**FaceClusterList \*FaceClusters();**

**Remarks:**

Returns a face cluster list, which groups selected faces into "clusters" for

transformation. See class **FaceClusterList** for more information. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh.

**Prototype:**

**EdgeClusterList \*EdgeClusters();**

**Remarks:**

Returns an edge cluster list, which groups selected edges into "clusters" for transformation. See class **EdgeClusterList** for more information. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh.

**Prototype:**

**Tab<DWORD> \*VertexClusters(DWORD sl);**

**Remarks:**

Returns an index of which cluster, if any, each vertex is in. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameter.

**Parameters:**

**DWORD sl**

Selection level. This should be either **MESH\_EDGE** or **MESH\_FACE**, to indicate whether the vertex cluster information should be based on edge or face clusters. Note that this parameter is ignored if there's already a vertex cluster cache.

**Return Value:**

A table of DWORD's is returned, one for each vertex. If (**\*VertexClusters(sl)**) [i] is UNDEFINED, vertex i is not in any cluster. Otherwise, the value for vertex i is the cluster index.

**Prototype:**

**Tab<Point3> \*ClusterNormals(DWORD sl);**

**Remarks:**

Returns average normals for each cluster. If cached, the cache is returned.

Otherwise a cache is allocated and computed from the current mesh and the parameter. Note that cluster centers and normals are computed and cached at the same time, when you call either method.

**Parameters:**

**DWORD sl**

Selection level. This should be either MESH\_EDGE or MESH\_FACE, to indicate whether the clusters we're talking about are the edge or face clusters. Note that this parameter is ignored if there's already a cluster normal cache.

**Return Value:**

A table of Point3's is returned, one for each cluster. The values are already normalized to length 1.

**Prototype:**

**Tab<Point3> \*ClusterCenters(DWORD sl);**

**Remarks:**

Returns mean centers for each cluster. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameter. Note that cluster centers and normals are computed and cached at the same time, when you call either method.

**Parameters:**

**DWORD sl**

Selection level. This should be either MESH\_EDGE or MESH\_FACE, to indicate whether the clusters we're talking about are the edge or face clusters. Note that this parameter is ignored if there's already a cluster center cache.

**Return Value:**

A table of Point3's is returned, one for each cluster.

**Prototype:**

**Matrix3 ClusterTM(int clust);**

**Remarks:**

Uses the current cluster center and normal caches to return the "objectspace to clusterspace" transform. This is the transform of the "local" axis in moving edge or face clusters in Edit(able) Mesh. If the cluster centers & normals have

not been cached, the identity matrix is returned; thus the control over whether this is an edge or face cluster is handled by the last call to ClusterCenters or ClusterNormals.

**Parameters:**

**int clust**

The cluster you want the transform for.

**Prototype:**

**Tab<Point3> \*VertexNormals();**

**Remarks:**

Returns a table of local average normals for vertices. This is equivalent to the average normals computed by the standalone function:

**void AverageVertexNormals(Mesh & mesh, Tab<Point3> & vnormals)**

If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh.

**Prototype:**

**Tab<float> \*VSWeight(BOOL useEdgeDist, int edgeIts, BOOL ignoreBack, float falloff, float pinch, float bubble);**

**Remarks:**

Returns Vertex Selection weights (for affect region). If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters. Weights are based on an Affect Region type falloff from the current selection.

**Parameters:**

**BOOL useEdgeDist**

If useEdgeDist is TRUE, the distance between vertices is computed along edges. If FALSE, it's computed directly through space.

**int edgeIts**

This indicates the maximum number of edges the algorithm may travel along in finding the distance between vertices. (Maximum path length.)

WARNING: If useEdgeDist is FALSE, this is an n-squared algorithm: it

compares every vertex not in the cluster with every vertex in it. If `useEdgeDist` is `TRUE`, the time it takes is proportional to the number of verts in the cluster times `edgeIts`.

### **BOOL ignoreBack**

If `TRUE`, vertices with a normal (as computed in `VertexNormals`) that points more than 90 degrees away from the average normal of the selection are not given any partial selections. They're either 1 if selected or 0 otherwise.

### **float falloff**

The limit distance of the effect. If `distance > falloff`, the function will always return 0.

### **float pinch**

Use this to affect the tangency of the curve near `distance=0`. Positive values produce a pointed tip, with a negative slope at 0, while negative values produce a dimple, with positive slope.

### **float bubble**

Use this to change the curvature of the function. A value of 1.0 produces a half-dome. As you reduce this value, the sides of the dome slope more steeply. Negative values lower the base of the curve below 0.

### **Return Value:**

Returns a table of float values, one per vertex, that are 1.0 if the vertex is in the current selection, 0.0 if it's more than `falloff` distance (or more than `edgeIts` edges, if (`useEdgeDist`)), and **`AffectRegionFunction(*SelectionDist(useEdgeDist, edgeIts), falloff, pinch, bubble)`** otherwise.

### **Prototype:**

```
Tab<float> *SelectionDist(BOOL useEdgeDist, int edgeIts);
```

### **Remarks:**

Computes the current distance of each vertex from the current selection. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters. The term "Selected verts" below refers to the vertices that are selected in the mesh's current selection level. (See the **Mesh** method **GetTempSel** for details.)

### **Parameters:**

### **BOOL useEdgeDist**

If TRUE, the distance between vertices is computed along edges. If FALSE, it's computed directly through space.

### **int edgeIts**

This indicates the maximum number of edges the algorithm may travel along in finding the distance between vertices. (Maximum path length.).

WARNING: If **useEdgeDist** is FALSE, this is an n-squared algorithm: it compares every nonselected vertex with every selected one. If **useEdgeDist** is TRUE, the time it takes is proportional to the number of selected vertices times edgeIts.

### **Return Value:**

A table consisting of one float value per vertex. If this value is 0, the vertex is either selected or on top of a selected vertex. Otherwise it represents the distance to the closest selected vertex. If **useEdgeDist** is TRUE, values of -1.0 are returned for vertices with no edgeIts-length path to a selected vertex.

### **Prototype:**

```
Tab<float> *ClusterDist(DWORD sl, int clustId, BOOL useEdgeDist, int edgeIts);
```

### **Remarks:**

Computes the current distance of each vertex from the specified cluster. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters.

### **Parameters:**

#### **DWORD sl**

Indicates whether we should use edges (MESH\_EDGE) or faces (MESH\_FACE) to construct the clusters, if needed.

#### **int clustId**

The index of the cluster we're measuring distance from.

#### **BOOL useEdgeDist**

If useEdgeDist is TRUE, the distance between vertices is computed along edges. If FALSE, it's computed directly through space.

#### **int edgeIts**

This indicates the maximum number of edges the algorithm may travel along

in finding the distance between vertices. (Maximum path length.)

**WARNING:** If **useEdgeDist** is FALSE, this is an n-squared algorithm: it compares every vertex not in the cluster with every vertex in it. If useEdgeDist is TRUE, the time it takes is proportional to the number of verts in the cluster times edgeIts.

**Return Value:**

A table consisting of one float value per vertex. If this value is 0, the vertex is either selected or on top of a vertex in the cluster. Otherwise it represents the distance to the closest selected vertex. If **useEdgeDist** is TRUE, values of -1.0 are returned for vertices with no edgeIts-length path to a vertex in the cluster.

**Prototype:**

**Tab<Point3> \*EdgeExtDir(Tab<Point3> \*edir, int extrusionType);**

**Remarks:**

Returns the direction each vertex should be going, after a topological edge extrusion, to handle the geometric extrusion. This should be obtained after applying a **MeshDelta::ExtrudeEdges()** to the mesh to obtain valid results. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters.

**Parameters:**

**Tab<Point3> \*edir**

This should be the edge direction table filled out by MeshDelta::ExtrudeEdges. It is necessary.

**int extrusionType**

This is one of **MESH\_EXTRUDE\_CLUSTER** or **MESH\_EXTRUDE\_LOCAL**, to indicate whether vertices should move according to cluster or local face normals.

**Return Value:**

A table of Point3's, one per vertex, representing the direction each vertex should move for further extrusion. The size of each nonzero entry is set to 1.

**Prototype:**

**Tab<Point3> \*FaceExtDir(int extrusionType);**

**Remarks:**

Returns the direction each vertex should be going, after a topological face extrusion, to handle the geometric extrusion. This should be obtained after applying a MeshDelta::ExtrudeFaces to the mesh to obtain valid results. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters.

**Parameters:**

**int extrusionType**

This is one of **MESH\_EXTRUDE\_CLUSTER** or **MESH\_EXTRUDE\_LOCAL**, to indicate whether vertices should move according to cluster or local face normals.

**Return Value:**

A table of Point3's, one per vertex, representing the direction each vertex should move for further extrusion. The size of each nonzero entry is set to 1.

**Prototype:**

**Tab<Point3> \*CurrentExtDir();**

**Remarks:**

This computes nothing; it merely returns the current extrusion direction cache, if any. The extrusion direction is controlled by the first call to EdgeExtDir or FaceExtDir since the last invalidation. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters.

**Prototype:**

**Tab<Point3> \*OutlineDir(int extrusionType);**

**Remarks:**

This produces the "Outline" direction of all vertices, based on the current face selection. "Outlining" is the direction vertices move to move edges of the current face selection outward at a constant rate. They are not set to length 1, but rather to whatever "rate" best makes the outline edges movemost consistently, without changing their angles.

**Parameters:****int extrusionType**

This is one of MESH\_EXTRUDE\_CLUSTER or MESH\_EXTRUDE\_LOCAL, to indicate whether vertices should move according to cluster or local face normals.

**Prototype:**

**MeshChamferData \*ChamferData();**

**Remarks:**

Returns the cache of a ChamferData for use in the MeshDelta methods,

**void ChamferEdges (Mesh & m, BitArray eset, MeshChamferData &mcd, AdjEdgeList \*ae=NULL);**

**void ChamferMove (Mesh & m, MeshChamferData &mcd, float amount, AdjEdgeList \*ae=NULL);**

**void ChamferVertices (Mesh & m, BitArray vset, MeshChamferData &mcd, AdjEdgeList \*ae=NULL);**

Unlike other MeshTempData methods, this method makes no calculations based on the current mesh, but merely supplies a memory cache.

**Data Invalidation Methods:****Prototype:**

**void Invalidate(DWORD part);**

**Remarks:**

Invalidates all data based on the specified part of the mesh. In the following chart, the columns represent the channels GEOM\_CHANNEL (G), TOPO\_CHANNEL (T), SELECT\_CHANNEL (S), and SUBSEL\_TYPE\_CHANNEL (U).

X's indicate dependency of the specified data cache on the given channel.

**Method to get cache G T S U**

**AdjEList X**

**AdjFList X**

**FaceClusters X X**

**EdgeClusters X X**  
**VertexClusters X X X**  
**ClusterCenters X X X X**  
**ClusterNormals X X X X**  
**VertexNormals X X**  
**SelectionDist X X X X**  
**ClusterDist X X X X**  
**VSWeight X X X X**

The extrusion direction methods could also be said to be dependent on all four channels, but is currently handled separately in freeBevelInfo. ChamferData is handled in freeChamferData, and is not based on the cached mesh.

Sample use: Suppose you use a MeshDelta to modify a mesh, twice:

```
DoStuffToMesh (Mesh & m) {  
MeshTempData foo;  
foo.SetMesh (&m);  
MeshDelta md(m);  
md.Op1 (m, foo.AdjEList()); // insert op of choice here  
md.Apply (m);  
foo.Invalidate (md.PartsChanged ());  
md.ClearAllOps ();  
md.Op2 (m, foo.VSWeights ());  
md.Apply (m);  
foo.Invalidate (md.PartsChanged ());  
}
```

Only the parts of foo that are dependent on what was changed by the first meshdelta are freed. The other parts, if any, remain cached for further operations.

**Parameters:**

**DWORD part**

One or more of the following channels:

**GEOM\_CHANNEL, TOPO\_CHANNEL, SELECT\_CHANNEL,  
SUBSEL\_TYPE\_CHANNEL**

**Prototype:**

**void InvalidateDistances();**

**Remarks:**

Uncaches (frees) the distance dependent data returned by VSWeight, SelectionDist, and ClusterDist.

**Prototype:**

**void InvalidateAffectRegion();**

**Remarks:**

Frees the VSWeight data. This is useful, e.g., if the mesh has not changed, but you wish to change the falloff, pinch, or bubble parameters to get new vertex selection weights.

**Prototype:**

**void freeClusterDist();**

**Remarks:**

Mainly for internal use, this frees just the cluster distance data.

**Prototype:**

**void freeBevelInfo();**

**Remarks:**

Frees only the extrusion direction data.

**Prototype:**

**void freeChamferData();**

**Remarks:**

Frees only the chamfer data structure.

**Prototype:**

**void freeAll();**

**Remarks:**

Frees all cached data.

## Class MeshChamferData

See Also: [Class Mesh](#), [Class MeshDelta](#), [Template Class Tab](#).

class MeshChamferData

### Description:

This class is available in release 3.0 and later only.

This class contains all the data needed to move points as the user drags a chamfer. It's created by the topological change that happens at the start of the chamfer. It is used to maintain chamfer information between several **MeshDelta** methods. The strategy is this: The chamfer operation is divided into two parts, the topological change and a later geometric change. (This works well for EditableMesh, where the topology change is completed first, then apply a series of geometry changes as the user spins a spinner or drags a mouse. Each geometry change is undone before the next is applied, but the topology change only happens once.)

This class is filled in by the topological change with the "directions" for all the geometric and mapping vert changes:

### Data Members:

public:

**Tab<Point3> vdir;**

This table contains the directions of movement for each vertex, scaled in such a manner to produce a consistent chamfer.

**Tab<float> vmax;**

This table contains the limits of motion for each vertex - stopping the vertices at these limits will prevent them from crossing each other or over far edges.

**Tab<UVVert> \*mdir;**

For each active map channel **mp**, **mdir[mp]** represents the directions of movement of the map verts for that channel. (Map verts need to be moved as well, otherwise the maps get distorted.)

### Methods:

public:

### Prototype:

**MeshChamferData();**

**Remarks:**

Constructor. This sets **mdir** to NULL.

**Prototype:**

**MeshChamferData(const Mesh &m);**

**Remarks:**

Constructor.

**Parameters:**

**const Mesh &m**

The mesh to init this MeshChamferData object from, allocating the vertex and mapping vertex tables as appropriate.

**Prototype:**

**~MeshChamferData();**

**Remarks:**

Destructor. If **mdir** is allocated it is seleted.

**Prototype:**

**void InitToMesh(const Mesh &m);**

**Remarks:**

This method sets up a **MeshChamferData** based on a given mesh, allocating the vertex and mapping vertex tables as appropriate.

**Parameters:**

**const Mesh &m**

The Mesh to init from.

**Prototype:**

**void setNumVerts(int nv, bool keep=TRUE, int resizer=0);**

**Remarks:**

This method simply allocates the vdir and vmax tables, and initializes the new members of vmax to 0. This method can be applied to an existing **MeshChamferData** to reflect an increase in vertices by VClone or VCreate

operations.

**Parameters:**

**int nv**

The number of verts

**bool keep=TRUE**

TRUE to keep if resized; FALSE to discard.

**int resizer=0**

The number of elements the vdir and vmax tables are resized beyond their current size.

**The following methods are not part of the class, but are useful for debugging**

**Prototype:**

**DllExport void MeshChamferDataDebugPrint (MeshChamferData & mcd, int mapNum);**

**Description**

This function uses calls to DebugPrint() to output all the data in the specified MeshChamferData to the DebugPrint buffer during debug runs. It is available for programmers' use, providing easy access to MeshChamferData during development. It ought to be removed for release builds.

**Parameters:**

**MeshChamferData & mcd**

The MeshChamferData we want to investigate.

**int mapNum**

The number of map channels in the Mesh associated with this MeshChamferData. (For historical reasons, this information is not kept in the MeshChamferData class.) Generally this is retrieved with a call to Mesh::getNumMaps().

## Class EdgeClusterList

See Also: [Class Mesh](#).

class EdgeClusterList

### Description:

This class is available in release 3.0 and later only.

This is a list of edge "clusters" for a given mesh. A typical application would be in Edit(able) Mesh, where the user has selected a two separate groups of edges on different parts of the mesh and wants to extrude them both, or rotate both around their local centers. Each "cluster" is a contiguous group of selected edges. Like AdjEdgeLists and AdjFaceLists, this class is only defined in relation to some mesh.

Note: for construction of this list, an edge is considered selected on both sides if it's selected on either. If you select the diagonal on top of a box, you probably only selected one of (face 2, edge 2 = 8) or (face 3, edge 2 = 11). But edges 8 and 11 will both be in the same cluster.

### Data Members:

public:

#### **DWORDTab clust;**

The cluster IDs of all the edges -- this table has size mesh::numFaces\*3. clust[i] is UNDEFINED if edge i is not in any cluster (ie is totally unselected).

#### **DWORD count;**

The number of clusters.

### Methods:

public:

### Prototype:

**EdgeClusterList(Mesh &mesh, BitArray &esel, AdjEdgeList &adj);**

### Remarks:

Constructor.

Creates an edge cluster list from the current selection. All adjacent selected edges are grouped into the same cluster.

**Parameters:**

**Mesh &mesh**

The mesh associated with this EdgeClusterList.

**BitArray &esel**

The bit array containing the edge selection data for the mesh.

**AdjEdgeList &adj**

The adjacent edge list for the mesh.

**Prototype:**

**DWORD ID(int f, int e) {return clust[f\*3+e];};**

**Remarks:**

Returns the cluster ID for face f, edge e.

**Parameters:**

**int f**

The index of the face in the mesh.

**int e**

The index of the edge in the mesh.

**Prototype:**

**void MakeVertCluster(Mesh &mesh, Tab<DWORD> &vclust);**

**Remarks:**

Creates a list of cluster IDs for vertices.

**Parameters:**

**Mesh &mesh**

The mesh associated with this EdgeClusterList.

**Tab<DWORD> & vclust**

This is where the output goes: vclust is set to size mesh.numVerts, and the value of each entry in this table tells which cluster the vertex has been assigned to, based on the edges it's on. If vertex "v" is not in any clusters (ie none of the edges that use it are in any clusters), vclust[v] is UNDEFINED.

**Prototype:**

```
void GetNormalsCenters(Mesh &mesh, Tab<Point3> & norm,  
Tab<Point3> & ctr);
```

**Remarks:**

This method extracts normal and center information for the various clusters of the mesh.

**Parameters:**

**Mesh &mesh**

The mesh to evaluate.

**Tab<Point3> & norm**

This table has its sizes set to the number of clusters in the cluster list. Normals are computed as the normalized average of the area-normal vectors of all faces in the cluster.

**Tab<Point3> & ctr**

This table has its sizes set to the number of clusters in the cluster list. Centers are the average location of the face centers or edge centers -- thus a point on three faces or edges in the same cluster has more weight than a point on one face in the cluster.

**Prototype:**

```
DWORD operator[](int i);
```

**Remarks:**

Access operator. Returns the cluster ID for edge i (indexed as 3\*face+edge).

**The following functions are not part of a class but are available for use:**

**Function:**

```
float AffectRegionFunction(float dist, float falloff, float pinch, float  
bubble);
```

**Remarks:**

This function is available in release 3.0 and later only.

This is the standard affect region function, based on a distance and the three affect region parameters (same as the editable mesh).

This function is a cubic curve which returns 1 at distance 0, 0 if distance is greater than falloff, and other values for distance between 0 and falloff. To "see" this function graphed, look at the curve in the Soft Selection parameters in Edit Mesh, Editable Mesh, Mesh Select, or Volume Select. This function currently is constructed as follows:

```
float u = ((falloff - dist)/falloff);  
float u2 = u*u, s = 1.0f-u;  
return (3*u*bubble*s + 3*u2*(1.0f-pinch))*s + u*u2;"
```

#### **Parameters:**

##### **float dist**

The distance to the selection. The method for computing this distance is up to the developer; for example in Mesh Select, it's the distance to the nearest selected vertex, while in Volume Select (with a box or sphere selection region) it's the distance to the selection volume.

##### **float falloff**

The limit distance of the effect. If distance > falloff, the function will always return 0.

##### **float pinch**

Use this to affect the tangency of the curve near distance=0. Positive values produce a pointed tip, with a negative slope at 0, while negative values produce a dimple, with positive slope.

##### **float bubble**

Use this to change the curvature of the function. A value of 1.0 produces a half-dome. As you reduce this value, the sides of the dome slope more steeply. Negative values lower the base of the curve below 0.

#### **Return Value:**

Returns the strength of the Affect Region function at the given distance. (In selection modifiers, this is the "soft selection" amount, the amount it's considered selected. A vertex at a distance with a return value of .25, for instance, will be affected 1/4 as strongly in a deformation as a fully selected vertex.)

#### **Function:**

```
void MatrixFromNormal(Point3& normal, Matrix3& mat);
```

**Remarks:**

This function is available in release 3.0 and later only.

This function creates a matrix with the normal as a Z-axis. The X and Y axes are chosen arbitrarily.

**Parameters:****Point3& normal**

The input normal is specified here.

**Matrix3& mat**

The output matrix.

**Function:**

```
void AverageVertexNormals(Mesh & mesh, Tab<Point3> & vnormals);
```

**Remarks:**

This function is available in release 3.0 and later only.

This function creates vertex normals that are weighted averages of faces using each vertexSmoothing groups are not used in these computations -- the normals are those you would expect with a totally smooth mesh.

**Parameters:****Mesh & mesh**

The mesh whose average vertex normals are computed.

**Tab<Point3> & vnormals**

The output vertex normals. . This will be set to size **mesh.numVerts**.

**Function:**

```
Point3 AverageSelVertNormal(Mesh& mesh);
```

**Remarks:**

This function is available in release 3.0 and later only.

This function computes and returns the average normal of a group of selected vertices.

**Parameters:****Mesh& mesh**

The mesh to check. The function uses **mesh.vertSel** to check for selected verts.

**Function:**

**Point3 AverageSelVertCenter(Mesh& mesh);**

**Remarks:**

This function is available in release 3.0 and later only.

This function computes and returns the average center of a group of selected vertices.

**Parameters:**

**Mesh& mesh**

The mesh to check. The function uses **mesh.vertSel** to check for selected verts.

**Function:**

**void DeselectHiddenFaces(Mesh &mesh);**

**Remarks:**

This function is available in release 3.0 and later only.

Removes hidden faces from the **mesh.faceSel** selection array.

**Parameters:**

**Mesh& mesh**

The mesh to check.

**Function:**

**void DeselectHiddenEdges(Mesh &mesh);**

**Remarks:**

This function is available in release 3.0 and later only.

This function removes edges on hidden faces from the **mesh.edgeSel** selection array.

**Parameters:**

**Mesh& mesh**

The mesh to check.

**Function:**

**void HiddenFacesToVerts(Mesh &mesh, BitArray &alsoHide);**

**Remarks:**

This function is available in release 3.0 and later only.

This function hides vertices that are only used by hidden faces. If **alsoHide** has size **mesh.numVerts**, it is used to indicate other vertices that should also be hidden. Note that passing **mesh.vertHide** as **alsoHide** will NOT WORK, as mesh.vertHide is overwritten before alsoHide is read.

**Parameters:****Mesh &mesh**

The mesh to check.

**BitArray &alsoHide**

If specified, this is used to indicate other vertices that should also be hidden.

**Function:**

**void SelectionDistance(Mesh & mesh, float \*selDist);**

**Remarks:**

This function is available in release 3.0 and later only.

This function computes distances from selected vertices (as indicated by **mesh.VertexTempSel()**) to non-selected ones.

NOTE: This is an order-of-n-squared algorithm. Each unselected vert is compared with each selected vert. So, if you have a mesh with 1000 sel verts and 1000 unsel verts, this involves a million compares.

**Parameters:****Mesh & mesh**

The mesh to check.

**float \*selDist**

This is assumed to be a float array of size **mesh.numVerts**. It is set to -1 for all verts if there is no selection. Otherwise, selected vertices have a value of 0, and nonselected vertices have the distance to the nearest selected vertex.

**Function:**

```
void SelectionDistance(Mesh & mesh, float *selDist, int iters,  
AdjEdgeList *ae=NULL);
```

**Remarks:**

This function is available in release 3.0 and later only.

This function computes distances from selected vertices (as indicated by **mesh.VertexTempSel()**) to non-selected ones along edge paths. **selDist** is assumed to be a float array of size **mesh.numVerts**. **selDist** is set to -1 for all verts if there is no selection. Otherwise, selected vertices have **selDist** value 0; non-selected vertices that are "iters" or fewer edges away from a selected vertex are assigned the shortest edge-path distance to a selected vertex; and non-selected vertices that are more than **iters** edges away are set to -1. The **AdjEdgeList** is computed by the algorithm if the one passed is NULL; otherwise you can save time by passing a cached one in.

This is NOT an n-squared algorithm like the one above. It's more a sort of order-of-n-times-(iters-squared).

**Parameters:**

**Mesh & mesh**

The mesh to check.

**float \*selDist**

An array of floats of size **mesh.numVerts**.

**int iters**

If 0, Selection Distance is computed from each vertex to the nearest selected vertex, regardless of topology. This is a VERY EXPENSIVE ALGORITHM, which takes almost 4 times as long for twice as many vertices. If iters is non-zero, it represents the number of edges one should "travel" in trying to find the nearest selected vertex -- this means that it only takes twice as long for twice as many verts. (This is like the Edge Distance parameter in EMesh's Soft Selection dialog.) If iters is 0, ae is irrelevant and may be left as NULL. If iters is nonzero, an Adjacent Edge List is required, and will be computed internally from the mesh if ae is NULL. (If you've got an AdjEdgeList for this mesh handy, pass it in, otherwise don't worry about it.)

Note also that if iters is nonzero, the distance is computed along the edges, not directly through space. If there is no selected vertex within an iters-length path, a vertex is assigned a 0 selection value.

**AdjEdgeList \*ae=NULL**

The optional adjacent edge list.

**Function:**

**void ClustDistances(Mesh & mesh, DWORD numClusts, DWORD \*vclust, Tab<float> \*\*clustDist);**

**Remarks:**

This function is available in release 3.0 and later only.

Computes distances from nonselected vertices in the mesh to each of the vertex clusters. This is a VERY EXPENSIVE ALGORITHM, which takes almost 4 times as long for twice as many vertices.

Preparation for this method would typically look like:

```
// given Mesh msh, FaceClusterList fclust:  
Tab<DWORD> vclust;  
fclust.MakeVertCluster (msh, vclust);  
Tab<float> ** clustDist;  
clustDist = new (Tab<float> *)[fclust.count];  
for (int i=0; i<fclust.count; i++) clustDist[i] = new Tab<float>;  
ClustDistances (mesh, fclust.count, vclust.Addr(0), clustDist);
```

Then (\*clustDist[c])[v] would give the distance from vertex v in the mesh to the vertices of cluster c.

**Parameters:**

**Mesh &mesh**

The mesh the clusters are based on.

**DWORD numClusts**

The number of clusters in this mesh.

**DWORD \*vclust**

A pointer to an array of vertex cluster IDs. Typically this is a pointer to the data in the table created by EdgeClusterList::GetVertClusters or FaceClusterList::GetVertClusters.

**Tab<float> \*\*clustDist**

This is an array of <numClusts> pointers to tables that will be used to store distances from various clusters. Each table will be set to the size of

mesh.numVerts and filled with distances to the cluster that table represents.

**Function:**

```
void ClustDistances(Mesh & mesh, DWORD numClusts, DWORD  
*vclust, Tab<float> **clustDist, int iters, AdjEdgeList  
*ae=NULL);
```

**Remarks:**

This function is available in release 3.0 and later only.

Computes distances from nonselected vertices in the mesh to each of the vertex clusters. Unlike the other version which doesn't have an iters or ae parameter, this is a linear algorithm which computes distance along a finite number of edges.

Preparation for this method would typically look like:

```
// given Mesh msh, FaceClusterList fclust:  
Tab<DWORD> vclust;  
fclust.MakeVertCluster (msh, vclust);  
Tab<float> ** clustDist;  
clustDist = new (Tab<float> *)[fclust.count];  
for (int i=0; i<fclust.count; i++) clustDist[i] = new Tab<float>;  
ClustDistances(mesh, fclust.count, vclust.Addr(0), clustDist);
```

Then (\*clustDist[c])[v] would give the distance from vertex v in the mesh to the vertices of cluster c.

**Parameters:**

**Mesh & mesh**

The mesh the clusters are based on.

**DWORD numClusts**

The number of clusters in this mesh.

**DWORD \*vclust**

A pointer to an array of vertex cluster IDs. Typically this is a pointer to the data in the table created by EdgeClusterList::GetVertClusters or FaceClusterList::GetVertClusters.

**Tab<float> \*\*clustDist**

This is an array of <numClusts> pointers to tables that will be used to store

distances from various clusters. Each table will be set to the size of mesh.numVerts and filled with distances to the cluster that table represents.

**int iters**

The maximum number of edges to travel along looking for a vertex in the given cluster.

**AdjEdgeList \*ae=NULL**

Edge length computations require an adjacent edge list. If you don't pass one in this parameter, it'll have to construct its own from the mesh.

## Class UVWMapper

See Also: [Class Mesh](#), [Class Matrix3](#), [Class Point3](#).

class UVWMapper

### Description:

This class is available in release 3.0 and later only.

Prior to release 3.0, developers could implement **Object::ApplyUVWMap()** in their objects, but didn't have access to the algorithm 3ds max uses internally to turn the mapping types (MAP\_BOX, MAP\_PLANE, etc) into an actual vertex-to-mapping-coordinate function. This class now makes this available.

The constructors for the class initialize the data members with information about the mapping desired. The main method, **MapPoint()**, maps a point in object space into the UVW map defined by this mapper.

Note: **typedef Point3 UVVert;**

### Data Members:

public:

#### **int type;**

The mapping type. One of the following values:

**MAP\_PLANAR**

**MAP\_CYLINDRICAL**

**MAP\_SPHERICAL**

**MAP\_BALL**

**MAP\_BOX**

**MAP\_FACE**

#### **int cap;**

This is used with **MAP\_CYLINDRICAL**. If nonzero, then any face normal that is pointing more vertically than horizontally will be mapped using planar coordinates.

#### **float utile;**

Number of tiles in the U direction.

#### **float vtile;**

Number of tiles in the V direction.

**float wtile;**

Number of tiles in the W direction.

**int uflip;**

If nonzero the U values are mirrored.

**int vflip**

If nonzero the V values are mirrored.

**int wflip;**

If nonzero the W values are mirrored.

**Matrix3 tm;**

This defines the mapping space. As each point is mapped, it is multiplied by this matrix, and then it is mapped.

**Methods:**

public:

**Prototype:**

**UVWMapper();**

**Remarks:**

Constructor. The data members are initialized as follows:

**type = MAP\_BOX;**

**utile = 1.0f;**

**vtile = 1.0f;**

**wtile = 1.0f;**

**uflip = 0;**

**vflip = 0;**

**wflip = 0;**

**cap = 0;**

**tm.IdentityMatrix();**

**Prototype:**

**UVWMapper(int type, const Matrix3 &tm, int cap=FALSE, float  
utile=1.0f, float vtile=1.0f, float wtile=1.0f,int uflip=FALSE, int  
vflip=FALSE, int wflip=FALSE);**

**Remarks:**

Constructor. The data members are initialized to the values passed.

**Prototype:**

**UVWMapper(UVWMapper& m);**

**Remarks:**

Constructor. The data members are initialized from the UVWMapper passed.

**Prototype:**

**UVVert MapPoint(Point3 p, const Point3 & norm, int \*nan=NULL);**

**Remarks:**

This method maps a point in object space into the UVW map defined by this mapper. This gives the UVW coordinates for the specified point according to this mapper's mapping scheme.

**Parameters:****Point3 p**

The location of a vertex, i.e. the point being mapped. This point should NOT be transformed by the UVWMapper's tm, as this happens internally.

**const Point3 & norm**

The direction of the surface normal at p. This information is only required for types MAP\_BOX or MAP\_CYLINDRICAL. See the method **NormalMatters()** below.

**int \*nan=NULL**

If non-NULL, this points to an int which should be set to FALSE if this mapping is good for all faces using this vertex, or TRUE if different faces should have different mapping coordinates. This is generally set to TRUE more often than absolutely necessary to make sure nothing is missed.

**Return Value:**

The mapped point.

**Prototype:**

**UVVert TileFlip(UVVert uvw);**

**Remarks:**

Applies the UVWMap's tile and flip parameters to the given UVVert, and returns the result..

**Parameters:**

**UVVert uvw**

The input UVVert.

**Return Value:**

The modified UVVert.

**Prototype:**

**int MainAxis(const Point3 & n);**

**Remarks:**

This method indicates which direction the given vector "chiefly points", after vector transformation by the UVWMapper's transform.

**Parameters:**

**const Point3 & n**

The input vector whose main axis is determined.

**Return Value:**

One of the following values:

**0:** tm.VectorTransform(n) points mainly in the +x direction.

**1:** tm.VectorTransform(n) points mainly in the +y direction.

**2:** tm.VectorTransform(n) points mainly in the +z direction.

**3:** tm.VectorTransform(n) points mainly in the -x direction.

**4:** tm.VectorTransform(n) points mainly in the -y direction.

**5:** tm.VectorTransform(n) points mainly in the -z direction.

**Prototype:**

**bool NormalMatters();**

**Remarks:**

This method lets you know whether the current mapping type uses the normal information. If FALSE, it doesn't matter what value you pass as a normal to MapPoint. If TRUE, the MainAxis of the normal is used to determine the

mapping.

# Class IMeshSelect

See Also: [Class Mesh](#), [Class LocalModData](#), [Class IMeshSelectData](#).

class IMeshSelect

## Description:

This class is available in release 2.0 and later only.

This class provides access to selection data for the Mesh Select Modifier, Editable Mesh and Edit Mesh modifier.

To get a pointer to this interface given a pointer to a modifier or editable mesh object, use the following macro (defined in **ANIMTBL.H**). Using this macro, given any Animatable, it is easy to ask for the interface.

```
#define GetMeshSelectInterface(anim) ((IMeshSelect*)anim->GetInterface(I_MESHSELECT))
```

A plug-in developer may use this macro as follows:

```
IMeshSelect *ims = GetMeshSelectInterface(anim);
```

This return value will either be NULL or a pointer to a valid Mesh Select interface.

## Methods:

### Prototype:

```
virtual DWORD GetSelLevel()=0;
```

### Remarks:

Returns the current level of selection for the modifier.

### Return Value:

One of the following values:

**IMESHSEL\_OBJECT**: Object level.

**IMESHSEL\_VERTEX**: Vertex level.

**IMESHSEL\_FACE**: Face level.

**IMESHSEL\_EDGE**: Edge level.

### Prototype:

```
virtual void SetSelLevel(DWORD level)=0;
```

**Remarks:**

Sets the selection level of the modifier.

**Parameters:****DWORD level**

One of the following values:

**IMESHSEL\_OBJECT**: Object level.

**IMESHSEL\_VERTEX**: Vertex level.

**IMESHSEL\_FACE**: Face level.

**IMESHSEL\_EDGE**: Edge level.

**Prototype:**

**virtual void LocalDataChanged()=0;**

**Remarks:**

This method must be called when the **LocalModData** of the modifier is changed. Developers can use the methods of **IMeshSelectData** to get and set the actual selection for vertex, face and edge. When a developers does set any of these selection sets this method must be called when done.

**Prototype:**

**virtual BOOL HasWeightedVertSel();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if this modifier or object has weighted vertex selection data (Soft Selection data), FALSE if not.

**Default Implementation:**

**{ return FALSE; }**

**Prototype:**

**virtual BOOL CanAssignWeightedVertSel();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if this modifier or object can assign weighted vertex selection

data; FALSE if it cannot.

**Default Implementation:**

```
{ return FALSE; }
```

## Class IMeshSelectData

See Also: [Class ModContext](#), [Class BitArray](#), [Class IMeshSelect](#), [Class Interface](#), [Class GenericNamedSelSetList](#).

class IMeshSelectData

### Description:

This class is available in release 2.0 and later only.

When a developer gets the **LocalModData** from the **ModContext** of the Mesh Select Modifier or Edit Mesh Modifier, they may cast it to this class and use these methods. They may be used to get/set the vert/face/edge selection state of the modifier. This class also provides access to the named sub-object selection sets.

To get a pointer to this interface given a pointer to a modifier use the following macro (defined in **ANIMTBL.H** ). Using this macro, given any Animatable, it is easy to ask for the interface.

```
#define GetMeshSelectDataInterface(anim)  
((IMeshSelectData*)anim->GetInterface(I_MESHSELECTDATA))
```

A plug-in developer may use this macro as follows:

```
IMeshSelectData *imsd = GetMeshSelectDataInterface(anim);
```

This return value will either be NULL or a pointer to a valid Mesh Select Data interface.

### Methods:

#### Prototype:

```
virtual BitArray GetVertSel()=0;
```

#### Remarks:

Returns a **BitArray** that reflects the current vertex selection. There is one bit for each vertex. Bits that are 1 indicate the vertex is selected.

#### Prototype:

```
virtual BitArray GetFaceSel()=0;
```

#### Remarks:

Returns a **BitArray** that reflects the current face selection. There is one bit

for each face. Bits that are 1 indicate the face is selected.

**Prototype:**

**virtual BitArray GetEdgeSel()=0;**

**Remarks:**

Returns a **BitArray** that reflects the current edge selection. There is one bit for each edge. Bits that are 1 indicate the edge is selected.

**Prototype:**

**virtual void SetVertSel(BitArray &set, IMeshSelect \*imod, TimeValue t)=0;**

**Remarks:**

Sets the vertex selection of the modifier.

**Parameters:**

**BitArray &set**

There is one bit for each vertex. Bits that are 1 indicate the vertex is selected.

**IMeshSelect \*imod**

This parameter is available in release 3.0 and later only.

Points to the IMeshSelect instance (generally this is a modifier).

**TimeValue t**

This parameter is available in release 3.0 and later only.

The current time at which the call is made.

**Prototype:**

**virtual void SetFaceSel(BitArray &set, IMeshSelect \*imod, TimeValue t)=0;**

**Remarks:**

Sets the face selection of the modifier.

**Parameters:**

**BitArray &set**

There is one bit for each face. Bits that are 1 indicate the face is selected.

### **IMeshSelect \*imod**

This parameter is available in release 3.0 and later only.

Points to the IMeshSelect instance (generally this is a modifier).

### **TimeValue t**

This parameter is available in release 3.0 and later only.

The current time at which the call is made.

### **Prototype:**

```
virtual void SetEdgeSel(BitArray &set, IMeshSelect *imod,  
TimeValue t)=0;
```

### **Remarks:**

Sets the edge selection of the modifier.

### **Parameters:**

#### **BitArray &set**

There is one bit for each edge. Bits that are 1 indicate the edge is selected.

#### **IMeshSelect \*imod**

This parameter is available in release 3.0 and later only.

Points to the IMeshSelect instance (generally this is a modifier).

#### **TimeValue t**

This parameter is available in release 3.0 and later only.

The current time at which the call is made.

### **Prototype:**

```
virtual GenericNamedSelSetList &GetNamedVertSelList()=0;
```

### **Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to an instance of **GenericNamedSelSetList** used for storing vertex level named selection sets. This class provides access to and the ability to manipulate a list of named selection sets.

### **Prototype:**

```
virtual GenericNamedSelSetList &GetNamedEdgeSelList()=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to an instance of **GenericNamedSelSetList** used for storing edge level named selection sets. This class provides access to and the ability to manipulate a list of named selection sets.

**Prototype:**

```
virtual GenericNamedSelSetList &GetNamedFaceSelList()=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to an instance of **GenericNamedSelSetList** used for storing face level named selection sets. This class provides access to and the ability to manipulate a list of named selection sets.

**Prototype:**

```
virtual void GetWeightedVertSel(int nv, float *sel) {};
```

**Remarks:**

This method is available in release 3.0 and later only.

Retrieves the weighted vertex selections data (Soft Selection data).

**Parameters:**

**int nv**

The number of vertices.

**float \*sel**

An array of floats to store the results.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void SetWeightedVertSel(int nv, float *sel, IMeshSelect *imod, TimeValue t);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the weighted vertex selection data (Soft Selection data) to the values passed.

**Parameters:**

**int nv**

The number of vertices.

**float \*sel**

An array of floats with the data.

**IMeshSelect \*imod**

Points to the IMeshSelect object.

**TimeValue t**

The time at which to set the data.

**Default Implementation:**

```
{}
```

## Class View

See Also: [Class GeomObject](#), [Class Interface](#), [Class Control](#), [Class Matrix3](#), [Class Point3](#).

```
class View : public InterfaceServer
```

### Description:

This class is passed in to **GeomObject::GetRenderMesh()** to allow objects to do view dependent rendering. It is also passed to **Control::EvalVisibility()**. For example particle systems use this to have the particles exactly face the camera (if this option is enabled). If **GetRenderMesh()** is called by the renderer, the methods of this class are implemented by the system. If a plug-in is calling this method, they must implement these methods. The sample code below shown a null implementation that may be used if a viewport is not involved:

```
class NullView : public View {
public:
    Point2 ViewToScreen(Point3 p)
        { return Point2(p.x,p.y); }
    NullView() {
        worldToView.IdentityMatrix();
        screenW=640.0f; screenH = 480.0f;
    }
};
```

### Data Members:

```
public:
```

```
float screenW, screenH;
```

These hold the screen dimensions in pixels for width and height.

```
Matrix3 worldToView;
```

A transformation matrix from world into view space. This is into the camera's space.

```
int projType;
```

This data member is available in release 2.0 and later only.

The view projection type: **0** is perspective, **1** is parallel.

**float fov;**

This data member is available in release 2.0 and later only.

The field of view in radians.

**float pixelSize;**

This data member is available in release 2.0 and later only.

The pixel size setting.

**Matrix3 affineTM;**

This data member is available in release 2.0 and later only.

This is the world to camera transformation matrix.

**DWORD flags;**

This data member is available in release 3.0 and later only.

The following flag is defined.

**RENDER\_MESH\_DISPLACEMENT\_MAP**

Indicates that Displacement Mapping is enabled. Note that this flag should be tested, and not **Interface::GetRendDisplacement()**, because the values may not be the same (for instance when rendering in the Materials Editor).

**Methods:****Prototype:****virtual Point2 ViewToScreen(Point3 p)=0;****Remarks:**

This method is used to convert a point in view space to screen space. This includes any perspective projection.

**Parameters:****Point3 p**

The point in view space.

**Return Value:**

The point in screen space (in pixel coordinates).

**Prototype:****virtual BOOL CheckForRenderAbort();**

**Remarks:**

This method is available in release 3.0 and later only.

This method should be used by **GetRenderMesh()** implementations that require a lot of processing time. This allows these processes to be interrupted by the user. An example of this in use is the extensive computations done for displacement mapping. These may be interrupted by the user during a render.

So, any implementation of **GetRenderMesh()** which takes a long time should periodically call this method to see if the user has canceled the render

**Return Value:**

Returns TRUE iff user has cancelled; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

## Class SpecialFX

See Also: [Class ReferenceTarget](#), [Class SFXParamDlg](#), [Class IRendParams](#), [Class Atmospheric](#), [Class Effect](#), [Class AppendGizmoRestore](#), [Class DeleteGizmoRestore](#).

```
class SpecialFX : public ReferenceTarget
```

### Description:

This class is available in release 3.0 and later only.

This is the base class for Atmospheric, Renderer Effect, and Shader Plug-Ins. It contains a few methods common to each of those plug-in classes.

### Data Members:

public:

**TSTR name;**

This is the name which appears in Track View.

### Methods:

public:

### Prototype:

**virtual BOOL Active(TimeValue t);**

### Remarks:

Implemented by the Plug-In.

Returns TRUE if the effect is active; otherwise FALSE.

### Parameters:

**TimeValue t**

The time at which to check.

### Default Implementation:

```
{ return FALSE; }
```

### Prototype:

**virtual TSTR GetName();**

### Remarks:

Implemented by the Plug-In.

This method is used to retrieve the name for the plug-in. This name will appear in the track view and the list of current atmospheric or rendering effects.

**Prototype:**

**virtual void Update(TimeValue t, Interval& valid);**

**Remarks:**

Implemented by the Plug-In.

This method is called once per frame when the renderer begins. This gives the atmospheric or rendering effect the chance to cache any values it uses internally so they don't have to be computed on every frame.

**Parameters:**

**TimeValue t**

The current time of the call.

**Interval& valid**

The validity interval of the cache created by the plug-in. The plug-in may set this for its own use. The plug-in can then check if the cache is up to date and update it if not.

**Prototype:**

**virtual SFXParamDlg \*CreateParamDialog(IRendParams \*ip);**

**Remarks:**

This method is available in release 4.0 and later only.

This method creates and returns a new instance of a class derived from **SFXParamDlg** to manage the user interface. This put up a modal dialog that lets the user edit the plug-ins parameters.

**Parameters:**

**IRendParams \*ip**

This is the interface given to the plug-in so it may display its parameters.

**Default Implementation:**

**{ return NULL; }**

**Prototype:**

```
virtual BOOL SetDlgThing(SFXParamDlg* dlg);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implement this if you are using the ParamMap2 AUTO\_UI system and the effect has secondary dialogs that don't have the effect as their 'thing'. Called once for each secondary dialog for you to install the correct thing.

Note: Developers needing more information on this method can see the remarks for [MtlBase::CreateParamDlg\(\)](#) which describes a similar example of this method in use (in that case it's for use by a texture map plug-in).

**Parameters:**

```
SFXParamDlg* dlg
```

Points to the ParamDlg.

**Return Value:**

Return TRUE if you process the dialog; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual int NumGizmos();
```

**Remarks:**

Implemented by the Plug-In.

If an atmospheric or rendering effect has references to gizmos or other objects in the scene it can optionally provide access to the object list. This method returns the number of gizmos or objects the plug-in has.

**Default Implementation:**

```
{return 0;}
```

**Prototype:**

```
virtual INode *GetGizmo(int i);
```

**Remarks:**

Implemented by the Plug-In.

Returns a pointer to the 'i-th' gizmo or object node.

**Parameters:**

**int i**

The index of the gizmo to return.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual void DeleteGizmo(int i);
```

**Remarks:**

Implemented by the Plug-In.

Deletes the 'i-th' gizmo or object from those used by the plug-in.

**Parameters:**

**int i**

The index of the gizmo to delete.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void AppendGizmo(INode *node);
```

**Remarks:**

Implemented by the Plug-In.

Adds the specified node to the end of the list of gizmos used by the plug-in.

**Parameters:**

**INode \*node**

Points to the node to append.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL OKGizmo(INode *node);
```

**Remarks:**

Implemented by the Plug-In.

This method is called to approve a node for possible use as gizmo. Return TRUE if the node is okay; otherwise FALSE.

**Parameters:**

**INode \*node**

The node to check.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void EditGizmo(INode *node);
```

**Remarks:**

Implemented by the Plug-In.

This method is called to select the specified gizmo and displays parameters for it (if any).

In the Special Effects section of the light dialog there is a button labelled 'Setup'. The Setup button brings up the Environment dialog (for Atmospherics) or the Render Effects dialog (for Render Effects) and selects the chosen effect. It also selects the "gizmo" within that effect, so if there are particular parameters for each gizmo the user will see them. Pressing that button causes this method to be called.

**Parameters:**

**INode \*node**

The gizmo node.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void InsertGizmo(int i, INode *node);
```

**Remarks:**

Implemented by the Plug-In.

Inserts the specified gizmo node into the list of gizmos. This method must be defined to use the **DeleteGizmoRestore** class.

**Parameters:**

**int i**

The zero based index of the position in the list of where the insertion should take place.

**INode \*node**

The gizmo node to insert.

**Default Implementation:**

```
{ assert(0); }
```

**Prototype:**

```
IOResult Save(ISave *isave);
```

**Remarks:**

Implemented by the System.

To facilitate naming atmospheric or rendering effects, a 'name' string has been added to the base class. This method should be called from the developers sub-classed Atmospheric or Effects plug-in to save the name.

**Prototype:**

```
IOResult Load(ILoad *iload);
```

**Remarks:**

Implemented by the System.

To facilitate naming atmospheric or rendering effects, a 'name' string has been added to the base class. This method should be called from the developers sub-classed Atmospheric or Effects plug-in to load the name.

## Class AppendGizmoRestore

See Also: [Class RestoreObj](#), [Class Atmospheric](#), [Class Effect](#), [Class SpecialFX](#), [Class INode](#).

```
class AppendGizmoRestore: public RestoreObj
```

### Description:

This class is available in release 3.0 and later only.

This class enables implementing undo of Gizmo assignment in Atmosphere and Effects classes. This class provides implementations of the **RestoreObj** methods. An instance of this class can be put on the Hold stack when a Gizmo is appended. For example:

```
if (theHold.Holding())
```

```
theHold.Put(new AppendGizmoRestore(this,node));
```

All methods of this class are implemented by the System.

### Data Members:

public:

```
SpecialFX *fx;
```

Points to the Atmosphere of Effect.

```
INode *node;
```

Points to the gizmo node.

### Methods:

public:

### Prototype:

```
AppendGizmoRestore(SpecialFX *f, INode *n);
```

### Remarks:

Constructor. The data members are initialized to the values passed.

## Class DeleteGizmoRestore

See Also: [Class RestoreObj](#), [Class Atmospheric](#), [Class Effect](#), [Class SpecialFX](#), [Class INode](#).

```
class DeleteGizmoRestore: public RestoreObj
```

### Description:

This class is available in release 3.0 and later only.

This class enables implementing undo of Gizmo deletion in Atmosphere and Effects classes. This class provides implementations of the **RestoreObj** methods. An instance of this class can be put on the Hold when a Gizmo is deleted. For example:

```
if (theHold.Holding())
```

```
theHold.Put(new DeleteGizmoRestore(this,nodes[i],i));
```

All methods of this class are implemented by the System.

### Data Members:

public:

```
SpecialFX *fx;
```

Points to the Atmosphere of Effect.

```
INode *node;
```

Points to the gizmo node.

```
int num;
```

The index of the gizmo which is being deleted.

### Methods:

public:

### Prototype:

```
DeleteGizmoRestore(SpecialFX *a, INode *n, int i);
```

### Remarks:

Constructor. The data members are initialized to the parameter passed.

## Class RenderGlobalContext

See Also: [Class ShadeContext](#), [Class Renderer](#), [Class RenderInstance](#), [Class Texmap](#), [Class Atmospheric](#), [Class Matrix3](#), [Class Point2](#), [Class Point3](#), [Class AColor](#), [Class Color](#), [Class IRenderElementMgr](#), [Class IRenderElement](#).

```
class RenderGlobalContext : public BaseInterfaceServer
```

### Description:

A pointer to an instance of this class is a data member of the ShadeContext (**RenderGlobalContext \*globContext;**). This can be used by materials, texmaps, etc. to retrieve information about the global rendering environment. This is information such as the renderer in use, the project type for rendering, the output device width and height, several matrices for transforming between camera and world coordinates, the environment map, the atmospheric effects, the current time, field rendering information, and motion blur information.

Note that raytracing (and all shading calculations in the default renderer) take place in camera space.

When a ray intersects on the face edge it can happen that no intersection is returned. One way to handle this situation is to perturb the ray minimally so it will point in a slightly different direction. This presumes that you are fairly sure that the no intersection is probably not what you're looking for. **IntersectRay()** is linear in the number of faces so **NUM\_ATTEMPTS** should be kept small.

```
for (perturb=0; perturb < NUM_ATTEMPTS; perturb++)
{
    Matrix3 ptb;
    float prop = ((float)rand()) / ((float) RAND_MAX);
    // gets random rotation of up to half a degree.
    float ang = PI*prop/360.0f;
    switch (perturb%3) {
        case 0: ptb = RotateXMatrix (ang); break;
        case 1: ptb = RotateYMatrix (ang); break;
        case 2: ptb = RotateZMatrix (ang); break;
    }
    ray.dir = ptb*ray.dir;
```

```
// try IntersectRay() again, see if you get a hit.  
}
```

### Data Members:

public:

**Renderer \*renderer;**

A pointer to the active renderer.

**int projType;**

Returns the type of projection used during rendering. One of the following values:

**PROJ\_PERSPECTIVE**

**PROJ\_PARALLEL**

**int devWidth;**

The width in pixels of the output device.

**int devHeight;**

The height in pixels of the output device.

**float xscale;**

The X scale factor for mapping from world space to screen space.

**float yscale;**

The Y scale factor for mapping from world space to screen space.

**float xc;**

The X center point used in mapping from world space to screen space.

**float yc;**

The Y center point used in mapping from world space to screen space.

**BOOL antialias;**

TRUE if antialiasing is enabled; otherwise FALSE.

**Matrix3 camToWorld;**

This matrix may be used to transform coordinates from camera space to world space.

**Matrix3 worldToCam;**

This matrix may be used to transform coordinates from world space to camera space.

**float nearRange;**

The near range setting of the camera.

**float farRange;**

The far range setting of the camera.

**float devAspect;**

The **pixel** aspect ratio of a device pixel. This is the height / width.

**float frameDur;**

This defines the duration of one frame in floating point units. This is used, for example, by video post rendering where the user can stretch time. A video post frame might be 1/2 a frame long for instance.

**Texmap \*envMap;**

The environment map (which may be NULL).

**Color globalLightLevel;**

This parameter is available in release 3.0 and later only.

This is the global light level.

**Atmospheric \*atmos;**

The atmosphere effects (which may be NULL).

**ToneOperator\* pToneOp;**

This data member is available in release 4.0 and later only.

The tone operator, may be NULL

**TimeValue time;**

The current time.

**BOOL wireMode;**

This parameter is available in release 3.0 and later only.

TRUE if rendering in wire frame mode; otherwise FALSE.

**float wire\_thick;**

This parameter is available in release 3.0 and later only.

The global wire thickness.

**BOOL force2Side;**

TRUE if force two-sided rendering enabled; otherwise FALSE.

**BOOL inMtlEdit;**

TRUE if the rendering is being done in the materials editor; otherwise FALSE.

**BOOL fieldRender;**

TRUE if field rendering is being done; otherwise FALSE.

**BOOL first\_field;**

TRUE if this is the first field; FALSE if it's the second.

**BOOL field\_order;**

Determines which field is first. 0 if the even first; 1 if odd first.

**BOOL objMotBlur;**

This is used for handling object motion blur in ray-trace maps and materials. TRUE if object motion blur is enabled; FALSE if it's disabled.

**int nBlurFrames;**

This is used for handling object motion blur in ray-trace maps and materials. The number of object motion blur time slices. See [Class RenderInstance](#).

**Methods:**

**Prototype:**

**IRenderElementMgr \*GetRenderElementMgr();**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the Render Element Manager. See [Class IRenderElementMgr](#).

**Prototype:**

**void SetRenderElementMgr(IRenderElementMgr \*pIRenderElementMgr);**

**Remarks:**

This method is available in release 4.0 and later only.

Sets the render element manager being used.

**Parameters:**

**IRenderElementMgr \*pIRenderElementMgr**

Points to the render element manager to set.

**Prototype:**

**int NRenderElements();**

**Remarks:**

This method is available in release 4.0 and later only.

Returns the number of render elements.

**Prototype:**

**IRenderElement\* GetRenderElement(int n);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the specified render element (or NULL if not found).

**Parameters:**

**int n**

The zero based index of the render element.

**Prototype:**

**Point2 MapToScreen(Point3 p);**

**Remarks:**

Computes the screen space coordinates of the point passed in world coordinates. This is implemented as:

**Parameters:**

**Point3 p**

The point to map to screen space.

**Prototype:**

**virtual FilterKernel\* GetAAFilterKernel();**

**Remarks:**

Returns a pointer to the current anti-aliasing filter from the renderer. See [Class FilterKernel](#).

**Prototype:**

**virtual float GetAAFilterSize();**

**Remarks:**

Returns the filter size of the current anti-aliasing filter.

**Prototype:**

**virtual int NumRenderInstances();**

**Remarks:**

Returns the number of **RenderInstances**.

**Prototype:**

```
virtual RenderInstance* GetRenderInstance(int i);
```

**Remarks:**

Returns a pointer to the 'i-th' **RenderInstance**.

**Parameters:**

**int i**

Specifies which **RenderInstance** to return (**0** through **NumRenderInstances()-1**).

**Prototype:**

```
virtual AColor EvalGlobalEnvironMap(ShadeContext &sc, Ray  
&r, BOOL applyAtmos);
```

**Remarks:**

This method evaluates the global environment map using the specified ray as a point of view, and returns the resulting color.

**Parameters:**

**ShadeContext &sc**

The shade context.

**Ray &r**

Defines the direction of view of the environment. See [Class Ray](#).

**BOOL applyAtmos**

TRUE if atmospheric effects should be considered; otherwise FALSE.

**Default Implementation:**

```
{ return AColor(0.0f,0.0f,0.0f,1.0f); }
```

**Prototype:**

```
virtual void IntersectRay(RenderInstance *inst, Ray& ray, ISect  
&isct, ISectList &xpList, BOOL findExit);
```

**Remarks:**

This method takes the specified **ray** and intersects it with the single RenderInstance **inst**.

**Parameters:****RenderInstance \*inst**

The render instance to intersect. The Mesh may be retrieved via **Mesh& m = \*(inst->mesh);**

**Ray& ray**

Defines the direction to check. This is the point to look from, and a normal vector specifying the direction to look. See [Class Ray](#).

**ISect &isct**

The information about the first opaque object hit by the ray is returned here. See [Structure ISect](#).

**ISectList &xplist**

The list of transparent objects that are intersected on the way to the opaque one are returned here. See [Class ISectList](#).

**BOOL findExit**

TRUE to compute the exit point; FALSE to not compute it. Once a ray has been intersected with a transparent object and you want to find out where the refracted ray leaves the object, this parameter may be set to TRUE. This allows the ray to look at the inside faces of the object and compute the intersection point at exit.

**Prototype:**

```
virtual BOOL IntersectWorld(Ray &ray, int skipID, ISect &hit,  
ISectList &xplist, int blurFrame = NO_MOTBLUR);
```

**Remarks:**

This method takes the specified **ray** and intersects it with the entire 3ds max scene.

**Parameters:****Ray &ray**

Defines the direction to check. This is the point to look from, and a normal vector specifying the direction to look. See [Class Ray](#).

## **int skipID**

This specifies an ID (from **RenderInstance::nodeID**) that is skipped in the intersection computations. This is used to prevent self intersection.

## **ISect &hit**

The information about the first opaque object hit by the ray is returned here. See [Structure ISect](#).

## **ISectList &xplist**

The list of transparent objects that are intersected on the way to the opaque one are returned here. See [Class ISectList](#).

## **int blurFrame = NO\_MOTBLUR**

**NO\_MOTBLUR** is used for non-motion blurred objects. If this is not equal to **NO\_MOTBLUR**, it should be in the range **0** to **nBlurFrames-1**. In that case, this method will only consider blur objects corresponding to that blur sub-frame.

When object motion blur is turned on, for each object, several objects are generated. Each of these objects is given a number. This corresponds to the value **RenderInstance::objMotBlurFrame**.

This method will always intersect objects that aren't motion blurred. However, if this is set to a number other than **NO\_MOTBLUR**, then when it comes to a motion blurred object, it will only look at the sub-object corresponding to the specified slice in time.

This may be used to do a kind of dither where for each of the sub-samples, this number is randomly selected. In this way the different motion blur slices will basically blur together and give a motion blurred ray trace result.

## **Prototype:**

```
virtual ViewParams *GetViewParams();
```

## **Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to a class which describes the properties of a view being rendered.. See [Class ViewParams](#).

## **Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);**

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:**

**int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

**Default Implementation:**

**{ return 0; }**

**Prototype:**

**virtual FILE\* DebugFile();**

**Remarks:**

This method is used internally.

**Default Implementation:**

**{ return NULL; }**

## Class ShadeContext

See Also: [Working with Materials](#), [Class LightDesc](#), [Class RenderGlobalContext](#), [Class Color](#), [Class ShadeOutput](#), [Class AColor](#), [Class UVGen](#), [Class Box3](#), [Class IPoint2](#), [Class Point2](#), [Class IRenderElement](#), [Class Point3](#), [Class Class\\_ID](#).

```
class ShadeContext : public InterfaceServer
```

### Description:

This class is passed to materials and texture maps. It contains all the information necessary for shading the surface at a pixel.

Normally, the **ShadeContext** is provided by the 3ds max renderer. However developers that need to create their own **ShadeContext** for use in passing to the texture and material evaluation functions can do so by deriving a class from **ShadeContext** and providing implementations of the virtual methods. Some sample code is available demonstrating how this is done in `\MAXSDK\SAMPLES\OBJECTS\LIGHT.CPP` (see the code for **class SCLight : public ShadeContext**). The default implementations of these methods are shown for developers that need to create their own **ShadeContext**. Note that raytracing (and all shading calculations in the default renderer) takes place in camera space.

For additional information on the methods **DP()**, **Curve()**, **DUVW()** and **DPdUVW()** see [Additional Notes](#).

All methods are implemented by the system unless noted otherwise.

### Data Members:

public:

#### **ULONG mode;**

One of the following values:

#### **SCMODE\_NORMAL**

In normal mode, the material should do the entire illumination including transparency, refraction, etc.

#### **SCMODE\_SHADOW**

In shadow mode, you are just trying to find out what color the shadow is that is falling on an object. In this case, all you care about is transmitted

color.

### **BOOL doMaps**

Indicates if texture maps should be applied.

### **BOOL filterMaps;**

Indicates if textures should be filtered.

### **BOOL shadow**

Indicates if shadows should be applied.

### **BOOL backFace;**

Indicates if we are on the back side of a 2-sided face.

### **int mtlNum**

The material number of the face being shaded. This is the sub-material number for multi-materials.

### **Color ambientLight**

This is the color of the ambient light.

### **int nLights;**

This is the number of lights being used in a render, which is the number of active lights in the scene, or if there are none, 2 for the default lights. For example, this is used in the Standard material in a loop like this:

```
LightDesc *l;  
for (int i=0; i<sc.nLights; i++) {  
l = sc.Light(i);  
..etc  
}
```

### **int rayLevel;**

This data member is available in release 2.0 and later only.

This is used to limit the number of reflections for raytracing. For instance, if you're rendering a hall of mirrors, and the ray is reflecting back and forth, you don't want the raytracing to go forever. Every time **Texmap::EvalColor()** gets called again on a ray you create a new **ShadeContext** and bump up the **rayLevel** one. This allows you to test this value and see if it has reached the limit of how deep to go (if it reaches a maximum level, you can return black for example).

Note that it is conceivable that more than one raytrace material can be in effect

at a time (from different developers). In such a case, where one surface might have one raytracer and another surface a different one, and a ray was bouncing back and forth between them, each needs to be aware of the other. This is why this value is here -- the two texmaps each modify and check it.

### **int xshadeID;**

This data member is available in release 2.0 and later only.

This is currently not used.

### **RenderGlobalContext \*globContext;**

This data member is available in release 2.0 and later only.

Points to an instance of **RenderGlobalContext**. This class describes the properties of the global rendering environment. This provides information such as the renderer in use, the project type for rendering, the output device width and height, several matrices for transforming between camera and world coordinates, the environment map, the atmospheric effects, the current time, field rendering information, and motion blur information.

### **LightDesc \*atmosSkipLight;**

The light description of lights to prevent self shadowing by volumetric lights.

### **RenderGlobalContext \*globContext;**

A pointer to the rendering global context.

### **ShadeOutput out;**

This is where the material should leave its results.

The following is a discussion of blending the ShadeContext.out.c and ShadeContext.out.t together to get the final color:

The (c,t) returned by shaders is interpreted as follows: t.r is the (premultiplied) alpha for the r-channel, etc.

So if you want to composite (c,t) over a background b,

color = b\*t + c ( where the multiplication of b and t multiplies the individual components ).

When you want to convert a (c,t) to a simple R,G,B,Alpha, just average together the components of t to get Alpha. (and use the r,g,b components of c directly).

### **Methods:**

### **Prototype:**

## **ShadeContext()**

### **Remarks:**

Constructor. The data members are initialized as follows:

```
mode = SCMODE_NORMAL; nLights = 0; shadow = TRUE;  
rayLevel = 0; globContext = NULL; atmosSkipLight = NULL;
```

### **Prototype:**

```
void ResetOutput(int n = -1)
```

### **Remarks:**

Sets the surface color output and surface transparency output to Black.

### **Parameters:**

```
int n = -1
```

By supplying a negative value this method will clear elements but leave the number of elements unchanged.

### **Prototype:**

```
Class_ID ClassID();
```

### **Remarks:**

This method is available in release 2.0 and later only.

Returns the Class\_ID of this ShadeContext. This is used to distinguish different ShadeContexts.

### **Default Implementation:**

```
{ return Class_ID(0,0); }
```

### **Prototype:**

```
virtual BOOL InMtlEditor()=0;
```

### **Remarks:**

This method is available in release 2.0 and later only.

Returns TRUE if this rendering is for the material editor sample sphere (geometry); otherwise FALSE.

### **Prototype:**

**virtual int Antialias();**

**Remarks:**

Returns the state of the antialiasing switch in the renderer dialog - TRUE if on; FALSE if off.

**Default Implementation:**

**{return 0;}**

**Prototype:**

**virtual int ProjType();**

**Remarks:**

This method returns the projection type.

**Return Value:**

A value of 0 indicates perspective projection; a value of 1 indicates parallel projection.

**Default Implementation:**

**{return 0;}**

**Prototype:**

**virtual LightDesc\* Light(int n)=0;**

**Remarks:**

This method returns the 'i-th' light. Use data member **nLights** to get the total number of lights.

**Parameters:**

**int n**

Specifies the light to return.

**Prototype:**

**virtual TimeValue CurTime()=0;**

**Remarks:**

Returns the current time value (the position of the frame slider).

**Return Value:**

The current time.

**Prototype:**

```
virtual int NodeID();
```

**Remarks:**

Returns the node ID for the item being rendered or -1 if not set. This ID is assigned when the scene is being rendered - each node is simply given an ID - 0, 1, 2, 3, etc.

**Default Implementation:**

```
{return -1;}
```

**Prototype:**

```
virtual INode *Node();
```

**Remarks:**

Returns the INode pointer of the node being rendered. This pointer allows a developer to access the properties of the node. See [Class INode](#).

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual Object *GetEvalObject();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the evaluated object for this node. When rendering, usually one calls **GetRenderMesh()** to get the mesh to render. However, at certain times you might want to get the object itself from the node. For example, you could then call **ClassID()** on the object and determine its type. Then the object could be operated on procedurally (for instance you could recognize it as a true sphere, cylinder or torus). Note that this method will return NULL if object is motion blurred.

For example, here is how you can check if the object is a particle system:

```
// ...
```

```
Object *ob = sc.GetEvalObject();  
if (ob && ob->IsParticleSystem()) {
```

```
// ...
```

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual Point3 BarycentricCoords()
```

**Remarks:**

The coordinates relative to triangular face. The barycentric coordinates of a point **p** relative to a triangle describe that point as a weighted sum of the vertices of the triangle. If the barycentric coordinates are **b0**, **b1**, and **b2**, then:

$$\mathbf{p} = \mathbf{b0} * \mathbf{p0} + \mathbf{b1} * \mathbf{p1} + \mathbf{b2} * \mathbf{p2};$$

where **p0**, **p1**, and **p2** are the vertices of the triangle. The Point3 returned by this method has the barycentric coordinates stored in the its three coordinates. These coordinates are relative to the current triangular face being rendered. These barycentric coordinates can be used to interpolate **any** quantity whose value is known at the vertices of the triangle. For example, if a radiosity shader had available the illumination values at each of the three vertices, it could determine the illumination at the current point using the barycentric coordinates.

**Default Implementation:**

```
{ return Point3(0,0,0);}
```

**Prototype:**

```
virtual int FaceNumber()=0;
```

**Remarks:**

Returns the index of the face being rendered. For the scan-line renderer, which renders only triangle meshes, this is the index of the face in the Mesh data structure. This is meant for use in plug-in utilities such as a radiosity renderer, which stores a table of data, indexed on face number, in the Nodes's AppData, for use in a companion material.

**Prototype:**

**virtual Point3 Normal()=0;**

**Remarks:**

Returns the interpolated normal (in camera space). This is the value of the face normal facing towards the camera. This is affected by **SetNormal()** below.

**Prototype:**

**virtual void SetNormal(Point3 p)**

**Remarks:**

This method will set the value of the face normal facing towards the camera. This may be used to temporarily perturb the normal. The Standard material uses this for example because it implements bump mapping. It changes the normal and then calls other lighting functions, etc. These other method then see this changed normal value. When it is done it puts back the previous value.

**Parameters:**

**Point3 p**

The normal to set.

**Prototype:**

**virtual Point3 OrigNormal();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the original surface normal (not affected by **SetNormal()** above.)

**Default Implementation:**

```
{ return Normal(); }
```

**Prototype:**

**virtual float Curve();**

**Remarks:**

This is an estimate of how fast the normal is varying. For example if you are doing environment mapping this value may be used to determine how big an area of the environment to sample. If the normal is changing very fast a large area must be sampled otherwise you'll get aliasing. This is an estimate of

$dN/dsx$ ,  $dN/dsy$  put into a single value.

**Prototype:**

**virtual Point3 Gnormal()=0**

**Remarks:**

This returns the geometric normal. For triangular mesh objects this means the face normal. Normals are unit vectors.

**Prototype:**

**virtual Point3 ReflectVector()=0;**

**Remarks:**

This takes the current view vector and the current normal vector and calculates a vector that would result from reflecting the view vector in the surface. This returns the reflection vector.

**Prototype:**

**virtual Point3 RefractVector(float ior)=0;**

**Remarks:**

This is similar to the method above however it calculates the view vector being refracted in the surface. This returns the refraction vector.

**Parameters:**

**float ior**

The relative index of refraction between the air and the material.

**Prototype:**

**virtual void SetIOR(float ior);**

**Remarks:**

This method is available in release 2.0 and later only.

Set index of refraction.

**Parameters:**

**float ior**

The index of refraction to set. This value can be any positive (non-zero) value.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual float GetIOR();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the index of refraction.

**Default Implementation:**

```
{ return 1.0f; }
```

**Prototype:**

```
virtual Point3 CamPos()=0;
```

**Remarks:**

Returns the camera position in camera space. For the 3ds max renderer this will always be 0,0,0.

**Prototype:**

```
virtual Point3 V()=0
```

**Remarks:**

This method returns the unit view vector, from the camera towards P, in camera space.

**Prototype:**

```
virtual void SetView(Point3 p)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Sets the view vector as returned by **V()**.

**Parameters:**

**Point3 p**

The view vector set.

**Prototype:**

**virtual Point3 OrigView();**

**Remarks:**

This method is available in release 2.0 and later only.

This is the original view vector that was not affected by **ShadeContext::SetView()**.

**Default Implementation:**

**{ return V(); }**

**Prototype:**

**virtual Point3 P()=0**

**Remarks:**

Returns the point to be shaded in camera space.

**Prototype:**

**virtual Point3 DP()=0**

**Remarks:**

This returns the derivative of P, relative to the pixel. This gives the renderer or shader information about how fast the position is changing relative to the screen.

**Prototype:**

**virtual void DP(Point3& dpdx, Point3& dpdy);**

**Remarks:**

This returns the derivative of P, relative to the pixel - same as above. This method just breaks it down into x and y.

**Prototype:**

**virtual Point3 PObj()=0**

**Remarks:**

Returns the point to be shaded in object coordinates.

**Prototype:**

**virtual Point3 DPObj()=0**

**Remarks:**

Returns the derivative of **PObj()**, relative to the pixel.

**Prototype:**

**virtual Box3 ObjectBox()=0**

**Remarks:**

Returns the object extents bounding box in object coordinates.

**Prototype:**

**virtual Point3 PObjRelBox()=0**

**Remarks:**

Returns the point to be shaded relative to the object box where each component is in the range of -1 to +1.

**Prototype:**

**virtual Point3 DPObjRelBox()=0**

**Remarks:**

Returns the derivative of **PObjRelBox()**. This is the derivative of the point relative to the object box where each component is in the range of -1 to +1.

**Prototype:**

**virtual void ScreenUV(Point2& uv, Point2 &duv)=0;**

**Remarks:**

Retrieves the point relative to the screen where the lower left corner is 0,0 and the upper right corner is 1,1.

**Parameters:**

**Point2& uv**

The point.

**Point2 &duv**

The derivative of the point.

**Prototype:**

```
virtual IPoint2 ScreenCoord()=0;
```

**Remarks:**

Returns the integer screen coordinate (from the upper left).

**Prototype:**

```
virtual Point2 SurfacePtScreen();
```

**Remarks:**

This method is available in release 3.0 and later only.

Return the surface point at the center of the fragment in floating point screen coordinates. See the documentation for **Sampler::DoSample()** for an explanation of the use of this method. See [Class Sampler](#).

**Default Implementation:**

```
{ return Point2(0.0,0.0); }
```

**Prototype:**

```
virtual Point3 UVW(int channel=0)=0;
```

**Remarks:**

Returns the UVW coordinates for the point.

**Parameters:**

```
int channel=0;
```

Specifies the channel for the values. One of the following:

**0**: Vertex Color Channel.

**1** through **99**: Mapping Channels.

**Prototype:**

```
virtual Point3 DUVW(int channel=0)=0
```

**Remarks:**

This method returns the UVW derivatives for the point. This is used for filtering texture maps and antialiasing procedurals that are using UVW. Note that in standard 3ds max textures, the **UVGen** class is used, and it calls this method itself. See the methods **UVGen::GetBumpDP()** for more details

for using **UVGen**. If you are not using **UVGen** then you can use this method and **UVW()**. **UVW()** gets the UVW coordinates of the point and **DUVW()** gets the change in the UVWs for the point. This tells you a maximum change for each of UVW. This tells you how much of the area of the map to sample. So when you call the **Bitmap** method **GetFiltered(float u, float v, float du, float dv, BMM\_Color\_64 \*ptr)** this tells you how big the sample should be. This lets you filter or average over this area to keep the map from aliasing.

**Parameters:**

**int channel=0;**

Specifies the channel for the values. One of the following:

**0**: Vertex Color Channel.

**1** through **99**: Mapping Channels.

**Prototype:**

**virtual void DPdUVW(Point3 dP[3], int channel=0)=0**

**Remarks:**

This returns the bump basis vectors for UVW in camera space. Note that if you want to retrieve these bump basis vectors that are altered by the UVGen instance use the method **UVGen::GetBumpDP()**. Also see the Advanced Topics section [Working with Materials and Textures](#) for more details on bump mapping.

**Parameters:**

**Point3 dP[3]**

The bump basic vectors. **dP[0]** is a vector corresponding to the U direction. **dp[1]** corresponds to V, and **dP[2]** corresponds to W.

**int channel=0;**

Specifies the channel for the values. One of the following:

**0**: Vertex Color Channel.

**1** through **99**: Mapping Channels.

**Prototype:**

**virtual int BumpBasisVectors(Point3 dP[2], int axis, int channel=0);**

**Remarks:**

This method is available in release 4.0 and later only.

This method should replace DpDUVW over time but is left in place as not to break 3<sup>rd</sup> party plugins. If this method returns 1, that is assumed to mean it is implemented, and it will be used instead of DpDUVW.

**Parameters:**

**Point3 dP[2]**

The bump basic vectors. **dP[0]** is a vector corresponding to the U direction. **dp[1]** corresponds to V, and **dP[2]** corresponds to W.

**int axis**

Specified the 2D cases for: **AXIS\_UV**, **AXIS\_VW**, or **AXIS\_WU**.

**int channel=0;**

Specifies the channel for the values. One of the following:

**0**: Vertex Color Channel.

**1** through **99**: Mapping Channels.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

**virtual Point3 UVWNormal(int channel=0);**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns a vector in UVW space normal to the face in UVW space. This can be  $\text{CrossProd}(U[1]-U[0], U[2]-U[1])$ , where  $U[i]$  is the texture coordinate at the  $i$ -th vertex of the current face. This may be used for hiding textures on back side of objects.

**Parameters:**

**int channel=0;**

Specifies the channel for the values. One of the following:

**0**: Vertex Color Channel.

1 through 99: Mapping Channels.

**Default Implementation:**

```
{ return Point3(0,0,1); }
```

**Prototype:**

```
virtual float RayConeAngle();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the angle of a ray cone hitting this point. It gets increased/decreased by curvature on reflection.

Visualize a small pyramid, with the top at the eye point, and its sides running through each corner of the pixel to be rendered, then onto the scene. Then visualize a small cone fitting inside this pyramid. This method returns the angle of that cone. When rendering, if the ray cone goes out and hits a flat surface, the angle of reflection will always be constant for each pixel.

However, if the ray cone hits a curved surface, the angle will change between pixels. This change in value give some indication of how fast the sample size is getting bigger.

**Default Implementation:**

```
{ return 0.0f; }
```

**Prototype:**

```
virtual float RayDiam();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the diameter of the ray cone at the pixel point (the point it intersects the surface being shaded). This is a dimension in world units. As a ray is propogated it is updated for each new surface that is encountered.

**Default Implementation:**

```
{ return Length(DP()); }
```

**Prototype:**

```
virtual AColor EvalEnvironMap(Texture *map, Point3 view)
```

**Remarks:**

This is used by the Standard material to do the reflection maps and the refraction maps. Given the map, and a direction from which you want to view it, this method changes the view vector to be the specified vector and evaluates the function.

**Parameters:**

**Texmap \*map**

The map to evaluate.

**Point3 view**

The view direction.

**Return Value:**

The color of the map, in r, g, b, alpha.

**Prototype:**

**virtual void GetBGColor(class Color &bgCol, class Color &transp, int fogBG=TRUE)=0**

**Remarks:**

Retrieves the background color and the background transparency.

**Parameters:**

**class Color &bgCol**

The returned background color.

**class Color &transp**

The returned transparency.

**int fogBG**

Specifies you want the current atmospheric shaders to be applied to the background color. If TRUE the shaders are applied; if FALSE they are not.

**Prototype:**

**virtual float CamNearRange()**

**Remarks:**

Returns the camera near range set by the user in the camera's user interface.

**Prototype:**

## **virtual float CamFarRange()**

### **Remarks:**

Returns the camera far range set by the user in the camera's user interface.

### **Prototype:**

**virtual Point3 PointTo(const Point3& p, RefFrame ito)=0;**

### **Remarks:**

Transforms the specified point from internal camera space to the specified space.

### **Parameters:**

**const Point3& p**

The point to transform.

**RefFrame ito**

The space to transform the point to. One of the following values:

**REF\_CAMERA**

**REF\_WORLD**

**REF\_OBJECT**

### **Return Value:**

The transformed point, in the specified space.

### **Prototype:**

**virtual Point3 PointFrom(const Point3& p, RefFrame ifrom)=0;**

### **Remarks:**

Transforms the specified point from the specified coordinate system to internal camera space.

### **Parameters:**

**const Point3& p**

The point to transform.

**RefFrame ifrom**

The space to transform the point from. One of the following values:

**REF\_CAMERA**

**REF\_WORLD**

## **REF\_OBJECT**

### **Return Value:**

The transformed point in camera space.

### **Prototype:**

**virtual Point3 VectorTo(const Point3& p, RefFrame ito)=0;**

### **Remarks:**

Transform the vector from internal camera space to the specified space.

### **Parameters:**

**const Point3& p**

The vector to transform.

**RefFrame ito**

The space to transform the vector to. One of the following values:

**REF\_CAMERA**

**REF\_WORLD**

**REF\_OBJECT**

### **Prototype:**

**virtual Point3 VectorFrom(const Point3& p, RefFrame ifrom)=0;**

### **Remarks:**

Transform the vector from the specified space to internal camera space.

### **Parameters:**

**const Point3& p**

The vector to transform.

**RefFrame ifrom**

The space to transform the vector from. One of the following values:

**REF\_CAMERA**

**REF\_WORLD**

**REF\_OBJECT**

### **Prototype:**

**virtual Point3 VectorToNoScale(const Point3& p, RefFrame ito);**

**Remarks:**

This method is available in release 3.0 and later only.

Transform the vector from internal camera space to the specified space without scaling.

**Parameters:**

**const Point3& p**

The vector to transform.

**RefFrame ito**

The space to transform the vector to. One of the following values:

**REF\_CAMERA**

**REF\_WORLD**

**REF\_OBJECT**

**Prototype:**

**virtual Point3 VectorFromNoScale(const Point3& p, RefFrame ifrom);**

**Remarks:**

This method is available in release 3.0 and later only.

Transform the vector from the specified space to internal camera space without scaling.

Note: This method was added to correct a problem that was occurring in 3D Textures when the bump perturbation vectors were transformed from object space to camera space, so they are oriented correctly as the object rotates. If the object has been scaled, this transformation causes the perturbation vectors to be scale also, which amplifies the bump effect. This method is used to rotate the perturbation vectors so they are correctly oriented in space, without scaling them.

**Parameters:**

**const Point3& p**

The vector to transform.

**RefFrame ifrom**

**RefFrame ifrom**

The space to transform the vector from. One of the following values:

**REF\_CAMERA**  
**REF\_WORLD**  
**REF\_OBJECT**

**Prototype:**

**virtual void SetGBufferID(int gbid);**

**Remarks:**

When a map or material is evaluated (in **Shade()**, **EvalColor()** or **EvalMono()**), if it has a non-zero **gbufID**, it should call this routine to store the **gbid** into the shade context.

Note: Normally a texmap calls this method so the index would be set for all of the area covered by the texture. There is no reason that this has to be done for every pixel however. A texture could just set the ID for particular pixels. This could allow post processing routines (for example a glow) to only process part of a texture and not the entire thing. For example, at the beginning of texmap's **EvalColor()** one typically has code that does:

**if (gbufid) sc.SetGBufferID(gbufid);**

This takes the **gbufid** (which is in **MtlBase**) and (if it is non-zero) stores it into the shade context. The renderer, after evaluating the **Shade()** function for the material at a pixel, looks at the gbufferID left in the shade context, and stores it into the gbuffer at that pixel. So if the texmap adds another condition like

**if (inHotPortion)**

**if (gbufid) sc.SetGBufferID(gbufid);**

It will do it for just the chosen pixels.

**Parameters:**

**int gbid**

The ID to store.

**Prototype:**

**virtual AColor EvalGlobalEnvironMap(Point3 dir);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the color of the global environment map from the given view direction.

**Parameters:**

**Point3 dir**

Specifies the direction of view.

**Default Implementation:**

```
{ return AColor(0,0,0,0); }
```

**Prototype:**

```
LightDesc *GetAtmosSkipLight();
```

**Remarks:**

This method is available in release 3.0 and later only.

This method, along with **SetAtmosSkipLight()** below, are used by the lights to avoid self-shadowing when applying atmospheric shadows. This method returns a pointer to the **LightDesc** instance currently calling the **Atmosphere::Shade()** method when computing atmospheric shadows.

Here's how they are used:

(1) When computing the atmospheric shadows:(somewhere in **::LightDesc::Illuminate()**) do the following:

```
sc.SetAtmosSkipLight(this);
```

```
sc.globContext->atmos->Shade(sc, lightPos, sc.P(), col, trans);
```

```
sc.SetAtmosSkipLight(NULL);
```

(2) In **LightDesc::TraverseVolume()** do the following:

```
if (sc.GetAtmosSkipLight()==this)
```

```
return;
```

**Default Implementation:**

```
{ return atmosSkipLight; }
```

**Prototype:**

```
void SetAtmosSkipLight(LightDesc *It);
```

**Remarks:**

This method is available in release 3.0 and later only.

This method sets the **LightDesc** instance currently calling the **Atmosphere::Shade()** method. See **GetAtmosSkipLight()** above.

**Parameters:**

**LightDesc \*lt**

Points to the LightDesc to set.

**Default Implementation:**

```
{ atmosSkipLight = lt; }
```

**Prototype:**

```
virtual BOOL GetCache(Texture *map, AColor &c);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method is used with texture maps only. If a map is multiply instanced within the same material, say on the diffuse channel and on the shininess channel, it will return the same value each time its evaluated. Its a waste of processor time to reevaluate the map twice. This method allows you to cache the value so it won't need to be computed more than once.

Note that the cache is automatically cleared after each ShadeContext call. This is used within one evaluation of a material hierarchy.

**Parameters:**

**Texture \*map**

Points to the texture storing the cache (usually the plug-ins **this** pointer).

**AColor &c**

The color to store.

**Return Value:**

TRUE if the color was returned; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Sample Code:**

This code from `\MAXSDK\SAMPLES\MATERIALS\NOISE.CPP`

and shows how the cache is retrieved and stored:

```
RGBA Noise::EvalColor(ShadeContext& sc) {  
  Point3 p,dp;  
  if (!sc.doMaps) return black;  
  
  AColor c;  
  // If the cache exists, return the color  
  if (sc.GetCache(this,c))  
    return c;  
  // Otherwise compute the color  
  ...  
  // At the end of the eval the cache is stored  
  sc.PutCache(this,c);
```

**Prototype:**

```
virtual BOOL GetCache(Texmap *map, float &f);
```

**Remarks:**

This method is available in release 2.0 and later only.

Retrieves a floating point value from the cache. See the AColor version above for details.

**Parameters:**

**Texmap \*map**

Points to the texmap storing the cache (usually the plug-ins **this** pointer).

**float &f**

The value to store.

**Return Value:**

TRUE if the value was returned; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual BOOL GetCache(Texmap *map, Point3 &p);
```

**Remarks:**

This method is available in release 2.0 and later only.

Retrieves a Point3 value from the cache. See the AColor version above for details.

**Parameters:**

**Texmap \*map**

Points to the texmap storing the cache (usually the plug-ins **this** pointer).

**Point3 &p**

The point to store.

**Return Value:**

TRUE if the value was returned; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void PutCache(Texmap *map, const AColor &c);
```

**Remarks:**

This method is available in release 2.0 and later only.

Puts a color to the cache. See the method **GetCache(Texmap \*map, const AColor &c)** above for details.

**Parameters:**

**Texmap \*map**

Points to the texmap storing the cache (usually the plug-ins **this** pointer).

**const AColor &c**

The color to store.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void PutCache(Texmap *map, const float f);
```

**Remarks:**

This method is available in release 2.0 and later only.

Puts a floatint point value to the cache. See the method **GetCache(Texmap \*map, const AColor &c)** above for details.

**Parameters:**

**Texmap \*map**

Points to the texmap storing the cache (usually the plug-ins **this** pointer).

**const float f**

The floating point value to store.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void PutCache(Texmap *map, const Point3 &p);
```

**Remarks:**

This method is available in release 2.0 and later only.

Puts a floatint point value to the cache. See the method **GetCache(Texmap \*map, const AColor &c)** above for details.

**Parameters:**

**Texmap \*map**

Points to the texmap storing the cache (usually the plug-ins **this** pointer).

**const Point3 &p**

The Point3 value to store.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void TossCache(Texmap *map);
```

**Remarks:**

This method is available in release 2.0 and later only.

Removes the specified cache.

**Parameters:**

**Texmap \*map**

Points to the texmap storing the cache (usually the plug-ins **this** pointer).

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual FILE* DebugFile();
```

**Remarks:**

This method is used internally.

**Prototype:**

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0);
```

**Remarks:**

This method is available in release 3.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

**Parameters:**

**int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value (defined uniquely for each **cmd**).

**Default Implementation:**

```
{ return 0; }
```

**Prototype:****bool IsPhysicalSpace() const;****Remarks:**

This method is available in release 4.0 and later only.

This method returns TRUE if the operator really maps physical values to RGB, otherwise FALSE. This method is provided so shaders can determine whether the shading calculations are in physical or RGB space.

**Prototype:****void ScaledToRGB(T& energy) const;****Remarks:**

This method is available in release 4.0 and later only.

This method will map a scaled energy value into RGB. This converts a color value which will be stored in energy.

**Parameters:****T& energy**

The converted color value.

**Prototype:****float ScaledToRGB(float energy) const;****Remarks:**

This method is available in release 4.0 and later only.

This method will map a scaled energy value into RGB. This converts a monochrome value which will be returned.

**Parameters:****float energy**

The scaled energy value.

**Prototype:****void ScaledToRGB();****Remarks:**

This method is available in release 4.0 and later only.

This method will map an energy value into **out.c** into RGB. The converted value is stored in **out.c**.

**Prototype:**

**void ScalePhysical(T& energy) const;**

**Remarks:**

This method is available in release 4.0 and later only.

This method will scale physical values so they can be used in the renderer.

This converts a color value which will be stored in energy.

**Parameters:**

**T& energy**

The converted color value.

**Prototype:**

**float ScalePhysical(float energy) const;**

**Remarks:**

This method is available in release 4.0 and later only.

This method will scale physical values so they can be used in the renderer.

This converts a monochrome value which will be returned.

**Parameters:**

**float energy**

The energy value.

**Prototype:**

**void ScaleRGB(T& energy) const;**

**Remarks:**

This method is available in release 4.0 and later only.

This method will scale RGB values, just supplied to invert ScalePhysical. This converts a color value which will be stored in energy.

**Parameters:**

**T& energy**

The converted color value.

**Prototype:**

**float ScaleRGB(float energy) const;**

**Remarks:**

This method is available in release 4.0 and later only.

This method will scale RGB values, just supplied to invert ScalePhysical. This converts a monochrome value which will be returned.

**Parameters:**

**float energy**

The energy value.

**Prototype:**

**LightDesc \*GetAtmosSkipLight();**

**Remarks:**

This method is available in release 4.0 and later only.

This method can be used to determine from within the ShadeContext if a volumetric light should be prevented from generating self-shadows.

**Default Implementation:**

```
{ return atmosSkipLight; }
```

**Prototype:**

**void SetAtmosSkipLight(LightDesc \*lt);**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the volumetric light that should be prevented from generating self-shadows.

**Parameters:**

**LightDesc \*lt**

A pointer to the light to set.

**Default Implementation:**

```
{ atmosSkipLight = lt; }
```

**Prototype:**

```
virtual int NRenderElements();
```

**Remarks:**

This method is available in release 4.0 and later only.  
Returns the number of render elements.

**Default Implementation:**

```
{ return globContext->NRenderElements(); }
```

**Prototype:**

```
virtual IRenderElement *GetRenderElement(int n);
```

**Remarks:**

This method is available in release 4.0 and later only.  
Returns an interface to the 'i-th' render element.

**Parameters:**

**int n**

The zero based index of the render element to return.

**Default Implementation:**

```
{ return globContext->GetRenderElement(n); }
```

**Prototype:**

```
virtual Color DiffuseIllum();
```

**Remarks:**

This method is available in release 4.0 and later only.  
Computes and returns the incoming diffuse illumination color (for matte/shadow).

**Notes on the functions DP(), Curve(), DUVW() and DPdUVW()**

The functions DP(), Curve(), and DUVW() are all for the purposes of antialiasing. They give the amount of variation of various quantities across the pixel. DPdUVW() is for the purposes of bump mapping. This section describes the method that 3ds max's renderer uses for computing them.

(1) Point3 DP();

This gives the approximate dimension of a 3D box which bounds the surface fragment cut out of the surface by the current pixel being rendered. It can be used for antialiasing 3D textures.

First calculate (where  $U = \{u,v,w\}$  )

$dP/dx$ : the derivative of a point on the surface relative to the screen x-coordinate.

$dP/dy$ : the derivative of a point on the surface relative to the screen y-coordinate.

Then take the sum of the absolute values of these:

$DP = \text{abs}(dP/dx) + \text{abs}(dP/dy)$ .

(2) Point3 DUVW();

This is similar to DP(), but bounds the change in UVW space. It is used for texture filtering.

First calculate (where  $U = \{u,v,w\}$  )

$dU/dx$ : the derivative of  $\{u,v,w\}$  relative to the screen x-coordinate.

$dU/dy$ : the derivative of  $\{u,v,w\}$  relative to the screen y-coordinate.

Then take the sum of the absolute values of these:

$DUVW = DU = \text{abs}(dU/dx) + \text{abs}(dU/dy)$ .

(3) float Curve();

This gives an approximation of the amount of curvature of the surface. It is basically a measure of the length of a vector representing the rate of change of the surface normal across the pixel. It is used to scale the sample size in reflection mapping. (high curvature==>big sample).

Here is our implementation:

```

float SContext::Curve() {
    calc_derivs(); // calculate dxdx, etc.
    Point3 nv0 = actf->GetVertexNormal(0); // surface normal at vertex 0
    Point3 nv1 = actf->GetVertexNormal(1); // surface normal at vertex 1
    Point3 nv2 = actf->GetVertexNormal(2); // surface normal at vertex 2
    nv1 -= nv0;
    nv2 -= nv0;
    float dx2 = LengthSquared(dsdX*nv1 + dtdx*nv2);
    float dy2 = LengthSquared(dsdY*nv1 + dtdy*nv2);
    return (float)sqrt(0.5*(dx2+dy2));
}

```

Notes:

calc\_derivs() calculates the terms dsdx,dsdy, dtdx, dtdy.

(s,t) are skew coordinates of the current point in the face relative to the face vertices (V0,V1,V2):

(i.e. the current point  $P = V0 + s*(V1-V0) + t*(V2-V0)$ ;) )

dsdx is derivative of s relative to screen x.

dsdy is derivative of s relative to screen y.

dtdx is derivative of t relative to screen x.

dtdy is derivative of t relative to screen y.

dx2 is the length squared of the change in normal relative to screen x.

dy2 is the length squared of the change in normal relative to screen y.

(4) void DPdUVW(Point3 dP[3]);

This method returns the 3 basis vectors that describe U,V,and W axes in XYZ space.

The following function computes the U and V bump basis vectors for a triangle given the texture coordinates at the three vertices of the triangle ( tv[] ) and the 3D coordinates at the vertices ( v[] ). It is simply a solution using

linear algebra for the U and V axes in terms of the XYZ coordinates. It returns

$$b[0] = DP/DU$$

$$b[1] = DP/DV$$

This function does not compute DP/DW, which at present is a shortcoming of the scanline renderer. It also makes the assumption that the bump basis vectors are constant over a given face, which has worked out successfully.

```
void ComputeBumpVectors(const Point3 tv[3], const Point3 v[3], Point3  
bvec[3]) {
```

```
float uva,uvb,unc,uvd,uvk;
```

```
Point3 v1,v2;
```

```
uva = tv[1].x-tv[0].x;
```

```
uvb = tv[2].x-tv[0].x;
```

```
unc = tv[1].y-tv[0].y;
```

```
uvd = tv[2].y-tv[0].y;
```

```
uvk = uvb*unc - uva*uvd;
```

```
v1 = v[1]-v[0];
```

```
v2 = v[2]-v[0];
```

```
if (uvk!=0) {
```

```
  bvec[0] = (unc*v2-uvd*v1)/uvk;
```

```
  bvec[1] = (uva*v2-uvb*v1)/uvk;
```

```
}
```

```
else {
```

```
  if (uva!=0)
```

```
    bvec[0] = v1/uva;
```

```
  else if (uvb!=0)
```

```
bvec[0] = v2/uvb;  
else  
bvec[0] = Point3(0.0f,0.0f,0.0f);  
if (uvc!=0)  
bvec[1] = v1/uvc;  
else if (uvd!=0)  
bvec[1] = v2/uvd;  
else  
bvec[1] = Point3(0.0f,0.0f,0.0f);  
}  
bvec[2] = Point3(0,0,1);  
}
```

# Class NURBSExtrudeSurface

See Also: [Class NURBSSurface](#).

```
class NURBSExtrudeSurface : public NURBSSurface
```

## Description:

This class is available in release 2.0 and later only.

This class defines a dependent extrude surface. An extrude surface is extruded from a curve sub-object. It is similar to a surface created with the Extrude modifier. The advantage is that an extrude sub-object is part of the NURBS model, so you can use it to construct other curve and surface sub-objects. Methods are available to get/set the parent index and id, get/set the extrusion vector and get/set the extrusion distance.

All methods of this class are implemented by the system.

## Friend Classes:

```
friend class NURBSSet
```

## Methods:

protected:

## Prototype:

```
void Clean(NURBSIdTab ids);
```

## Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

## Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

## Prototype:

```
NURBSExtrudeSurface();
```

## Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNExtrudeSurface;  
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mXForm.IdentityMatrix();  
mDistance = 0.0;  
mCurveStartParam = 0.0;
```

**Prototype:**

```
virtual ~NURBSExtrudeSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the parent object.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent();**

**Remarks:**

Returns the index in the **NURBSSet** of the parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Prototype:**

**void SetAxis(TimeValue t, Matrix3& ray);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the extrusion axis at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the axis system.

**Matrix3& ray**

The extrusion axis.

**Prototype:**

**Matrix3& GetAxis(TimeValue t);**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the extrusion axis at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the axis.

**Prototype:**

**void SetDistance(TimeValue t, double d);**

**Remarks:**

Sets the length of the extrusion at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the distance.

**double d**

The distance to set.

**Prototype:**

**double GetDistance(TimeValue t);**

**Remarks:**

Returns the length of the extrusion at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the distance.

**Prototype:**

**void SetCurveStartPoint(TimeValue t, double startpoint);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the start point at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the start point.

**double startpoint**

The start point in the range 0.0 to 1.0.

**Prototype:**

**double GetCurveStartPoint(TimeValue t);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the start point at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the start point.

**Operators:**

**Prototype:**

**NURBSExtrudeSurface & operator=(const  
NURBSExtrudeSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSExtrudeSurface& surf**

The surface to assign.

## Class NURBSSet

See Also: [Class NURBSObject](#), [Class NURBSDisplay](#), [Class TessApprox](#), [Class NURBSFuseSurfaceCV](#), [Class NURBSFuseCurveCV](#), [Template Class Tab](#).

```
class NURBSSet : public NURBSObject
```

### Description:

This class is available in release 2.0 and later only.

This class contains a table of **NURBSObject** entities used to make up the set. Additionally it has two fuse tables: one for fuse curves and one for fuse surfaces. These are used to allow the CVs in the curves or surfaces to be 'stitched' together so if one curve or surface moves the other moves with it. This class also has information required to tessellate the objects to triangle meshes for use in the viewports and the production renderer.

All methods of this class are implemented by the system.

### Data Members:

protected:

**TessApprox \*mpVTess;**

This object describes the properties of a tessellation approximation to the mathematical surface for representation in the viewports.

**TessApprox \*mpRTess;**

This object describes the properties of a tessellation approximation to the mathematical surface for the production renderer.

**TessApprox \*mpRTessDisp;**

This object describes the tessellation properties for displacement mapping in the production renderer.

**TessApprox \*mpVTessCurve;**

This object describes the properties of a tessellation approximation to the mathematical curve for representation in the viewports.

**TessApprox \*mpRTessCurve;**

This object describes the properties of a tessellation approximation to the mathematical curve for the production renderer.

**float mTessMerge;**

Controls the tessellation of surface sub-objects whose edges are joined or very

nearly joined. When input to a modifier -- such as Mesh Select -- requires a mesh, and when NURBS surfaces are tessellated for production rendering, by default 3ds max adjusts the tessellation of adjoining surfaces to match each other, in terms of the number of faces along the edges. The Merge parameter controls how this is done. If Merge is zero, adjoining faces are unchanged. Increasing the value of Merge increases the distance 3ds max uses to calculate how edges should match, guaranteeing no gaps between the surfaces when they are rendered.

**Tab<NURBSObject\*> mObjects;**

A table of pointers to the **NURBSObjects** in the set.

**Object \*mpObject;**

The instantiated object in the scene associated with this **NURBSSet**. This is NULL if there isn't one.

**NURBSDisplay mDisplay;**

Controls the display of the object in the viewport..

public:

**Tab<NURBSFuseSurfaceCV> mSurfFuse;**

A table of objects used to allow surfaces in the set to relate to on another.

**Tab<NURBSFuseCurveCV> mCurveFuse;**

A table of objects used to allow curves in the set to relate to on another.

**Methods:**

**Prototype:**

**NURBSSet();**

**Remarks:**

Constructor. The data members are initialized as follows:

**mpObject = NULL;**

**mpVTess = NULL;**

**mpRTess = NULL;**

**mpRTessDisp = NULL;**

**mpVTessCurve = NULL;**

**mpRTessCurve = NULL;**

**mTessMerge = 0.01f;**

```
mDisplay.mDisplayCurves = TRUE;  
mDisplay.mDisplaySurfaces = TRUE;  
mDisplay.mDisplayLattices = FALSE;  
mDisplay.mDisplayShadedLattice = FALSE;  
mDisplay.mDisplaySurfCVLattices = TRUE;  
mDisplay.mDisplayCurveCVLattices = TRUE;  
mDisplay.mDisplayDependents = TRUE;  
mDisplay.mDisplayTrimming = TRUE;  
mDisplay.mDegradeOnMove = TRUE;
```

**Prototype:**

```
~NURBSSet();
```

**Remarks:**

Destructor. Any tessellation objects are deleted.

**Prototype:**

```
void Clean();
```

**Remarks:**

This method is available in release 3.0 and later only.

This removes the NURBS Set connection to a live NURBS object. One use of this is so you can call **CreateNURBSObject()** twice with the same **NURBSSet**. You call **Clean()** in between the calls. Another use is if you want to use the API to copy a NURBS object. You could call **GetNURBSSet()** followed by a **Clea()** followed by a **CreateNURBSObject()**.

**Prototype:**

```
int GetNumObjects();
```

**Remarks:**

Returns the number of objects in the set.

**Prototype:**

**void SetObject(int index, NURBSObject\* obj);**

**Remarks:**

This method sets an object in the table of objects maintained by the set. If the **index** is to an existing object in the set this will replace that object. If it is a new index, all the objects which follow this one in the set are set to NULL and the one passed is set.

**Parameters:**

**int index**

If the index is an existing object in the set this will replace the object. If it is a new index, all the objects which follow this one are set to NULL and the one passed is set.

**NURBSObject\* obj**

Points to the object to add to the table.

**Prototype:**

**NURBSObject\* GetNURBSObject(int index);**

**Remarks:**

Returns a pointer to the specified object in the table.

**Parameters:**

**int index**

The zero based index of the object to return.

**Prototype:**

**NURBSObject\* GetNURBSObject(NURBSId id);**

**Remarks:**

Returns a pointer to the specified object in the table.

**Parameters:**

**NURBSId id**

The Id of the object to return.

**Prototype:**

**void SetObject(int index, NURBSObject\* obj);**

**Remarks:**

Sets the specified NURBSObject pointer in the table to the specified pointer. If the value of index is greater than the number of items in the table, the table is resized and any non-initialized pointers are set to NULL.

**Parameters:**

**int index**

The zero based index of the object to set.

**NURBSObject\* obj**

Points to the object to set.

**Prototype:**

**int AppendObject(NURBSObject\* obj);**

**Remarks:**

Adds the specified object pointer to the end of the table of object pointers.

Note: This method is ONLY used for adding an object to a **NURBSSet** that is not yet in the scene. To add an object to an existing scene use the global function **AddNURBSObjects()**.

**Parameters:**

**NURBSObject\* obj**

The pointer to the object to append.

**Return Value:**

Returns the number of objects in the table prior to appending.

**Prototype:**

**void RemoveObject(int index);**

**Remarks:**

Removes the specified object pointer from the table.

**Parameters:**

**int index**

The zero based index of the object to remove.

**Prototype:**

```
void DeleteObjects();
```

**Remarks:**

Deletes all the objects that are in the table.

This method frees all the **NURBSObjects** in a **NURBSSet**. A developer using a **NURBSSet** must call this method to free all the memory when done.

**Prototype:**

```
TessApprox* GetProdTess(NURBSTessType  
type=kNTessSurface);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the TessApprox object for the production renderer for the specified tessellation operation.

**Parameters:**

```
NURBSTessType type=kNTessSurface
```

The type of tessellation. See [List of NURBSTessTypes](#).

**Prototype:**

```
TessApprox* GetViewTess(NURBSTessType  
type=kNTessSurface);
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the TessApprox object for the viewport renderer for the specified tessellation operation.

**Parameters:**

```
NURBSTessType type=kNTessSurface
```

The type of tessellation. See [List of NURBSTessTypes](#).

**Prototype:**

```
void SetProdTess(TessApprox& tess, NURBSTessType  
type=kNTessSurface);
```

**Remarks:**

Sets the TessApprox object for the production renderer for the specified tessellation operation.

**Parameters:****TessApprox& tess**

The tessellation object.

**NURBSTessType type=kNTessSurface**

This parameter is available in release 3.0 and later only.

The type of tessellation. See [List of NURBSTessTypes](#).

**Prototype:**

```
void SetViewTess(TessApprox& tess, NURBSTessType  
type=kNTessSurface);
```

**Remarks:**

Sets the TessApprox object for the viewport renderer for the specified tessellation operation.

**Parameters:****TessApprox& tess**

The tessellation object.

**NURBSTessType type=kNTessSurface**

This parameter is available in release 3.0 and later only.

The type of tessellation. See [List of NURBSTessTypes](#).

**Prototype:**

```
void ClearViewTess(NURBSTessType type=kNTessSurface);
```

**Remarks:**

This method is available in release 3.0 and later only.

Clears (deletes) the specified tessellation object used for viewport rendering.

**Parameters:****NURBSTessType type=kNTessSurface**

The type of tessellation. See [List of NURBSTessTypes](#).

**Prototype:**

**void ClearProdTess(NURBSTessType type=kNTessSurface);**

**Remarks:**

This method is available in release 3.0 and later only.

Clears (deletes) the specified tessellation object used for production rendering.

**Parameters:**

**NURBSTessType type=kNTessSurface**

The type of tessellation. See [List of NURBSTessTypes](#).

**Prototype:**

**void GetTessMerge();**

**Remarks:**

Returns the tessellation merge value.

**Prototype:**

**void SetTessMerge(float merge);**

**Remarks:**

Sets the tessellation merge value.

**Parameters:**

**float merge**

The tessellation merge value. Controls the tessellation of surface sub-objects whose edges are joined or very nearly joined. When input to a modifier -- such as Mesh Select -- requires a mesh, and when NURBS surfaces are tessellated for production rendering, by default 3ds max adjusts the tessellation of adjoining surfaces to match each other, in terms of the number of faces along the edges. The Merge parameter controls how this is done. If Merge is zero, adjoining faces are unchanged. Increasing the value of Merge increases the distance 3ds max uses to calculate how edges should match, guaranteeing no gaps between the surfaces when they are rendered.

**Prototype:**

**Object\* GetMAXObject();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the instantiated object in the scene associated with this **NURBSSet**. This is NULL if there isn't one.

**Prototype:**

**NURBSDisplay GetDisplaySettings();**

**Remarks:**

This method is available in release 2.5 and later only.

Returns the display settings for this **NURBSSet**.

**Prototype:**

**void SetDisplaySettings(NURBSDisplay& disp);**

**Remarks:**

This method is available in release 2.5 and later only.

Sets the display settings used by this **NURBSSet**.

**Parameters:**

**NURBSDisplay& disp**

The settings to use.

# Class TessApprox

See Also: [Class NURBSSet](#).

class TessApprox

## Description:

This class is available in release 2.0 and later only.

This class describes the properties of a tessellation approximation to the mathematical surface.

All methods of this class are implemented by the system.

## Data Members:

public:

### TessType type;

These are the types of tessellation (one of which is obsolete). One of the following values:

#### TESS\_SET

This is the old form of tessellation for Bezier Patches. This is also the default for these patches. For instance, if you create a Quad Patch and apply an Edit Patch modifier, then exit sub-object mode, you'll see a panel in the rollup for 'Tessellation'. The top choice is 'Fixed (original)'. This is the same type of tessellation done in 3ds max 1.x.

#### TESS\_PARAM

Specifies parametric tessellation. This provides for a fixed number of **u** by **v** tessellations. There are **u** times **v** quadrilaterals and each one is split up into two triangles.

#### TESS\_SPATIAL

Specifies spatial tessellation. This uses **edge** as its parameter. This specifies that the size of the tessellation will be the **edge** length (see below). In view dependent tessellation **edge** is specified in pixels.

#### TESS\_CURVE

Specifies view dependent tessellation. This uses the **ang** and **dist** data members described below.

#### TESS\_LDA

This option is available in release 3.0 and later only.

Specifies a method which combines the spatial (edge-length) method and the curvature (distance and angle) methods. This uses the **ang**, **dist** and **edge** data members below.

### **TESS\_REGULAR**

This option is available in release 3.0 and later only.

Generates a fixed, regular tessellation across the surface. There are no additional parameters.

### **TESS\_ISO**

Obsolete -- Do Not Use.

## **ViewConfig vpt\_cfg;**

This determines what is displayed in the interactive renderer. These correspond to the controls in the user interface (under Surface/Approximation/Viewports). This is not available for bezier patches. One of the following values:

### **ISO\_ONLY**

Only Iso lines. Iso(parametric) lines are similar to contour lines. The lines show where the NURBS surface has a constant U value or V value or both. Iso line representations can be less crowded and easier to visualize than wire mesh representations..

### **ISO\_AND\_MESH**

Iso lines and the mesh. When chosen, wireframe viewports display iso line representations of the surface, and shaded viewports display the shaded surface.

### **MESH\_ONLY**

Just the mesh. When chosen, wireframe viewports display the surface as a wire mesh, and shaded viewports display the shaded surface. In wireframe viewports, this option lets you see the curve approximation used for viewports.

## **TessSubdivStyle subdiv;**

This data member is available in release 3.0 and later only.

The type of subdivision. One of the following values:

### **SUBDIV\_TREE**

Subdivides the surface using a binary tree.

### **SUBDIV\_GRID**

Subdivides the surface using a regular grid.

### **SUBDIV\_DELAUNAY**

Subdivides the surface using nearly equilateral triangles.

### **BOOL view;**

Specifies if this is view dependent tessellation. If TRUE this will tessellate less finely the farther away from the camera the object is. If FALSE the tessellation does not change based on distance from the camera.

### **int u;**

This is used for parametric tessellation. This is the number of tessellations in u. This is the number of sub-divisions for a knot span for the surface.

### **int v;**

This is used for parametric tessellation. This is the number of tessellations in v.

### **int u\_iso;**

This is used with the ISO line display. This is the number of additional **interior** iso lines in u (there are always lines along the outer edges).

### **int v\_iso;**

This is used with the ISO line display. This is the number of additional **interior** iso lines in v (there are always lines along the outer edges).

### **float ang;**

This is used in curvature dependent tessellation (**TESS\_CURVE**). If 0.0 is specified this is ignored. If specified this ensure that no two adjacent face normals exceed this angle between them. This value is specified in radians.

### **float dist;**

This is used in curvature dependent tessellation (**TESS\_CURVE**). If 0.0 is specified this is ignored. This specifies a distance that cannot be exceeded between a vertex on the mesh and the mathematical surface. This is defined as a percentage of the diagonal of the bounding box of the individual surface in object space. For instance if this was set to 1.0, the allowable error in generating a tessellation would be 1% of the bounding box diagonal distance of the surface. This would be 1/100 (1 %) of the diagonal distance of the bounding box. In this way if an object is scaled the tessellation remains the same. Additionally, if you have an object with a big surface and a little surface, the smaller surface will get tessellated more finely because its own bounding box is used. This prevents the smaller surface from just becoming a single triangle for example.

**float edge;**

This is the length of an edge to use in spatial (**TESS\_SPATIAL**) tessellation. In view dependent tessellation this is specified in pixels. If not in view dependent tessellation this is a percentage of the bounding box diagonal length.

**int minSub;**

This data member is available in release 3.0 and later only.

For Grid or Tree subdivisions, this limit controls the number of recursive decompositions that are performed during tessellation. This is the minimum number of recursions.

**int maxSub;**

This data member is available in release 3.0 and later only.

For Grid or Tree subdivisions, this limit controls the number of recursive decompositions that are performed during tessellation. This is the maximum number of recursions.

**int maxTris;**

This data member is available in release 3.0 and later only.

For Delaunay subdivision, this specifies the maximum number of triangles into which the surface will be divided.

**Methods:****Prototype:****TessApprox();****Remarks:**

Constructor. The data members are initialized as follows:

**type = TESS\_SET;****u = v = 2;****u\_iso = 2;****v\_iso = 3;****view = FALSE;****ang = 20.0f;****dist = 10.0f;****edge = 10.0f;****vpt\_cfg = ISO\_AND\_MESH;**

```
merge = 0.0f;  
minSub = 0;  
maxSub = 5;  
maxTris = 20000;  
subdiv = SUBDIV_TREE;  
showInteriorFaces = FALSE;
```

**Prototype:**

```
TessApprox(TessType type, float distance, float edge, float angle,  
TessSubdivStyle subdivStyle, int minSub, int maxSub, float m =  
0.0f);
```

**Remarks:**

This method is available in release 3.0 and later only.

Constructor. The data members are initialized to the values passed.

**Prototype:**

```
TessApprox(const TessApprox &tess);
```

**Remarks:**

Constructor. The data members are initialized from the object passed.

**Operators:**

**Prototype:**

```
TessApprox & operator=(const TessApprox& tess);
```

**Remarks:**

Assignment operator.

**Parameters:**

```
const TessApprox &tess
```

The object to assign.

**Prototype:**

```
int operator==(const TessApprox &tess) const;
```

**Remarks:**

Equality operator. Returns nonzero if they are equal; otherwise zero.

**Parameters:**

**const TessApprox &tess**

The object to compare.

## Class NURBSCurve

See Also: [Class NURBSObject](#), [Class NURBSTrimPoint](#).

```
class NURBSCurve : public NURBSObject
```

### Description:

This class is available in release 2.0 and later only.

This class describes the properties of a NURBS curve. This includes its number of trim points and its open/closed state. The **Evaluate()** method is used to compute points on the curve.

All methods of this class are implemented by the system.

### Data Members:

protected:

**int mMatID;**

This data member is available in release 3.0 and later only.

The material ID for the curve.

### Friend Classes:

**friend class NURBSCVCurve;**

**friend class NURBSPointCurve;**

**friend class NURBSBlendCurve;**

**friend class NURBSOffsetCurve;**

**friend class NURBSXFormCurve;**

**friend class NURBSMirrorCurve;**

**friend class NURBSFilletCurve;**

**friend class NURBSChamferCurve;**

**friend class NURBSIsoCurve;**

**friend class NURBSSurfaceEdgeCurve;**

**friend class NURBSProjectVectorCurve;**

**friend class NURBSProjectNormalCurve;**

**friend class NURBSSurfaceNormalCurve;**

**friend class NURBSNBlendSurface;**

**friend class NURBSRuledSurface;**

**friend class NURBSULoftSurface;**

**friend class NURBSUVLoftSurface;**

```
friend class NURBSExtrudeSurface;  
friend class NURBSLatheSurface;  
friend class NURBSCapSurface;  
friend class NURBS1RailSweepSurface;  
friend class NURBS2RailSweepSurface;  
friend class NURBSMultiCurveTrimSurface;
```

**Methods:**

public:

**Prototype:**

```
NURBSCurve();
```

**Remarks:**

Constructor. The data members are initialized as follows:

```
mKind = kNURBSCurve;
```

```
mMatID = 1;
```

**Prototype:**

```
~NURBSCurve();
```

**Remarks:**

Destructor.

**Prototype:**

```
BOOL IsClosed();
```

**Remarks:**

Returns TRUE if the curve is closed; otherwise FALSE.

**Prototype:**

```
int NumTrimPoints();
```

**Remarks:**

Returns the number of trim points in the curve.

**Prototype:**

**NURBSTrimPoint GetTrimPoint(TimeValue t, int i);**

**Remarks:**

Returns the 'i-th' trim point.

**Parameters:**

**TimeValue t**

The time to retrieve the trim point.

**int i**

The zero based index of the trim point to return.

**Prototype:**

**BOOL Evaluate(TimeValue t, double u, Point3& pt, Point3& tangent);**

**Remarks:**

Retrieves the point along the curve at the specified point and the tangent at that point.

**Parameters:**

**TimeValue t**

The time to evaluate the curve.

**double u**

Specifies the point along the curve to evaluate. This value must be between the **uMin** and **uMax** as returned from **GetParameterRange()**.

**Point3& pt**

The point along the curve is returned here.

**Point3& tangent**

The tangent at the specified point is returned here.

**Return Value:**

TRUE if the method was able to evaluate the curve; otherwise FALSE.

**Prototype:**

**void GetParameterRange(TimeValue t, double& uMin, double& uMax);**

**Remarks:**

Retrieves the minimum and maximum valid values for **u** as passed to **Evaluate()**.

**Parameters:**

**TimeValue t**

The time to get the parameter range of the curve.

**double& uMin**

The minimum value is returned here.

**double& uMax**

The maximum value is returned here.

**Prototype:**

**BOOL GetNURBSData(TimeValue t, int& degree, int& numCVs, NURBSCVTab& cvs, int& numKnots, NURBSKnotTab knots);**

**Remarks:**

This method is available in release 2.5 and later only.

Retrieves data about the NURBSCurve at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the NURBS information.

**int& degree**

The degree of the curve.

**int& numCVs**

The number of CVs.

**NURBSCVTab& cvs**

The table of CVs. Note: **typedef Tab<NURBSControlVertex>**

**NURBSCVTab;**

**int& numKnots**

The number of knots.

**NURBSKnotTab knots**

A table of knots in U. Note: **typedef Tab<double> NURBSKnotTab;**

**Return Value:**

TRUE if the data was retrieved; otherwise FALSE.

**Prototype:**

**int MatID();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the material ID for the curve.

**Prototype:**

**void MatID(int id);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the material ID for the curve.

**Parameters:**

**int id**

The ID to set.

**Operators:**

**Prototype:**

**NURBSCurve & operator=(const NURBSCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSCurve& curve**

The curve to assign.

## Class RendParams

See Also: [Class Atmospheric](#), [Class Effect](#), [Class IRenderElementMgr](#), [Class ToneOperator](#).

class RendParams

### Description:

This class has a set of data members, and these parameters are passed to the renderer when the renderer is opened. All methods of this class are implemented by the system.

### Data Members:

public:

#### **RendType rendType;**

The type of rendering to perform. See the [List of Render Types](#) for more information.

#### **BOOL isNetRender;**

Determines if this is a render on a network slave.

#### **BOOL fieldRender;**

If TRUE the image will be field rendered; otherwise frame rendered.

#### **int fieldOrder;**

The field order used. One of the following values:

**0** specifies even.

**1** specifies odd.

#### **TimeValue frameDur;**

This is used, for example, by video post. In video post you can stretch time. A video post frame might be 1/2 frame long for example. This data member defines the duration of one frame in TimeValue units.

#### **BOOL colorCheck;**

Determines if the color is ranged checked.

#### **int vidCorrectMethod;**

Video correction method. One of the following values:

**0** specifies FLAG (with black).

**1** specifies SCALE\_LUMA (scale luminance).

**2** specifies SCALE\_SAT (scale saturation).

**int ntscPAL;**

Determines if the color is range checked using NTSC or PAL standards. One of the following values:

**0** specifies NTSC.

**1** specifies PAL.

**BOOL superBlack;**

If TRUE Super Black is used.

**int sbThresh;**

Specifies the Super Black threshold.

**BOOL rendHidden;**

If TRUE hidden objects are rendered.

**BOOL force2Side;**

If TRUE two sided materials are used for all items in the scene.

**BOOL inMtlEdit;**

If TRUE the rendering is taking place in the material editor.

**float mtlEditTile;**

If rendering is taking place in the material editor, scale tiling.

**BOOL mtlEditAA;**

If TRUE antialiasing should be done in the material editor.

**BOOL multiThread;**

This is used internally.

**BOOL useEnvironAlpha;**

If TRUE one should use alpha from the environment map.

**BOOL dontAntialiasBG;**

If the low-order bit is set don't antialias against the background (this is often used for 'sprites' in video games). For 3ds max 1.1 and 1.2 (in 2.0 and later see **scanBandHeight** below), this parameter may also be used to access the height of the abuffer in scan lines. This may be obtained using the following syntax:

**abufBandHeight = rendpar.dontAntialiasBG>>8**

**BOOL useDisplacement;**

The apply displacement mapping setting.

**bool useRadiosity;**

This data member is available in release 4.0 and later only.  
Indicates if radiosity should be included in rendering.

**bool computeRadiosity;**

This data member is available in release 4.0 and later only.  
Indicates if radiosity should be computed before rendering.

**Texmap \*envMap;**

The environment map. This may be NULL.

**Atmospheric \*atmos;**

The atmosphere effects. This may be NULL. To the renderer it looks like there is only one atmosphere. You can use this atmosphere and it will go ahead and call all the individual atmospheric effects for you.

**Effect \*effect;**

This data member is available in release 3.0 and later only.  
The post-processing effects. This may be NULL if there aren't any.

**RadiosityEffect\* pRadiosity;**

This data member is available in release 4.0 and later only.  
Points to the radiosity effect.

**ToneOperator\* pToneOp;**

This data member is available in release 4.0 and later only.  
Points to the tone operator if present. This may be NULL

**TCHAR biFileName[MAX\_PATH];**

This data member is available in release 4.0 and later only.  
The bitmap output file name.

**TimeValue firstFrame;**

This data member is available in release 2.0 and later only.  
This is the first frame that will be rendered. This lets **Open()** know the first frame that will be rendered, so it will not have to evaluate at frame 0 when building.

**int scanBandHeight;**

This data member is available in release 2.0 and later only.  
This is the height of a scan band (for the default 3ds max scanline renderer).

**ULONG extraFlags;**

This data member is available in release 2.0 and later only.

## **RENDER\_HIDE\_FROZEN**

This option is available in release 4.0 and later only.

Instruct the renderer to hide frozen objects

### **int width;**

The image height.

### **int height;**

The image width.

### **BOOL filterBG;**

The filter background.

### **Methods:**

#### **Prototype:**

**RendParams();**

#### **Remarks:**

Constructor. The initialization can be seen in the header file RENDER.H.

#### **Prototype:**

**RenderMode GetRenderMode();**

#### **Remarks:**

This method is available in release 4.0 and later only.

Returns the rendering mode. One of the following values:

#### **RM\_Default**

This is being used for a normal rendering.

#### **RM\_IReshade**

The render is being used for interactive reshading.

#### **Prototype:**

**void SetRenderElementMgr(IRenderElementMgr  
\*pIRenderElementMgr);**

#### **Remarks:**

This method is available in release 4.0 and later only.

Sets the render element manager used.

**Parameters:****IRenderElementMgr \*pIRenderElementMgr**

Points to the render element manager to set.

**Prototype:****IRenderElementMgr \*GetRenderElementMgr();****Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the render element manager interface.

**Prototype:****virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);****Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.**Parameters:****int cmd**

The index of the command to execute.

**ULONG arg1=0**Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

# Class FrameRenderParams

See Also: [Class RenderParams](#), [Class Color](#), [Class Point2](#).

```
class FrameRenderParams : public BaseInterfaceServer
```

## Description:

This is passed to the renderer on every frame. This provides information about the ambient light color, the background color, the duration of one frame, etc.

## Data Members:

public:

### **Color ambient;**

The ambient light color.

### **Color background;**

The background color.

### **Color globalLightLevel;**

This data member is available in release 2.0 and later only.

This is a multiplier that scales the brightness of all scene lights: it doesn't affect the ambient light level.

### **float frameDuration;**

The duration of one frame in units of current frames. This describes how much scene time is used by one (video) frame time. For instance, in Video Post, you can scale time so it's stretched out or compressed. In this case, you may be rendering one frame to video, but because the scene is being run at a faster speed than normal, you in fact see say 2.5 frames of the scene. Things such as field rendering or motion blur must know about this so they know how long a frame is in terms of the time of the scene.

### **float relSubFrameDuration;**

This data member is available in release 2.0 and later only.

This is the relative fraction of **frameDuration** used by a subframe. Within the **frameDuration**, if you're sub-dividing the rendering up into multiple frames for scene motion blur, this may be used.

For instance, say the duration (frames) is set to 0.5 and the duration subdivisions is 5, then this data member would be 0.1. This means that each subframe is effectively covering only 1/10th of the frame duration.

This value is always less than 1.0.

**int regxmin, regxmax;**

These values were members of Class RendParams prior to release 3.

The x min and max boundary values for render region or render blowup in device coordinates.

**int regymin, regymax;**

These values were members of Class RendParams prior to release 3.

The y min and max boundary values for render region or render blowup in device coordinates.

**Point2 blowupCenter;**

This parameter is available in release 3.0 and later only.

The 2D point at the center of the render blowup region.

**Point2 blowupFactor;**

This parameter is available in release 3.0 and later only.

The X and Y scale factors for render blowup.

**Methods:**

**Prototype:**

**FrameRendParams();**

**Remarks:**

Constructor. The **frameDuration** is set to 1.0 and **relSubFrameDuration = 1.0f**.

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);**

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:**

**int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

## Class ObjLightDesc

See Also: [Class LightDesc](#), [Class LightRayTraversal](#), [Class INode](#), [Class Point3](#), [Class Matrix3](#), [Class Ray](#), [Class RendContext](#), [Class ShadeContext](#), [Class RenderInstance](#).

```
class ObjLightDesc : public LightDesc
```

### Description:

A light must be able to create one of these objects to give to the renderer. As the renderer is getting ready to render, it will ask for one of these from each of the lights. The **Illuminate()** method (inherited from **LightDesc**) is called by the renderer to illuminate a surface point.

There is an **ObjLightDesc** for every instance of the light. The renderer will ask each light object to produce one of these **ObjLightDescs**. It will then set this data up in the node's render data (See [Class RenderData](#)). For example in 3ds max's volume light implementation of **Atmospheric::Update()** it goes through its node references to lights and calls **GetRenderData()**. It then casts this as an **ObjLightDesc**. This is how a atmosphere effect can get access to these descriptors at render time.

### Data Members:

This data will be set up by the default implementation of **Update()**.

#### **public:**

#### **LightState ls;**

The light state structure. See [Structure LightState](#).

#### **INode \*inode;**

This parameter is the **INode** pointer of the instance of the light that created this **ObjLightDesc**.

#### **BOOL uniformScale;**

This indicates if the light's scale is uniform. TRUE if uniform; otherwise FALSE. This saves some steps in the renderer if the scale is uniform.

#### **Point3 lightPos;**

The position of the light in camera space.

#### **Matrix3 lightToWorld;**

This is effectively the light node's object TM. This matrix will transform

points from light space to world space.

**Matrix3 worldToLight;**

This matrix will transform points from world space to light space. This is the inverse of above.

**Matrix3 lightToCam;**

This matrix will transform points from light space to camera space. This is updated in **UpdateViewDepParams()**.

**Matrix3 camToLight;**

This matrix will transform points from camera space to light space. This is updated in **UpdateViewDepParams()**. For example, the renderer would have points in camera space. To figure out if a point was in shadow it would transform the point from camera space to light space using this matrix. It could then look in the shadow buffer to see if the point was in shadow.

**int renderNumber;**

This data member is available in release 3.0 and later only.

This is set by the renderer. It is used in

**RenderInstance::CastsShadowsFrom()**. This is a number used by the renderer to identify the lights so it can quickly determine if a given light casts shadows from a given object. It is for use by the renderer.

**Methods:**

**Prototype:**

**ObjLightDesc(INode \*n);**

**Remarks:**

Constructor. The **inode** data member is initialized to **n**.

**Prototype:**

**virtual ~ObjLightDesc();**

**Remarks:**

Destructor.

**Prototype:**

**virtual ExclList\* GetExclList()**

**Remarks:**

Implemented by the Plug-In.  
Retrieves the light's exclusion list.

**Return Value:**

See [Class NameTab](#).

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual int Update(TimeValue t, const RendContext &rc,  
RenderGlobalContext *rgc, BOOL shadows, BOOL  
shadowGeomChanged);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method is called once per render to update the light state for things that depend on the position of objects and lights in world space. A plug-in light could update any data it would need to here. The default implementation is shown below.

**Parameters:****TimeValue t**

The time of the render.

**const RendContext &rc**

See [Class RendContext](#).

**RenderGlobalContext \*rgc**

This pointer may be used to retrieve information about the global rendering environment.

**BOOL shadows**

TRUE if shadows are turned on (in the render parameters, not the light parameters); otherwise FALSE.

**BOOL shadowGeomChanged**

This tells the Update procedure that the geometry of the objects that are shadowed by the light has changed (TRUE) or not (FALSE). If it is a shadow

buffer, **shadowGeomChanged == TRUE** means it has to re-render the shadow buffer, **shadowGeomChanged == FALSE** means it can use the shadow buffer from the previous frame.

**Return Value:**

The value return should normally be 1. A returned value of 0 means an error has occurred (such as out of memory) and the render will be halted.

**Default Implementation:**

```
int ObjLightDesc::Update(TimeValue t, const RendContext& rc,
RenderGlobalContext *rgc, BOOL shadows, BOOL
shadowGeomChanged) {
  if (inode) {
    Interval valid;
    ObjectState os = inode->EvalWorldState(t);
    assert(os.obj->SuperClassID()==LIGHT_CLASS_ID);
    LightObject* lob = (LightObject *)os.obj;
    lob->EvalLightState(t, valid, &ls);
    lightToWorld = inode->GetObjTMAfterWSM(t);
    worldToLight = Inverse(lightToWorld);
    uniformScale = IsUniformScale(lightToWorld);
    affectDiffuse = ls.affectDiffuse;
    affectSpecular = ls.affectSpecular;
    ambientOnly = ls.ambientOnly;
  } else {
    uniformScale = TRUE;
    lightToWorld.IdentityMatrix();
    worldToLight.IdentityMatrix();
  }
  return 1;
}
```

**Prototype:**

```
virtual void UpdateGlobalLightLevel(Color globLightLevel);
```

**Remarks:**

This method is available in release 3.0 and later only.

Implemented by the Plug-In.

This method is called to update the light state that depends on the global light level.

**Parameters:**

**Color globLightLevel**

The global light level.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int UpdateViewDepParams(const Matrix3& worldToCam);
```

**Remarks:**

Implemented by the Plug-In.

This method is called to update the light state that depends on the view matrix. This is used to cache certain computed quantities that are dependent on where you are looking from. In a given scene at a given time, the system may render from several viewpoints. This is because of things like reflection maps and mirrors that need to get rendered. This method is called for each of these different viewpoints.

**Parameters:**

**const Matrix3& worldToCam**

The world space to camera space transformation matrix.

**Prototype:**

```
virtual Point3 LightPosition();
```

**Remarks:**

This method is from **LightDesc**. Here it provides a default implementation returning the **lightPos** data member.

**Default Implementation:**

```
{ return lightPos; }
```

**Prototype:**

```
virtual void TraverseVolume(ShadeContext& sc, const Ray &ray,  
int samples, float tStop, float attenStart, float attenEnd, DWORD  
flags, LightRayTraversal *proc);
```

**Remarks:**

Implemented by the Plug-In.

This function traverses a ray through the light volume. This method is implemented by plug-in lights.

Consider how the 3ds max atmospheric effects like the volume lights use this information. For each light the atmospheric effect is bound to, it calls the this method (**TraverseVolume()**) on the light. The volume light atmospheric effect passes a callback to this method (**proc**). The light then calls the **Step()** method of the callback for each partial segment of the ray. Given the illumination on the segment it computes the fog density over that segment.

The density may be constant if noise is not turned on, or it may change if noise is turned on. Using the fog density and the illumination it computes the light reflected off the atmosphere for the segment.

**Parameters:****ShadeContext& sc**

This is the **ShadeContext** passed into the **Shade()** method of the Atmospheric effect. The shade context passed in should only be used for state (like are shadows globally disabled). The position, normal, etc. serve no purpose.

**const Ray &ray**

Defines the world space ray that will be traversed.

**int samples**

The number of samples to sample along the ray. A reasonable range is from 25-100. This is more or less the suggested number of times the **proc->Step()** callback will be called. It is not precisely however because the system may take greater or fewer steps than specified as it needs to.

**float tStop**

This is the end of the **ray**. This is the point at which the traversal will stop (**ray.p+tStop\*ray.dir**). Note that the traversal can terminate earlier if the callback returns FALSE.

### **float attenStart**

Specifies a percent of the light attenuation distances that should be used for lighting during the traversal. This is used so a light can have an attenuation set to a certain percent, and then have the volume light be attenuated at a different point.

### **float attenEnd**

This specifies the ending percent of the light attenuation distances that should be used for lighting during the traversal.

### **DWORD flags**

There are three ways the shadow maps can be sampled. If none of these flags are set, the shadow map is sampled directly (this is the fastest). One of the following values:

#### **TRAVERSE\_LOWFILTSHADOWS**

This is a simple filtering where the system samples a point in the shadow map and then some of the neighboring points. This corresponds to 'Medium' in the Volume Light user interface (a value of 0 for flags is 'Low' -- just sampling the shadow map with no filtering at all).

#### **TRAVERSE\_HIFILTSHADOWS**

This is a higher resolution sampling. This corresponds to 'High' in the Volume Light user interface.

#### **TRAVERSE\_USESAMPLESIZE**

This produces the highest quality. This corresponds to 'Use Light Sample Range' in the Volume Light user interface. This is like a box filter, but also takes into consideration the position of the point within the pixel to do additional weighting.

### **LightRayTraversal \*proc**

A developer derives a class from **LightRayTraversal** and implements the **Step()** method. A pointer to it is passed here as the callback object.

### **Default Implementation:**

```
{}
```

# Class RenderInstance

See Also: [Class Mtl](#), [Class LightDesc](#), [Class INode](#), [Class Object](#), [Class Mesh](#), [Class Matrix3](#), [Class Box3](#), [Class Point3](#), [Class Interval](#).

class RenderInstance

## Description:

This class is available in release 2.0 and later only.

This class provides information about a single node being rendered. This includes information such as the mesh of the object, its material, unique node ID, object space bounding extents, number of lights affecting it, material requirements, and normals and vertex coordinates in various spaces (object and camera).

## Data Members:

public:

### **ULONG flags;**

The flags that describe the properties of this instance. See [List of Render Instance Flags](#).

### **Mtl \*mtl;**

This is the material from the node.

### **float wireSize;**

The wireframe size.

### **Mesh \*mesh;**

The mesh to be rendered. This is the result of **GeomObject::GetRenderMesh()**.

### **float vis;**

Object visibility (between 0.0 and 1.0). This is the value the visibility track evaluates to at a particular time.

### **int nodeID;**

A unique ID associated with the node. It's unique within the scene during a render.

### **int objMotBlurFrame;**

This will be equal to **NO\_MOTBLUR** for all non-blurred objects. For blurred objects, it takes on the values (**0..nBlurFrames-1**) for the successive

blur-instances.

**int objBlurID;**

The purpose of this is to differentiate blur-instances generated from different nodes. All the blur-instances for an object-motion-blurred object will have the same **objBlurID**. This is as distinct from **nodeID**, which is different for every instance. This makes it possible to easily avoid intersecting a ray with all blur-instances for an object. If

**RenderGlobalContext::IntersectWorld()** is being used, then passing in the **objBlurID** for the parameter **skipID** will have this effect.

The basic technique is this: When reflecting or refracting rays, and object motion blur is enabled, choose sub-frame times randomly for the different rays (effectively giving a coarse stochastic sampling of time).

**Matrix3 objToWorld;**

This matrix can be used to transform object coordinates to world coordinates.

**Matrix3 objToCam;**

This matrix can be used to transform object coordinates to camera coordinates.

**Matrix3 normalObjToCam;**

This matrix can be used for transforming surface normals from object space to camera space.

**Matrix3 camToObj;**

This matrix can be used to transform camera coordinates to object coordinates.

**Box3 obBox;**

The object space extents of the object being rendered.

**Point3 center;**

The object bounding sphere center (in camera coordinates)

**float radsq;**

The square of the bounding sphere's radius.

**Methods:**

**Prototype:**

**void SetFlag(ULONG f, BOOL b);**

**Remarks:**

Sets the specified flag(s) to the state passed.

**Parameters:**

## **ULONG f**

The flags to set. See [List of Render Instance Flags](#).

## **BOOL b**

The state to set; TRUE for on; FALSE for off.

### **Prototype:**

```
void SetFlag(ULONG f);
```

### **Remarks:**

Sets the specified flag(s) to on.

### **Parameters:**

**ULONG f**

The flags to set. See [List of Render Instance Flags](#).

### **Prototype:**

```
void ClearFlag(ULONG f);
```

### **Remarks:**

Clears the specified flag(s).

### **Parameters:**

**ULONG f**

The flags to set to zero. See [List of Render Instance Flags](#).

### **Prototype:**

```
BOOL TestFlag(ULONG f);
```

### **Remarks:**

Returns TRUE if the specified flag(s) are set; otherwise FALSE.

### **Parameters:**

**ULONG f**

The flags to set to zero. See [List of Render Instance Flags](#).

### **Prototype:**

```
virtual RenderInstance *Next()=0;
```

**Remarks:**

Returns a pointer to the next in **RenderInstance** in the list. A pointer to the first element in the list may be retrieved from **RenderGlobalContext::InstanceList()**.

**Prototype:**

**virtual Interval MeshValidity()=0;**

**Remarks:**

Returns the validity interval of the mesh of this render instance.

**Prototype:**

**virtual int NumLights()=0;**

**Remarks:**

Returns the number of lights affecting the node.

**Prototype:**

**virtual LightDesc \*Light(int n)=0;**

**Remarks:**

Returns a pointer to the LightDesc for the 'i-th' light affecting the node.

**Parameters:**

**int n**

Specifies which light.

**Prototype:**

**virtual BOOL CastsShadowsFrom(const ObjLightDesc& lt)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if this particular instance will cast shadows from the particular light based on the light's Exclusion/Inclusion list; FALSE if it won't cast shadows.

**Parameters:**

**const ObjLightDesc& lt**

Describes the light. See [Class ObjLightDesc](#).

**Prototype:**

**virtual INode \*GetINode()=0;**

**Remarks:**

Returns the INode pointer for the instance.

**Prototype:**

**virtual Object \*GetEvalObject()=0;**

**Remarks:**

Returns a pointer to the evaluated object for the instance. You can use this to get more information about the type of object being rendered. For instance you could look at the **Class\_ID** and recognize it as a sphere, a box, a torus, etc.

**Prototype:**

**virtual ULONG MtlRequirements(int mtlNum, int faceNum)=0;**

**Remarks:**

Returns the material requirements of the material assigned to the node. See [List of Material Requirement Flags](#).

**Parameters:**

**int mtlNum**

Specifies the number of the sub-material whose requirements should be returned. A value of **-1** may be passed to return a value generated by looping over all the sub-materials and ORing together the requirements.

**int faceNum**

This parameter is available in release 4.0 and later only.

This is the integer face number for objects which support material per face (if flag **INST\_MTL\_BYFACE** is set). See [Class IChkMtlAPI](#).

**Prototype:**

**virtual Point3 GetFaceNormal(int faceNum)=0;**

**Remarks:**

Returns the geometric normal of the specified face in object space.

**Parameters:**

**int faceNum**

Zero based index of the face whose normal is returned.

**Prototype:**

**virtual Point3 GetFaceVertNormal(int faceNum, int vertNum)=0;**

**Remarks:**

Returns the vertex normal of the specified face in camera coordinates.

**Parameters:**

**int faceNum**

Zero based index of the face in the mesh.

**int vertNum**

Zero based index of the vertex in the face.

**Prototype:**

**virtual void GetFaceVertNormals(int faceNum, Point3 n[3])=0;**

**Remarks:**

Returns the three vertex normals of the specified face in camera coordinates.

**Parameters:**

**int faceNum**

Zero based index of the face in the mesh.

**Point3 n[3]**

The normals are returned here.

**Prototype:**

**virtual Point3 GetCamVert(int vertnum)=0;**

**Remarks:**

Returns the coordinate for the specified vertex in camera coordinates

**Parameters:**

**int vertnum**

The zero based index of the vertex in the mesh.

**Prototype:**

**virtual void GetObjVerts(int fnum, Point3 obp[3])=0;**

**Remarks:**

Returns the vertices of the specified face in object coordinates.

**Parameters:**

**int fnum**

Zero based index of the face in the mesh.

**Point3 obp[3]**

The three vertices of the face in object coordinates.

**Prototype:**

**virtual void GetCamVerts(int fnum, Point3 cp[3])=0;**

**Remarks:**

Returns the vertices of the specified face in camera (view) coordinates.

**Parameters:**

**int fnum**

Zero based index of the face in the mesh.

**Point3 cp[3]**

The three vertices of the face in camera coordinates.

**Prototype:**

**virtual Mtl \*GetMtl(int faceNum)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Objects can provide a material as a function of face number via the interface provided by [Class IChkMtlAPI](#). This method will return

**RenderInstance::mtl** if flag **INST\_MTL\_BYFACE** is not set. If

**INST\_MTL\_BYFACE** is set it will return the proper by-face material. See

[List of Render Instance Flags](#).

**Parameters:**

## **int faceNum**

The zero based index of the face in the mesh.

Objects can provide a material as a function of face number via the IChkMtlAPI interface (chkmtlapi.h).

### **Prototype:**

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0)=0;
```

### **Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

### **Parameters:**

#### **int cmd**

The index of the command to execute.

#### **ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

#### **ULONG arg2=0**

Optional argument 2.

#### **ULONG arg3=0**

Optional argument 3.

### **Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

### **Default Implementation:**

```
{ return 0; }
```

## Class RendContext

See Also: [Class ObjLightDesc](#).

class RendContext

### Description:

This class is passed into the method **ObjLightDesc::Update()**. The methods of this class are implemented by the 3ds max scanline renderer. Developer creating other renderer plug-ins may choose to implement the methods of this class if they wish to use the same architecture. Developers who wish to take advantage of the 3ds max volumetric light effects should implement the methods of this class. The volumetric lights are set up to work with this mechanism.

### Methods:

#### Prototype:

**virtual int Progress(int done, int total) const;**

#### Remarks:

Implemented by the Plug-In.

This method is used to update the progress bar and check the keyboard and mouse for user cancellation. A plug-in renderer should implement this method by calling the **RendProgressCallback::Progress()** method on the **RendProgressCallback** passed in to the **Renderer::Render()** method.

#### Parameters:

**int done**

This is the number completed so far.

**int total**

This is the total number of things to complete.

#### Return Value:

Nonzero to continue; zero if the user has canceled.

#### Default Implementation:

```
{ return 1; }
```

#### Prototype:

**virtual Color GlobalLightLevel() const = 0;**

**Remarks:**

This method is available in release 2.0 and later only.

This is a multiplier that scales the brightness of all scene lights: it doesn't affect the ambient light level. It is included in **RenderContext** so the lights can use it to multiply times the light's color.

# Class PatchMesh

See Also: [Class Patch](#), [Class PatchVec](#), [Class PatchEdge](#), [Class PatchVert](#), [Class BitArray](#), [Class TessApprox](#), [Class PatchTVert](#), [Working with Patches](#).

```
class PatchMesh : public BaseInterfaceServer
```

## Description:

A patch mesh can be made up of any number of patches. Each of these patches can be three or four sided. Each edge of a patch can only be used by either one patch (which makes it an open edge) or two patches (which makes it a transitional edge between the two).

All methods of this class are implemented by the system.

## Friend Classes:

```
friend class Patch;
```

## Method Groups:

The following hyperlinks take you to the start of groups of related methods:

[Constructors / Destructor](#)

[Vert Access](#)

[TVert Access](#)

[Vec Access](#)

[Patch Access](#)

[Edge Access](#)

[RVert Access](#)

[Materials / Mapping](#)

[buildLinkages\(\) / computeInteriors\(\) / ApplyConstraints\(\) / GetEdge\(\)](#)

[Tri / Quad Patch Creation](#)

[Get/Set Mesh Steps and Adaptive Switch/TessApprox](#)

[Render / Snap / Hit Test](#)

[Display Flag Access](#)

[Selection Access](#)

[Data Flow Evaluation](#)

[Dump\(\)](#)

[Operators](#)

## Data Members:

```
public:
```

## Topology:

**int numVerts;**

The number of vertices.

**int numVecs;**

The number of vectors.

**int numPatches;**

The number of patches.

**int numEdges;**

The number of edges.

**Patch \*patches;**

The list of patches.

**PatchVec \*vecs;**

The list of PatchVecs.

**PatchEdge \*edges;**

The list of PatchEdges.

**Tab<HookPoint> hooks;**

This data member is available in release 3.0 and later only.

This table is used internally

**Geometry:**

**PatchVert \*verts;**

The list of PatchVerts.

**Texture coordinate assignment:**

**Tab<int> numTVerts;**

This data member is available in release 3.0 and later only.

A table containing the number of texture vertices for each channel.

**Tab<PatchTVert \*> tVerts;**

This data member is available in release 3.0 and later only.

A table containing pointers to the texture vertices for each channel.

Previous to R4.0 this was a Tab<UVVert \*>.

**Tab<TVPatch \*> tvPatches;**

This data member is available in release 3.0 and later only.

A table containing pointers to the texture vertex patches for each channel.

## **Material assignment**

**DWORD mtlIndex;**

The object level material.

## Selection

**BitArray vertSel;**

The selected vertices.

**BitArray edgeSel;**

The selected edges.

**BitArray patchSel;**

The selected patches.

**int bezVecVert;**

This is used internally.

**DWORD dispFlags;**

The display attribute flags. See [List of Patch Display Flags](#).

**DWORD selLevel;**

The current selection level. One of the following values:

**PATCH\_OBJECT** - Object level.

**PATCH\_VERTEX** - Vertex level.

**PATCH\_PATCH** - Patch level.

**PATCH\_EDGE** - Edge level.

**int cacheSteps;**

This data member is available in release 3.0 and later only.

The meshSteps used for the cache.

**BOOL cacheAdaptive;**

This data member is available in release 3.0 and later only.

The adaptive switch used for the mesh cache.

**Tab<Point3> extrudeDeltas;**

This data member is available in release 3.0 and later only.

This data member is for internal use only.

**Tab<ExtrudeData> extrudeData;**

This data member is available in release 3.0 and later only.

This data member is for internal use only.

**BitArray bevelEdges;**

This data member is available in release 3.0 and later only.

This data member is for internal use only.

## **Tab<float> edgeDistances;**

This data member is available in release 3.0 and later only.

This data member is for internal use only.

## **Methods:**

public:

## **Constructors / Destructor**

### **Prototype:**

**PatchMesh();**

### **Remarks:**

Constructor. Initializes data members (see **Init()** below).

### **Prototype:**

**PatchMesh(PatchMesh& fromPatch);**

### **Remarks:**

Constructor. This **PatchMesh** is initialized from the specified **PatchMesh**.

### **Prototype:**

**void Init();**

### **Remarks:**

Initializes the data members to default values:

```
meshSteps = 5;  
adaptive = FALSE;  
rVerts = NULL;  
cacheGW = NULL;  
numVerts = 0;  
numVecs = 0;  
numPatches = 0;  
numEdges = 0;  
patches = NULL;  
edges = NULL;  
numTVerts.ZeroCount();
```

```
tvPatches.ZeroCount();
tVerts.ZeroCount();
vecs = NULL;
verts = NULL;
mtlIndex = 0;
flags = 0;
snapVCt = 0;
snapPCt = 0;
snapV = NULL;
snapP = NULL;
dispFlags = DISP_LATTICE;
selLevel = PATCH_OBJECT;
bezVecVert = -1;
bdgBox.Init();
cacheSteps = -9999;
cacheAdaptive = -9999;
prodTess.u = 5;
prodTess.v = 5;
prodTess.dist = 2.0f;
prodTess.ang = 10.0f;
prodTess.edge = 1.0f;
prodTess.view = TRUE;
dispTess.type = TESS_CURVE;
dispTess.v = 5;
dispTess.dist = 2.0f;
dispTess.ang = 10.0f;
dispTess.edge = 1.0f;
dispTess.view = TRUE;
viewTess.u = 5;
viewTess.v = 5;
viewTess.view = FALSE;
viewTess.ang = 20.0f;
viewTess.dist = 10.0f;
viewTess.edge = 10.0f;
```

**Prototype:**

**~PatchMesh();**

**Remarks:**

Destructor. Frees up allocated arrays.

**Prototype:**

**void CopyPatchDataFrom(PatchMesh &fromPatchMesh);**

**Remarks:**

This method is available in release 3.0 and later only.

The following is similar to operator=, but just takes the major components, not the display flags, selection level, etc.

**Parameters:**

**PatchMesh &fromPatchMesh**

The patch mesh to copy from.

**Vert Access****Prototype:**

**BOOL setNumVerts(int ct, BOOL keep = FALSE);**

**Remarks:**

Sets the number of vertices in the patch mesh.

**Parameters:**

**int ct**

The new number of vertices.

**BOOL keep = FALSE**

If TRUE any old vertices are copied to the new array; otherwise they are freed.

**Return Value:**

TRUE if the number of vertices was allocated; otherwise FALSE.

**Prototype:**

**int getNumVerts();**

**Remarks:**

Returns the number of vertices in the patch mesh.

**Prototype:**

```
void setVert(int i, const Point3 &xyz)
```

**Remarks:**

Sets the 'i-th' vertex.

**Parameters:**

**int i**

The index of the vertex to set.

**const Point3 &xyz**

The vertex location.

**Prototype:**

```
void setVert(int i, float x, float y, float z)
```

**Remarks:**

Sets the 'i-th' vertex.

**Parameters:**

**int i**

The index of the vertex to set.

**float x, float y, float z**

The vertex location.

**Prototype:**

```
PatchVert &getVert(int i)
```

**Remarks:**

Returns the 'i-th' vertex.

**Parameters:**

**int i**

The index of the vertex to retrieve.

**Prototype:**

**PatchVert \*getVertPtr(int i)**

**Remarks:**

Returns the address of the 'i-th' vertex.

**Parameters:**

**int i**

The index of the vertex.

## **TVert Access**

**Prototype:**

**BOOL setNumTVerts(int ct, BOOL keep=FALSE);**

**Remarks:**

Sets the number of mapping verts in the original mapping channel (channel 1).

**Parameters:**

**int ct**

The number of map vertices desired.

**BOOL keep = FALSE**

If TRUE, any existing mapping verts are copied over into the new array.

**Return Value:**

TRUE if successful, FALSE if unsuccessful.

**Prototype:**

**int getNumTVerts() const;**

**Remarks:**

Returns the number of mapping vertices in the original mapping channel (channel 1).

**Prototype:**

**BOOL setNumTVertsChannel(int mp, int ct, BOOL keep=FALSE);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the number of mapping vertices in the original TV map or vertex color channel.

**Parameters:**

**int mp**

Specifies the channel. If 0, the number of vertices in the original mapping channel (map channel 1) is set. If nonzero, the number of vertices in the vertex color channel (map channel 0) is set.

**int ct**

The number of map vertices desired.

**BOOL keep=FALSE**

If TRUE, any existing mapping verts are copied over into the new array.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**int getNumTVertsChannel(int mp);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the number of mapping verts in the original map or vertex colors.

**Parameters:**

**int mp**

If 0, the number of vertices in the original mapping channel (map channel 1) is returned. If nonzero, the number of vertices in the vertex color channel (map channel 0) is returned.

**Prototype:**

**PatchTVert \*mapVerts(int mp);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns a pointer to the list of PatchTVerts for the specified channel of this patch.

**Parameters:**

**int mp**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**Prototype:**

**BOOL setNumMapVerts(int mp, int ct, BOOL keep = FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the number of mapping verts in the specified mapping channel.

**Parameters:****int mp**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int ct**

The number of mapping verts desired.

**BOOL keep=FALSE**

If TRUE any old vertices are copied to the new array; otherwise they are freed.

**Return Value:**

TRUE if the number of vertices was allocated; otherwise FALSE.

**Prototype:**

**int getNumMapVerts(int mp);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of mapping verts in the specified mapping channel.

**Parameters:****int mp**

In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**Prototype:**

**void setTVert(int i, const UVVert &xyz)**

**Remarks:**

Sets the 'i-th' map vertex.

**Parameters:**

**int i**

The index of the map vertex to set.

**const UVVert &xyz**

The value to set.

**Prototype:**

**void setTVert(int i, float x, float y, float z)**

**Remarks:**

Sets the 'i-th' map vertex.

**Parameters:**

**int i**

The index of the map vertex to set.

**float x, float y, float z**

The values to set.

**Prototype:**

**void setTVertChannel(int channel, int i, const UVVert &xyz);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the 'i-th' map vertex of the specified channel.

**Parameters:**

**int channel**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The index of the map vertex to set.

**const UVVert &xyz**

The value to set.

**Prototype:**

**void setTVertChannel(int channel, int i, float x, float y, float z);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the 'i-th' map vertex of the specified channel.

**Parameters:**

**int channel**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The index of the map vertex to set.

**float x, float y, float z**

The values to set.

**Prototype:**

**void setTVPatchChannel(int channel, int i, TVPatch &tvpatch);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the 'i-th' map patch.

**Parameters:**

**int channel**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The index of the map vertex to set.

**TVPatch &tvpatch**

The map patch to set.

**Prototype:**

**void setTVPatch(int i, TVPatch &tvpatch);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified map patch.

**Parameters:**

**int i**

The zero based index of the texture patch to set.

**TVPatch &tv**

The map patch to set.

**Prototype:**

**void setMapVert(int mp, int i, const UVVert &xyz);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified mapping vertex in the channel passed.

**Parameters:**

**int mp**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The zero based index of the vert to set.

**const UVVert &xyz**

The vert to set.

**Prototype:**

**void setMapVert(int mp, int i, float x, float y, float z);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified mapping vertex in the channel passed.

**Parameters:**

**int mp**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The zero based index of the vert to set.

**float x**

The x coordinate of the vert to set.

**float y**

The y coordinate of the vert to set.

**float z**

The z coordinate of the vert to set.

**Prototype:**

```
void setMapPatch(int mp, int i, const TVPatch &ttp);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified mapping patch in the channel passed.

**Parameters:****int mp**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The zero based index of the TVPatch to set.

**const TVPatch &ttp**

The map patch to set.

**Prototype:**

```
PatchTVert &getMapVert(int mp, int i);
```

**Remarks:**

This method is available in release 3.0 and later only.

Previous to R4, this method returned an UVVert&.

Returns a reference to the specified mapping vert from the specified channel.

**Parameters:****int mp**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The zero based index of the vert to get.

**Prototype:**

**PatchTVert \*getMapVertPtr(int mp, int i);**

**Remarks:**

This method is available in release 3.0 and later only.

Previous to R4, this method returned an UVVert\*.

Returns a pointer to the specified mapping vert from the specified channel.

**Parameters:**

**int mp**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The zero based index of the vert to get.

**Prototype:**

**TVPatch &getMapPatch(int mp, int i);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to the specified map patch from the specified channel.

**Parameters:**

**int mp**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The zero based index of the map patch to get.

**Prototype:**

**PatchTVert &getTVert(int i)**

**Remarks:**

Previous to R4, this method returned an UVVert&.

Returns the 'i-th' map vertex.

**Parameters:****int i**

The index of the map vertex to retrieve.

**Prototype:****PatchTVert \*getTVertPtr(int i)****Remarks:**

Previous to R4, this method returned an UVVert\*.

Returns a pointer to the 'i-th' map vertex.

**Parameters:****int i**

The index of the map vertex.

**Prototype:****PatchTVert &getTVertChannel(int channel, int i);****Remarks:**

This method is available in release 2.0 and later only.

Previous to R4, this method returned an UVVert&.

Returns the 'i-th' map vertex of the specified channel.

**Parameters:****int channel**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The index of the map vertex to retrieve.

**Prototype:****PatchTVert \*getTVertPtrChannel(int channel, int i);****Remarks:**

This method is available in release 2.0 and later only.

Previous to R4, this method returned an UVVert\*.

Returns a pointer to the 'i-th' map vertex of the specified channel.

**Parameters:****int channel**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The index of the texture vertex.

**TVPatch Access****Prototype:****BOOL setNumTVPatches(int ct, BOOL keep=FALSE, int oldCt=0);****Remarks:**

Sets the number of map patches in the original TV channel. (Note that **setNumMapPatches()** is called from **setNumPatches()**, so this doesn't need to be called separately once a map channel is active.)

Note:

If the map patches are NULL and keep = TRUE they stay NULL.

If the map patches are NULL and keep = FALSE they are allocated, and map verts also init themselves from the main vert array.

If the map patches are non-NULL and ct = 0 they are set to NULL (and freed)

**Parameters:****int ct**

The number of map patches desired -- should match the number of patches.

**BOOL keep=FALSE**

If TRUE, existing map patches are copied into the new map patch array. oldCt should specify how many patches were around previously.

**int oldCt=0**

The old number of patches. This is important for determining how much to copy over when keep is TRUE.

**Return Value:**

TRUE if storage has been allocated and the number is set; otherwise FALSE.

**Prototype:**

```
BOOL setNumTVPatchesChannel(int channel, int ct, BOOL  
keep=FALSE, int oldCt=0);
```

**Remarks:**

This method is available in release 2.0 and later only.

Sets the number of map patches in the original TV or vertex color channels. (Note that **setNumMapPatches()** is called from **setNumPatches()**, so this doesn't need to be called separately once a map channel is active.)

**Note:**

If the map patches are NULL and keep = TRUE they stay NULL.

If the map patches are NULL and keep = FALSE they are allocated, and map verts also init themselves from the main vert array.

If the map patches are non-NULL and ct = 0 they are set to NULL (and freed)

**Parameters:****int channel**

If 0, the number of map patches in the original map channel are set. If nonzero, the number of map patches in the vertex color channel is set.

**int ct**

The number of map patches desired -- should match the number of patches.

**BOOL keep=FALSE**

The keep flag. See above.

**int oldCt=0**

The old number of patches. This is important for determining how much to copy over when keep is TRUE.

**Return Value:**

TRUE if storage has been allocated and the number is set; otherwise FALSE.

**Prototype:**

```
BOOL setNumMapPatches(int channel, int ct, BOOL keep=FALSE, int  
oldCt=0);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the number of map patches in the specified map channel. (Note that this is called from **setNumPatches()**, so it doesn't need to be called separately once a map channel is active.)

Note:

If the map patches are NULL and keep = TRUE they stay NULL.

If the map patches are NULL and keep = FALSE they are allocated, and map verts also init themselves from the main vert array.

If the map patches are non-NULL and ct = 0 they are set to NULL (and freed).

**Parameters:**

**int channel**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int ct**

The number of map patches desired -- should match the number of patches.

**BOOL keep=FALSE**

If TRUE, existing map patches are copied into the new map patch array. oldCt should specify how many patches were around previously.

**int oldCt=0**

The old number of patches. This is important for determining how much to copy over when keep is TRUE.

**Return Value:**

TRUE if storage has been allocated and the number is set; otherwise FALSE.

**Prototype:**

**TVPatch &getTVPatchChannel(int channel, int i);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the 'i-th' texture patch structure of the specified channel.

**Parameters:**

**int channel**

The mapping channel. In this method, 0 is the vertex color channel, and channels 1 through MAX\_MESHMAPS-1 are the map channels.

**int i**

The index of the TVPatch.

**Prototype:**

**TVPatch &getTVPatch(int i);**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the 'i-th' texture patch from map channel 1.

**Parameters:**

**int i**

The index of the TVPatch.

## **Vec Access**

**Prototype:**

**BOOL setNumVecs(int ct, BOOL keep = FALSE);**

**Remarks:**

Sets the number of vectors.

**Parameters:**

**int ct**

The new vector count.

**BOOL keep = FALSE**

If TRUE any previous vectors are copied; otherwise they are freed.

**Return Value:**

TRUE if the number was allocated and set; otherwise FALSE.

**Prototype:**

**int getNumVecs();**

**Remarks:**

Returns the number of vectors.

**Prototype:**

**void setVec(int i, const Point3 &xyz)**

**Remarks:**

Sets the 'i-th' vector.

**Parameters:**

**int i**

The index of the vector to set.

**const Point3 &xyz**

The vector to set.

**Prototype:**

**void setVec(int i, float x, float y, float z)**

**Remarks:**

Sets the 'i-th' vector.

**Parameters:**

**int i**

The index of the vector to set.

**float x, float y, float z**

The vector values to set.

**Prototype:**

**PatchVec &getVec(int i)**

**Remarks:**

Returns the 'i-th' vector.

**Parameters:**

**int i**

The index of the vector to retrieve.

**Prototype:**

**PatchVec \*getVecPtr(int i)**

**Remarks:**

Returns the address of the 'i-th' vector.

**Parameters:****int i**

The index of the vector.

**Patch Access****Prototype:****BOOL setNumPatches(int ct, BOOL keep = FALSE);****Remarks:**

Sets the number of patches.

**Parameters:****int ct**

The new patch count.

**BOOL keep = FALSE**

If TRUE any previous patches are copied; otherwise they are freed.

**Return Value:**

TRUE if the number was allocated and set; otherwise FALSE.

**Prototype:****int getNumPatches();****Remarks:**

Returns the number of patches.

**Edge Access****Prototype:****BOOL setNumEdges(int ct, BOOL keep = FALSE);****Remarks:**

Sets the number of edges.

**Parameters:****int ct**

The new edge count.

**BOOL keep = FALSE**

If TRUE any previous edges are copied; otherwise they are freed.

**Return Value:**

TRUE if the number was allocated and set; otherwise FALSE.

**Prototype:**

**int getNumEdges();**

**Remarks:**

Returns the number of edges.

**RVert Access**

**Prototype:**

**PRVertex &getRVert(int i)**

**Remarks:**

This method is not currently used.

**Prototype:**

**PRVertex \*getRVertPtr(int i)**

**Remarks:**

This method is not currently used.

**Material / Mapping Access**

**Prototype:**

**void setMtlIndex(DWORD i)**

**Remarks:**

This method is no longer used.

**Prototype:**

**DWORD getMtlIndex()**

**Remarks:**

This method is no longer used.

**Prototype:**

**void setNumMaps(int ct, BOOL keep=TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

Set the number of texture maps used by this PatchMesh.

**Parameters:**

**int ct**

The number to use. This is a value between 2 and **MAX\_MESHMAPS-1**.

**BOOL keep=TRUE**

TRUE to keep the old mapping information after the resize; FALSE to discard it.

**Prototype:**

**void setMapSupport(int chan, BOOL init=TRUE);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will allocate mapping channels as needed. If the map channel is already present, no action is taken. Otherwise, the additional channels are created.

**Parameters:**

**int chan**

Specifies which channel. See [List of Mapping Channel Index Values](#). If zero (special vertex color channel) and **init** is TRUE, all vertex colors are initialized to white (1,1,1).

**BOOL init=TRUE**

If TRUE, the channel is initialized to match the PatchMesh's structure.

**Prototype:**

**int getNumMaps();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the number of mapping channels in use.

**Prototype:**

**BOOL getMapSupport(int mp);**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns TRUE if the specified mapping channel is supported; otherwise FALSE.

**Parameters:**

**int mp**

Specifies which channel. See [List of Mapping Channel Index Values](#).

**Prototype:**

**int NumMapChannels();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the maximum number of possible mapping channels.

**Prototype:**

**void ApplyUVWMap(int type,  
float utile, float vtile, float wtile,  
int uflip, int vflip, int wflip, int cap,  
const Matrix3 &tm, int channel=1);**

**Remarks:**

This method may be called to map this **PatchMesh** with UVW mapping coordinates.

**Parameters:**

**int type**

The mapping type. One of the following values:

**MAP\_PLANAR**

**MAP\_CYLINDRICAL**

**MAP\_SPHERICAL**

**MAP\_BALL**

**MAP\_BOX**

**float utile**

Number of tiles in the U direction.

**float vtile**

Number of tiles in the V direction.

**float wtile**

Number of tiles in the W direction.

**int uflip**

If nonzero the U values are mirrored.

**int vflip**

If nonzero the V values are mirrored.

**int wflip**

If nonzero the W values are mirrored.

**int cap**

This is used with **MAP\_CYLINDRICAL**. If nonzero, then any patch normal that is pointing more vertically than horizontally will be mapped using planar coordinates.

**const Matrix3 &tm**

This defines the mapping space. As each point is mapped, it is multiplied by this matrix, and then it is mapped.

**int channel=1**

This parameter is available in release 2.0 and later only.

This indicates which channel the mapping is applied to -- **channel==1** corresponds to the original texture channel. Note that this is a change from what it meant before release 3.0. Previously channel 1 referred to the color per vertex channel (and this parameter defaulted to 0).

**Prototype:**

**void ChangePatchToLinearMapping(int index);**

**Remarks:**

This method is available in release 4.0 and later only.

This method changes the mapping of a patch or all selected patches to linear.

**Parameters:**

**int index**

The index of the patch for which to change the mapping to linear. A value < 0 indicates all selected patches are to be changed to linear mapping.

**Prototype:**

**void ChangePatchToCurvedMapping(int index);**

**Remarks:**

This method is available in release 4.0 and later only.

This method changes the mapping of a patch or all selected patches to curved.

**Parameters:**

**int index**

The index of the patch for which to change the mapping to curved. A value < 0 indicates all selected patches are to be changed to curved mapping.

**Prototype:**

**BOOL ArePatchesLinearMapped(int index);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will check if one or all selected patches have linear mapping applied.

**Parameters:**

**int index**

The index of the patch for which to check if mapping is linear. A value < 0 indicates all selected patches are checked for linear mapping.

**Return Value:**

TRUE if the specified patch or selected patches have linear mapping applied, otherwise FALSE.

**Prototype:**

**BOOL ArePatchesCurvedMapped(int index);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will check if one or all selected patches have curved mapping applied.

**Parameters:****int index**

The index of the patch for which to check if mapping is curved. A value < 0 indicates all selected patches are checked for curved mapping.

**Return Value:**

TRUE if the specified patch or selected patches have curved mapping applied, otherwise FALSE.

**Prototype:**

**BOOL RecordTopologyTags();**

**Remarks:**

This method tags the points in the patch components to record our topology (this stores identifying values in the various aux2 fields in the Patch). This information can be used after topology-changing operations to remap information tied to vertices, edges and patches.

**Return Value:**

Returns TRUE if tagged successfully; otherwise FALSE.

**Prototype:**

**void Transform(Matrix3 &tm);**

**Remarks:**

This method is available in release 3.0 and later only.

Transforms the vertices and vectors of the patch mesh, re-computes the interior bezier points for each patch in the mesh and invalidates the geometry cache.

**Parameters:****Matrix3 &tm**

The matrix to transform with.

**Prototype:**

**BOOL Weld(float thresh, BOOL weldIdentical=FALSE, int startVert=0);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used internally.

**Prototype:**

**void DeletePatchParts(BitArray &delVerts, BitArray &delPatches);**

**Remarks:**

This method is available in release 3.0 and later only.

This is a method which may be used to delete sets of verts or patches.

**Parameters:**

**BitArray &delVerts**

A bit array with bits set for verts to delete.

**BitArray &delPatches**

A bit array with bits set for patches set to delete.

**Prototype:**

**void Subdivide(int type, BOOL propagate);**

**Remarks:**

This method is available in release 3.0 and later only.

Subdivides the selected edge or patch.

**Parameters:**

**int type**

One of the following values:

**SUBDIV\_EDGES**

Subdivides an edge.

**SUBDIV\_PATCHES**

Subdivides an entire patch

**BOOL propagate**

TRUE to propagate; FALSE to not propagate.

**Prototype:**

**void AddPatch(int type);**

**Remarks:**

This method is available in release 3.0 and later only.

Add a patch of the desired type to each selected edge that doesn't have two patches attached.

**Parameters:**

**int type**

One of the following values:

**PATCH\_TRI**

**PATCH\_QUAD**

**Prototype:**

**int AddHook();**

**Remarks:**

This method is available in release 3.0 and later only.

This method is obsolete and should not be used.

**Prototype:**

**int AddHook(int index);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is obsolete and should not be used.

**Prototype:**

**int AddHook(int vertIndex, int segIndex);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is obsolete and should not be used.

**Prototype:**

**int RemoveHook();**

**Remarks:**

This method is available in release 3.0 and later only.

This method is obsolete and should not be used.

**Prototype:**

**int UpdateHooks();**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used internally.

**Prototype:**

**void CreateExtrusion(int type = PATCH\_PATCH, BOOL  
edgeClone=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

This method will create all the necessary geometry for an extrusion operation.

**Parameters:**

**int type**

This option is available in release 4.0 and later only.

Specifies the extrusion type, either **PATCH\_PATCH** or **PATCH\_EDGE**.

If the extrusion type is set to **PATCH\_EDGE**, then the **edgeClone** parameter will tell the function to clone the selected edges prior to creating the extrusion geometry.

**BOOL edgeClone**

This option is available in release 4.0 and later only.

The edge clone flag. If set to **TRUE** the function will clone the selected edges prior to creating the extrusion geometry.

**Prototype:**

**Point3 AverageNormals(int type = PATCH\_PATCH);**

**Remarks:**

This method is available in release 3.0 and later only.

This method will allow you to obtain the average normal of all selected patches or edges.

**Parameters:****int type**

This option is available in release 4.0 and later only.

Specifies if the average normal return is based on all selected patches, using **PATCH\_PATCH**, or all selected edges using **PATCH\_EDGE**.

**Prototype:**

**Point3 PatchNormal(int index);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used internally.

**Prototype:**

**void MoveNormal(float amount, BOOL useLocalNorms, int type);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used internally.

**Prototype:**

**void FlipPatchNormal(int index);**

**Remarks:**

This method is available in release 4.0 and later only.

This method flips the normal of the specified patch. This is done by reordering the vertices. If the PatchMesh has textures assigned to it then the texture patches are processed as well.

**Parameters:****int index**

The index of the patch for which you want to flip the normal.

**Prototype:****void UnifyNormals(BOOL useSel);****Remarks:**

This method is available in release 4.0 and later only.

This method makes sure the patches in the operation set are all facing the same direction. This is determined by the vertex order around the edges of the patch.

**Parameters:****BOOL useSel**

If this parameter is set to TRUE, the operation set is the set of selected patches as indicated by the **patchSel** BitArray. If this parameter is set to FALSE, all patches are processed.

**Prototype:****void AutoSmooth(float angle, BOOL useSel, BOOL preventIndirectSmoothing);****Remarks:**

This method is available in release 4.0 and later only.

This method will perform automatic smoothing on the patch mesh. The smoothing angle is determined for each patch by computing the normals at each corner of the patch (using the corner vertex and the two edge vectors connected to that vertex), then averaging the normals of the two corners of each edge to arrive at a normal for that edge. These normals are used to determine whether adjacent patches are within the auto-smooth threshold angle.

**Parameters:****float angle**

The minimum angle between surface normals for smoothing to be applied, in radians.

**BOOL useSel**

If this parameter is set to TRUE then only the selected patches are smoothed.

**BOOL preventIndirectSmoothing**

TRUE to turn on; FALSE to leave it off. This matches the option in the

Smooth Modifier UI -- use this to prevent smoothing "leaks" when using this method. If you use this method, and portions of the patch mesh that should not be smoothed become smoothed, then try this option to see if it will correct the problem. Note that the problem it corrects is rare, and that checking this slows the automatic smoothing process.

**Prototype:**

**void CreateBevel();**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used internally.

**Prototype:**

**void Bevel(float amount, int smoothStart, int smoothEnd);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used internally.

**Prototype:**

**Point3 GetBevelDir(int patchVertID);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is for internal use.

**buildLinkages() / computeInteriors() / ApplyConstraints() /  
GetEdge()**

**Prototype:**

**BOOL buildLinkages(int patch = -1);**

**Remarks:**

This is an important method to call after putting together a **PatchMesh**. This method does the work to figure out how the **PatchMesh** is connected together, one patch to another. It determines which edges are used by which

patches and so on.

**Parameters:**

**int patch**

This option is available in release 4.0 and later only.

This optional parameter allows you to only update adjacency information for a single new patch. If the plugin code is adding new patches and not removing any others then you can call this method with the new patch index and the various adjacency information in vertices, vectors, and edges will be built. If you call this method for the entire object (no parameter or **patch** < 0) it will destroy all adjacency information and rebuild it from scratch.

**Return Value:**

TRUE if the patch mesh is valid, FALSE if it is not.

**Prototype:**

**void computeInteriors();**

**Remarks:**

This method computes the interior bezier points for each patch in the mesh. This method should be called after any modifications have been made to alter the **PatchMesh** (for example changes to point positions). If there are any automatic patches this will compute the interior vectors.

**Prototype:**

**void ApplyConstraints(BOOL selOnly = FALSE);**

**Remarks:**

This method may be called to apply the coplanar constraints to the patch mesh. The constraints may optionally only apply to selected vertices. There is a flag that may be set for a patch vertex (**PVERT\_COPLANAR**). For example, you can set this flag to make a vertex coplanar with its vectors. If this is done, then when this method is called, the patch code will then go through the **PatchMesh** and find the average plane that is used by the vertex and all the vectors associated with it. It will then constrain all the vectors to lie in this plane (by rotating them so that they lie on the plane). In this way there will be a consistent transition between the patches sharing the vertex.

**Parameters:**

**BOOL selOnly = FALSE**

If TRUE the constraints are only applied to the selected vertices; otherwise all vertices.

**Prototype:**

**int GetEdge(int v1, int v12, int v21, int v2, int p);**

**Remarks:**

This method is used internally.

**ClonePatchParts() / SingleEdgesOnly() / GetElement() / WeldEdges()**

**Prototype:**

**void ClonePatchParts(BitArray \*patches = NULL);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will copy the patches specified by the BitArray, or by the patch selection set if the BitArray **patches** pointer is NULL. This method is used by Editable Patch and Edit Patch to facilitate the shift-copy operations.

**Parameters:**

**BitArray \*patches**

The array containing the series of selected patches.

**Prototype:**

**BOOL SingleEdgesOnly();**

**Remarks:**

This method is available in release 4.0 and later only.

This method examines the selected edges and will return TRUE if all selected edges are used by only one single patch. If the method returns FALSE, there are no edges selected or any of the selected edges are used by more than one single patch.

**Prototype:****BitArray& GetElement(int index);****Remarks:**

This method is available in release 4.0 and later only.

This method will return a BitArray that defines the group of patches defining an element that contains the patch indicated by the specified index. A PatchMesh element is any set of patches sharing common vertices.

**Parameters:****int index**

The patch index for which to return the element.

**Prototype:****BOOL WeldEdges();****Remarks:**

This method is available in release 4.0 and later only.

This method will weld any edges which are selected and have the same endpoints.

**Return Value:**

TRUE if any welding took place, otherwise FALSE.

## Selection Weights

**Prototype:****float \*GetVSelectionWeights();****Remarks:**

This method is available in release 4.0 and later only.

This method provides direct access to the vertex weights array and is included to match a similar function in the Mesh class.

**Return Value:**

A pointer to the vertex weights array.

**Default Implementation:****{ return mpVertexWeights; }**

**Prototype:****void SupportVSelectionWeights();****Remarks:**

This method is available in release 4.0 and later only.

This method allocates a vertex weights array, if none is currently allocated.

This method is included to match a similar function in the Mesh class.

**Conversion to Mesh****Prototype:****void ComputeMesh(Mesh& msh, DWORD convertFlags);****Remarks:**

This method is available in release 4.0 and later only.

This method will produce the mesh version of the PatchMesh.

**Parameters:****Mesh& msh**

The mesh in which the resulting mesh should be stored.

**DWORD convertFlags**

The flags modifying the mesh process.

**PATCH\_CONVERT\_KEEPESEL**

This flag indicates that the subobject vertex and patch selections in the PatchMesh should be converted to subobject vertex and face selections in the mesh.

**PATCH\_CONVERT\_USESOFTSEL**

This flag indicates that soft selections should be used in the mesh to interpolate between selected and nonselected vertices. (Soft Selections cannot be based on edge or patch selection in the PatchMesh, only vertex selection.)

**Render / Snap / Hit Test****Prototype:****void render(GraphicsWindow \*gw, Material \*ma, RECT \*rp, int**

**compFlags, int numMat=1);**

**Remarks:**

Renders a patch mesh using the specified graphics window and array of materials.

**Parameters:**

**GraphicsWindow \*gw**

Points to the graphics window to render to.

**Material \*ma**

The list of materials to use to render the patch.

**RECT \*rp**

Specifies the rectangular region to render. If the patch mesh should be rendered to the entire viewport pass NULL.

**int compFlags**

One or more of the following flags:

**COMP\_TRANSFORM**

Forces recalculation of the model to screen transformation; otherwise attempt to use the cache.

**COMP\_IGN\_RECT**

Forces all polygons to be rendered; otherwise only those intersecting the box will be rendered.

**COMP\_LIGHTING**

Forces re-lighting of all vertices (as when a light moves); otherwise only re-light moved vertices

**COMP\_ALL**

All of the above flags.

**COMP\_OBJSELECTED**

If this bit is set then the node being displayed by this mesh is selected. Certain display flags only activate when this bit is set.

**COMP\_OBJFROZEN**

If this bit is set then the node being displayed by this mesh is frozen.

**int numMat=1**

The number of materials supported.

**Prototype:**

```
BOOL select(GraphicsWindow *gw, Material *ma, HitRegion *hr,  
int abortOnHit = FALSE, int numMat=1);
```

**Remarks:**

Checks the given HitRecord **hr** to see if it intersects the patch mesh object.

**Parameters:**

**GraphicsWindow \*gw**

Points to the graphics window to check.

**Material \*ma**

The list of materials for the patch mesh.

**HitRegion \*hr**

This describes the properties of a region used for the hit testing. See [Class HitRegion](#).

**int abortOnHit = FALSE**

If nonzero, the hit testing is complete after any hit; otherwise all hits are checked.

**int numMat=1**

The number of materials supported.

**Return Value:**

TRUE if the item was hit; otherwise FALSE.

**Prototype:**

```
void snap(GraphicsWindow *gw, SnapInfo *snap, IPoint2 *p,  
Matrix3 &tm);
```

**Remarks:**

Checks to see if there is a snap point near the given mouse point.

**Parameters:**

**GraphicsWindow \*gw**

The graphics window in which to check.

**SnapInfo \*snap**

This structure describes the snap settings used, and the results of the snap test. See [Structure SnapInfo](#).

**IPoint2 \*p**

The mouse point to check.

**Matrix3 &tm**

The object transformation matrix. This is the transformation to place the object into the world coordinate system.

**Prototype:**

```
BOOL SubObjectHitTest(GraphicsWindow *gw, Material *ma,  
HitRegion *hr,  
DWORD flags, SubPatchHitList& hitList, int numMat=1);
```

**Remarks:**

This method may be called to perform sub-object hit testing of the patch mesh.

**Parameters:****GraphicsWindow \*gw**

The graphics window associated with the viewport the patch mesh is being hit tested in.

**Material \*ma**

The list of materials for the patch mesh.

**HitRegion \*hr**

This describes the properties of a region used for the hit testing. See [Class HitRegion](#).

**DWORD flags**

Flags for sub object hit testing. One or more of the following values:

**SUBHIT\_PATCH\_SELOONLY**

Selected only.

**SUBHIT\_PATCH\_UNSELOONLY**

Unselected only.

**SUBHIT\_PATCH\_ABORTONHIT**

Abort hit testing on the first hit found.

**SUBHIT\_PATCH\_SELSSOLID**

This treats selected items as solid and unselected items as not solid. Treating an item as solid means the patch will be hit if the mouse is anywhere inside the patch region and not just over a visible edge.

## **SUBHIT\_PATCH\_VERTS**

Hit test vertices.

## **SUBHIT\_PATCH\_VECS**

Hit test vectors.

## **SUBHIT\_PATCH\_PATCHES**

Hit test patches.

## **SUBHIT\_PATCH\_EDGES**

Hit test edges.

## **SubPatchHitList& hitList**

See [Class SubPatchHitList](#).

## **int numMat=1**

The number of materials for the mesh.

### **Return Value:**

TRUE if the item was hit; otherwise FALSE.

## **Bounding Box Methods**

### **Prototype:**

**void buildBoundingBox();**

### **Remarks:**

Computes the bounding box of the patch mesh. The bounding box is stored with the patch mesh object, use **getBoundingBox()** to retrieve it.

### **Prototype:**

**Box3 getBoundingBox(Matrix3 \*tm=NULL);**

### **Remarks:**

Retrieves the bounding box of the patch mesh object.

### **Parameters:**

**Matrix3 \*tm=NULL**

The optional TM allows the box to be calculated in any space.

### **Prototype:**

**void renderGizmo(GraphicsWindow \*gw);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will render a gizmo version of the PatchMesh and is used primarily by Editable Patch and Edit Patch to facilitate the Show End Result feature.

**Parameters:**

**GraphicsWindow \*gw**

The graphics window associated with the viewport the patch mesh gizmo should be shown in.

## Data Flow Evaluation

**Prototype:**

**void InvalidateGeomCache();**

**Remarks:**

This method should be called when the **PatchMesh** changes. It invalidates the caches of the patch mesh.

**Prototype:**

**void FreeAll();**

**Remarks:**

Frees everything from the patch mesh.

**Prototype:**

**void ShallowCopy(PatchMesh \*amesh, ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation.

**Prototype:**

**void DeepCopy(PatchMesh \*amesh, ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation.

**Prototype:**

**void NewAndCopyChannels(ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation.

**Prototype:**

**void FreeChannels(ULONG\_PTR channels, int zeroOthers=1);**

**Remarks:**

This method is used internally in data flow evaluation.

## Display Flags Access

**Prototype:**

**void SetDispFlag(DWORD f);**

**Remarks:**

Sets the state of the specified display flags.

**Parameters:**

**DWORD f**

The flags to set. See [List of Patch Display Flags](#).

**Prototype:**

**DWORD GetDispFlag(DWORD f);**

**Remarks:**

Returns the state of the specified display flags.

**Parameters:**

**DWORD f**

The flags to get. See [List of Patch Display Flags](#).

**Prototype:**

**void ClearDispFlag(DWORD f);**

**Remarks:**

Clears the specified display flags.

**Parameters:**

**DWORD f**

The flags to clear. See [List of Patch Display Flags](#).

**Selection Access****Prototype:**

**BitArray& VertSel();**

**Remarks:**

Returns the bits representing the vertex selection status. See the Data Members above. See [Class BitArray](#).

**Prototype:**

**BitArray& PatchSel();**

**Remarks:**

Returns the bits representing the patch selection status. See the Data Members above. See [Class BitArray](#).

**Prototype:**

**BitArray& EdgeSel();**

**Remarks:**

Returns the bits representing the edge selection status. See the Data Members above. See [Class BitArray](#).

**Prototype:**

**BitArray VertexTempSel();**

**Remarks:**

Constructs a vertex selection list based on the current selection level. For example if the selection level is at object level all the bits are set. If the selection level is at vertex level only the selected vertex bits are set. See [Class BitArray](#).

## Tri / Quad Patch Creation

### Prototype:

```
BOOL MakeQuadPatch(int index, int va, int vab, int vba,  
int vb, int vbc, int vcb, int vc, int vcd, int vdc,  
int vd, int vda, int vad, int i1, int i2, int i3,  
int i4, DWORD sm);
```

### Remarks:

Create a quadrilateral patch given a patch index and a list of all the vertices, vectors, interiors, and a smoothing group.

### Parameters:

**int index** - The index of the patch to create ( $0 \leq \text{index} < \text{numPatches}$ ).

**int va** - The first vertex.

**int vab** - Vector ab.

**int vba** - Vector ba.

**int vb** - The second vertex.

**int vbc** - Vector bc.

**int vcb** - Vector cb.

**int vc** - The third vertex.

**int vcd** - Vector cd.

**int vdc** - Vector dc.

**int vd** - The fourth vertex.

**int vda** - Vector da.

**int vad** - Vector ad.

**int i1** - Interior 1.

**int i2** - Interior 2.

**int i3** - Interior 3.

**int i4** - Interior 4.

**DWORD sm** - The smoothing group.

### Return Value:

TRUE if the patch was created; otherwise FALSE.

### Prototype:

```
BOOL MakeTriPatch(int index, int va, int vab, int vba, int vb,
```

**int vbc, int vcb, int vc, int vca, int vac, int i1, int i2,  
int i3, DWORD sm);**

**Remarks:**

Create a triangular patch given a patch index and a list of all the vertices, vectors, interiors, and a smoothing group.

**Parameters:**

**int index** - The index of the patch to create ( $0 \leq \text{index} < \text{numPatches}$ ).

**int va** - The first vertex.

**int vab** - Vector ab.

**int vba** - Vector ba.

**int vb** - The second vertex.

**int vbc** - Vector bc.

**int vcb** - Vector cb.

**int vc** - The third vertex.

**int vca** - Vector ca.

**int vac** - Vector ac.

**int i1** - Interior 1.

**int i2** - Interior 2.

**int i3** - Interior 3.

**DWORD sm** - The smoothing group.

**Returns:**

TRUE if the patch was created; otherwise FALSE.

## **Get/Set Mesh Steps and Adaptive Switch/TessApprox**

**Prototype:**

**void SetMeshSteps(int steps);**

**Remarks:**

Sets the number of steps along each edge that determines how fine the mesh is generated off the patch.

**Parameters:**

**int steps**

The number of steps to set.

**Prototype:**

**int GetMeshSteps();**

**Remarks:**

Returns the number of mesh steps.

**Prototype:**

**void SetMeshStepsRender(int steps);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the Surface Render Steps setting.

**Parameters:**

**int steps**

The value to set.

**Prototype:**

**int GetMeshStepsRender();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the Surface Render Steps setting.

**Prototype:**

**void SetShowInterior(BOOL si);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the 'Show Interior Edges' value.

**Parameters:**

**BOOL si**

TRUE for on; FALSE for off.

**Prototype:**

**BOOL GetShowInterior();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the 'Show Interior Edge' setting; TRUE if on; FALSE if off.

**Prototype:**

```
void SetAdaptive(BOOL sw);
```

**Remarks:**

This is currently not used. Reserved for future use.

**Prototype:**

```
BOOL GetAdaptive();
```

**Remarks:**

This is currently not used. Reserved for future use.

**Prototype:**

```
void SetViewTess(TessApprox tess);
```

**Remarks:**

This method is available in release 2.0 and later only.

Sets the tessellation approximation object used for viewport rendering.

**Parameters:**

```
TessApprox tess
```

The tessellation approximation object to be used for viewport rendering.

**Prototype:**

```
TessApprox GetViewTess();
```

**Remarks:**

This method is available in release 2.0 and later only.

Returns the tessellation approximation object used for rendering in the viewports.

**Prototype:**

```
void SetProdTess(TessApprox tess);
```

**Remarks:**

This method is available in release 2.0 and later only.

Sets the tessellation approximation object used for production rendering.

**Parameters:**

**TessApprox tess**

The tessellation approximation object to be used for production rendering.

**Prototype:**

**TessApprox GetProdTess();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the tessellation approximation object used for production rendering.

**Prototype:**

**void SetDispTess(TessApprox tess);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the tessellation approximation object used for display in the viewports.

**Parameters:**

**TessApprox tess**

The tessellation approximation object to be used for the viewports.

**Prototype:**

**TessApprox GetDispTess();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the tessellation approximation object used for display in the viewports.

**Prototype:**

**BOOL GetViewTessNormals();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if normals are used from the viewport tesselator; otherwise FALSE.

**Prototype:**

**void SetViewTessNormals(BOOL use);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets if normals are used from the viewport tesselator.

**Parameters:**

**BOOL use**

TRUE to use normals; FALSE to not use them.

**Prototype:**

**BOOL GetProdTessNormals();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns TRUE if normals are used from the production renderer tesselator; otherwise FALSE.

**Prototype:**

**void SetProdTessNormals(BOOL use);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets if normals are used from the production renderer tesselator.

**Prototype:**

**BOOL GetViewTessWeld();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns TRUE if the viewport mesh is welded after tessellation; otherwise FALSE.

**Prototype:**

**void SetViewTessWeld(BOOL weld);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets if the viewport mesh is welded after tessellation; otherwise FALSE.

**Parameters:**

**BOOL weld**

TRUE to weld; FALSE to not weld.

**Prototype:**

**BOOL GetProdTessWeld();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the production renderer mesh is welded after tessellation; otherwise FALSE.

**Prototype:**

**void SetProdTessWeld(BOOL weld);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets if the production renderer mesh is welded after tessellation; otherwise FALSE.

**Parameters:**

**BOOL weld**

TRUE to weld; FALSE to not weld.

**Dump()**

**Prototype:**

**void Dump();**

**Remarks:**

This method may be called to dump the patch mesh structure via

**DebugPrint()**. See [Debugging](#).

**Prototype:**

**int IntersectRay(Ray& ray, float& at, Point3& norm);**

**Remarks:**

This method is available in release 3.0 and later only.

Calculates the intersection of the specified ray with this patch mesh object.

This method calls the method of the same name on the Mesh cache.

**Parameters:**

**Ray& ray**

Specifies the origin and direction of the ray to intersect with the patch mesh.

See [Class Ray](#).

**float& at**

The computed point of intersection on the surface of the patch mesh.

**Point3& norm**

The face normal at the point of intersection (**at**).

**Return Value:**

Nonzero if the ray intersected the mesh object; otherwise 0. Note that this method ignores backfaces when computing the result.

**Operators:**

**Prototype:**

**PatchMesh& operator=(PatchMesh& fromPatchMesh);**

**Remarks:**

Assignment operator from another **PatchMesh**.

**Prototype:**

**PatchMesh& operator=(Mesh& fromMesh);**

**Remarks:**

Assignment operator. This operator will do the conversion from a **Mesh** to a **PatchMesh**. Note that this can get very slow if there are a lot of faces in the

mesh. When the system does the conversion it must do a fair amount of work with its interior connection lists. It builds a data base of what is connected to what and makes sure that the **PatchMesh** doesn't have any places where an edge is used by more than two patches. Also, for every face in the mesh it generates a triangular patch. Therefore use this method with some caution as it can create some very complex **PatchMesh** objects.

# Class Patch

See Also: [Class PatchMesh](#), [Class TVPatch](#), [Working with Patches](#).

```
class Patch : public BaseInterfaceServer
```

## Description:

A **PatchMesh** is made up of a series of **Patch** objects derived from this class. This is similar to the way faces relate to a mesh. All methods of this class are implemented by the system.

## Data Members:

public:

**int type;**

The patch type. One of the following values:

**PATCH\_UNDEF**

Undefined.

**PATCH\_TRI**

Triangular patch.

**PATCH\_QUAD**

Quadrilateral patch.

**int v[4];**

A patch can have three or four vertices based on the **type**. The corner vertices on a patch are referred to as **a**, **b**, **c**, (and if it's a quad patch **d**). These are ordered **a**, **b**, **c**, **d** going counter-clockwise around the patch.

**int vec[8];**

A patch can have six or eight vector points. The vectors are referred to as follows: **ab** is the vector coming out of vertex **a** towards **b**. **ba** is the one coming out of **b** towards **a**. **bc** is the vector coming out of **b** towards **c**. **cb** is the one coming out of **c** towards **b**, and so on.

**int interior[4];**

A patch can have three or four interior vertices.

**Point3 aux[9];**

This is used internally for triangular patches only (degree 4 control points).

**int edge[4];**

Pointers into the edge list. There can be three or four depending on the patch

**type.**

**DWORD smGroup;**

The smoothing group. This defaults to 1. All patches are smoothed in a **PatchMesh**.

**DWORD flags;**

Patch Flags. The following value may be set.

**PATCH\_AUTO**

Interior vertices are computed automatically if this flag is set (and normally it is set). An example of when this flag would not be set is if you were creating a primitive using patches that needed to make special interior control points to create the shape. In this case you'd clear this flag and then put whatever values you needed into the **vec** array.

**PATCH\_HIDDEN**

The patch is hidden.

**PATCH\_LINEAR Mapping**

This option is available in release 4.0 and later only.

The patch is using the old linear mapping scheme.

**PATCH\_USE\_CRUVED\_MAPPING\_ON\_VERTEX\_COLOR**

This option is available in release 4.0 and later only.

The patch is using the new curved mapping for vertex colors also.

**int aux1;**

This data member is available in release 3.0 and later only.

This is used to track topology changes during editing (Edit Patch).

**int aux2;**

This data member is available in release 3.0 and later only.

This is used to track topology changes during editing (PatchMesh).

**Methods:**

**Prototype:**

**Patch();**

**Remarks:**

Constructor. The type is set to undefined, the smooth group is set to 1, and the flags indicate automatic interior points. Note: This constructor does not

allocate arrays. Use **SetType(type)**.

**Prototype:**

**Patch(Patch& fromPatch);**

**Remarks:**

Constructor. The **this** pointer is set to the **fromPatch**.

**Prototype:**

**~Patch();**

**Remarks:**

Destructor.

**Prototype:**

**void Init();**

**Remarks:**

This method is used by the constructors internally. Developers should not call this method.

**Prototype:**

**void setVerts(int \*vrt);**

**Remarks:**

The vertices are copied from the array passed. Based on the patch type either three or four vertices are copied.

**Parameters:**

**int \*vrt**

The vertices are set to these values. These values are indices into the **v** array.

**Prototype:**

**void setVerts(int a, int b, int c);**

**Remarks:**

Sets the vertices for Tri Patch to those passed.

**Parameters:**

**int a, int b, int c**

The vertices to set. **v[0]=a; v[1]=b; v[2]=c**; These values are indices into the **v** array.

**Prototype:**

**void setVerts(int a, int b, int c, int d)**

**Remarks:**

Sets the vertices for a Quad Patch to those passed.

**Parameters:**

**int a, int b, int c, int d**

The vertices to set. **v[0]=a; v[1]=b; v[2]=c; v[3] = d**; These values are indices into the **v** array.

**Prototype:**

**void setVecs(int ab, int ba, int bc, int cb, int ca, int ac)**

**Remarks:**

Sets the vectors for a Tri Patch to those passed.

**Parameters:**

**int ab, int ba, int bc, int cb, int ca, int ac;**

The vectors to set. **vec[0]=ab; vec[1]=ba; vec[2]=bc;**

**vec[3]=cb; vec[4]=ca; vec[5]=ac**; These values are indices into the **vec** array.

**Prototype:**

**void setVecs(int ab, int ba, int bc, int cb, int cd,  
int dc, int da, int ad)**

**Remarks:**

Sets the vectors for a Quad patch to those passed.

**Parameters:**

**int ab, int ba, int bc, int cb,**

**int cd, int dc, int da, int ad ;**

The vectors to set. **vec[0]=ab; vec[1]=ba; vec[2]=bc; vec[3]=cb; vec[4]=cd; vec[5]=dc; vec[6]=da, vec[7]=ad;** These values are indices into the **vec** array.

**Prototype:**

**void setInteriors(int a, int b, int c)**

**Remarks:**

Sets the interior vertex values for a Tri Patch.

**Parameters:**

The interior values to set: **interior[0]=a; interior[1]=b; interior[2]=c;** These values are indices into the **interior** array.

**Prototype:**

**void setInteriors(int a, int b, int c, int d)**

**Remarks:**

Sets the interior vertex values for a Tri Patch.

**Parameters:**

**int a, int b, int c, int d**

The interior values to set: **interior[0]=a; interior[1]=b; interior[2]=c; interior[3]=d;** These values are indices into the **interior** array.

**Prototype:**

**int getVert(int index);**

**Remarks:**

Returns the vertex specified by the index.

**Parameters:**

**int index**

The vertex to retrieve. This value may be 0, 1, 2 (or 3 if it's a quad patch).

**Prototype:**

**int \*getAllVerts(void);**

**Remarks:**

Returns a pointer to the vertex array.

**Prototype:**

**int WhichEdge(int v1, int v2);**

**Remarks:**

This method is available in release 4.0 and later only.

This method provides an easy way to find out which edge of a patch uses two vertices. Simply supply the two vertices of the desired edge and call this method.

**Parameters:**

**int v1, int v2**

The index of the two vertices.

**Return Value:**

The index of the edge within the patch (0 - 2 for triangular patches, 0 - 3 for quad patches).

CAUTION: Be sure to check the return value before using it for an index – If the two vertices supplied are not used as an edge on this patch, -1 is returned.

**Prototype:**

**int WhichVert(int v);**

**Remarks:**

This method is available in release 4.0 and later only.

This method provides an easy way to find out which corner of a patch uses a given vertex. Simply supply the vertex index and call this method.

**Parameters:**

**int v**

The index of the vertex.

**Return Value:**

The corner index of the given vertex within the patch (0 - 2 for triangular patches, 0 - 3 for quad patches).

CAUTION: Be sure to check the return value before using it for an index – If the vertex supplied is not used as a corner on this patch, -1 is returned.

**Prototype:**

**Point3 interp(PatchMesh \*pMesh, float u, float v);**

**Remarks:**

Quadrilateral patch interpolator. This method returns a point on the surface of the patch based on the specified u and v values.

**Parameters:**

**PatchMesh \*pMesh**

Points to the **PatchMesh** to interpolate.

**float u**

The u value in the range 0.0 to 1.0. This defines the distance along one axis of the patch.

**float v**

The v value in the range 0.0 to 1.0. This defines the distance along the other axis of the patch.

**Return Value:**

A point on the surface of the patch.

**Prototype:**

**Point3 interp(PatchMesh \*pMesh, float u, float v, float w);**

**Remarks:**

Triangle patch interpolator. This method returns a point on the surface of the patch based on the specified u, v and w values. The u, v, w values are barycentric coordinates.  $u+v+w = 1.0$ . If u is 1, and v and w are 0, the point is at the first vertex. If u is 0, v is 1, and w is 0, then the point is at the second vertex. If u and v are 0 and w is 1 then the point is at the third vertex. Varying positions between these values represent different positions on the patch.

**Parameters:**

**PatchMesh \*pMesh**

Points to the **PatchMesh** to interpolate.

**float u, float v, float w**

The barycentric coordinates.

**Return Value:**

A point on the surface of the patch.

**Prototype:**

**void ComputeAux(PatchMesh \*pMesh, int index);**

**Remarks:**

This method is used internally. It compute the degree-4 alias control points.

**Prototype:**

**void computeInteriors(PatchMesh\* pMesh);**

**Remarks:**

Whenever you are done working on a **PatchMesh**, this method should be called. If the interior vertices of the patch are automatic it will update them to correctly match the changes to the other vectors. This computes interior vertices considering this patch only.

**Parameters:**

**PatchMesh\* pMesh**

Points to the **PatchMesh** to compute the interior vertices of.

**Prototype:**

**void SetType(int type, BOOL init = FALSE);**

**Remarks:**

Sets the type of the patch to either Tri or Quad and optionally resets the arrays.

**Parameters:**

**int type**

The patch type. One of the following values:

**PATCH\_TRI** - Triangular Patch

**PATCH\_QUAD** - Quadrilateral Patch

**BOOL init = FALSE**

If TRUE the arrays are reset to undefined; otherwise they are left unchanged. Normally this is set to FALSE.

**Prototype:**

**void SetAuto(BOOL sw = TRUE);**

**Remarks:**

Sets the flag controlling if interior vertices are computed automatically.

**Parameters:**

**BOOL sw = TRUE**

TRUE to set; FALSE to clear.

**Prototype:**

**BOOL IsAuto();**

**Remarks:**

Returns TRUE if the **PATCH\_AUTO** flag is set; otherwise FALSE.

**Prototype:**

**void SetHidden(BOOL sw = TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the hidden state of the patch.

**Parameters:**

**BOOL sw = TRUE**

TRUE to hide; FALSE to unhide.

**Prototype:**

**BOOL IsHidden();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the patch is hidden; otherwise FALSE.

**Prototype:**

**void Dump();**

**Remarks:**

This may be called to dump the patch mesh structure via **DebugPrint()**. See [Debugging](#).

**Operators:**

**Prototype:**

**Patch& operator=(Patch& from);**

**Remarks:**

Assignment operator.

**Parameters:**

**Patch& from**

The patch to copy from.

# Class PatchVec

See Also: [Class PatchMesh](#), [Working with Patches](#).

class PatchVec

## Description:

This class represents a patch vector. This can be either an interior vector or an edge vector. All methods of this class are implemented by the system.

## Data Members:

public:

### **Point3 p;**

The vertex location.

### **int vert;**

The vertex which owns this vector.

### **IntTab patches;**

These are the patches that share this vector. If the edge is open there will only be one patch. In this case **patches[0]** will be used and **patches[1]** will be -1. Note: As of R4.0 the previous array of [2] has been replaced with the IntTab because vectors can now be used by more than two patches.

### **DWORD flags;**

The patch vector flag:

#### **PVEC\_INTERIOR**

This indicates the vector is an interior vector. These are the three vectors inside a tri patch or the four inside a quad patch.

### **int aux1;**

This data member is available in release 3.0 and later only.

Used to track topology changes during editing (Edit Patch).

### **int aux2;**

This data member is available in release 3.0 and later only.

Used to track topology changes during editing (PatchMesh).

## Methods:

## Prototype:

**PatchVec();**

**Remarks:**

Constructor. The location is set to 0,0,0. The vertex owner is set to undefined. The patches using the vector is set to undefined. The flags are set to 0.

**Prototype:**

**PatchVec(PatchVec &from);**

**Remarks:**

Constructor. The data members are initialized to those of the **from** patch vector.

**Parameters:**

**PatchVec &from**

The vector to copy from.

**Prototype:**

**void ResetData();**

**Remarks:**

Resets the data members. The vertex owner is set to undefined. The patches using the vector is set to undefined.

**Prototype:**

**BOOL AddPatch(int index);**

**Remarks:**

Adds the specified patch to this vector table.

**Parameters:**

**int index**

The index in the **PatchMesh** class patches table (**patches**) of the patch to add.

**Return Value:**

Returns TRUE if the patch was added; otherwise FALSE.

**Prototype:**

**void Transform(Matrix3 &tm);**

**Remarks:**

This method is available in release 3.0 and later only.  
Transforms the vertex location p by the specified matrix.

**Parameters:****Matrix3 &tm**

The matrix to transform the vertex.

**Prototype:**

**IOResult Save(ISave\* isave);**

**Remarks:**

This is used internally to save the data to the .MAX file.

**Prototype:**

**IOResult Load(ILoad\* iload);**

**Remarks:**

This is used internally to load the data from the .MAX file.

**Operators:****Prototype:**

**PatchVec& operator=(PatchVec& from);**

**Remarks:**

Assignment operator.

**Parameters:****PatchVec& from**

The patch vector to copy from.

## Class PatchVert

See Also: [Class PatchMesh](#), [Template Class Tab](#), [Working with Patches](#).

### Description:

This class stores the information associated with a patch vertex and provides methods to access the data associated with this vertex. All methods of this class are implemented by the system.

### Data Members:

public:

#### **Point3 p;**

The vertex location.

#### **IntTab vectors;**

The list of vectors attached to this vertex. There can be any number of vectors attached to a vertex. For example consider the north pole of a sphere made from a set of triangular patches. If there were 16 patches meeting at this point there would be 16 vectors. The table contains the indices of these vectors. This is set up automatically when a developer calls **buildLinkages()**.

#### Note: **typedef Tab<int> IntTab;**

Note that the methods below allow a developer to manipulate the tables of this class. Developers must be careful when doing so as it is easy to corrupt the patch data structures. It may be easier for developers to manipulate the patches, delete vertices, etc., by manipulating them in the **PatchMesh** and then call **buildLinkages()** again. The methods below do work however and may be used.

#### **IntTab patches;**

The list of patches using this vertex.

#### **IntTab edges;**

This data member is available in release 4.0 and later only.

The list of edges using this vertex. This table will be set up automatically when a developer calls **buildLinkages()**.

#### **DWORD flags;**

The patch vertex flags

**PVERT\_COPLANAR**

This constrains things such that this vertex and all the vectors that are attached to it are coplanar. If this is set, and you call **ApplyConstraints()**, the system will adjust the vectors to the average plane that they are in and then constrain them to it.

**PVERT\_CORNER**

The vertex is a corner.

**PVERT\_HIDDEN**

The vertex is hidden.

**int aux1;**

Used to track topo changes during editing (Edit Patch).

**int aux2;**

Used to track topology changes during editing (PatchMesh).

**Methods:**

**Prototype:**

**PatchVert();**

**Remarks:**

Constructor. The location is set to 0,0,0. The flags are set to 0.

**Prototype:**

**PatchVert(PatchVert &from);**

**Remarks:**

Constructor. The data members are copied from the from **PatchVert**.

**Parameters:**

**PatchVert &from**

The source PatchVert.

**Prototype:**

**~PatchVert();**

**Remarks:**

Destructor. Deletes the elements from the **vectors** table and **patches** table.

**Prototype:**

**void ResetData();**

**Remarks:**

This method deletes the elements from the **vectors** table and **patches** table.

**Prototype:**

**int FindVector(int index);**

**Remarks:**

Returns the index in this classes **vectors** table of the vector whose index is passed. If not found, -1 is returned.

**Parameters:**

**int index**

The index in the **PatchMesh** class vectors table (**vecs**) of the vector to find.

**Prototype:**

**void AddVector(int index);**

**Remarks:**

Adds the specified vector to this vector table.

**Parameters:**

**int index**

The index in the **PatchMesh** class vectors table (**vecs**) of the vector to add.

**Prototype:**

**void DeleteVector(int index);**

**Remarks:**

Deletes the specified vector from this vector table.

**Parameters:**

**int index**

The index in the **PatchMesh** class vectors table (**vecs**) of the vector to delete.

**Prototype:**

**int FindPatch(int index);**

**Remarks:**

Returns the index in this classes **patches** table of the patch whose index is passed. If not found, -1 is returned.

**Parameters:**

**int index**

The index in the **PatchMesh** class patches table (**patches**) of the patch to find.

**Prototype:**

**void AddPatch(int index);**

**Remarks:**

Adds the specified patch to this vector table.

**Parameters:**

**int index**

The index in the **PatchMesh** class patches table (**patches**) of the patch to add.

**Prototype:**

**void DeletePatch(int index);**

**Remarks:**

Deletes the patch specified by the index.

**Parameters:**

**int index**

The index in the **PatchMesh** class patches table (**patches**) of the patch to delete.

**Prototype:**

**int FindEdge(int index);**

**Remarks:**

Returns the index in this classes **edges** table of the patch whose index is

passed. If not found, -1 is returned.

**Parameters:**

**int index**

The index in the **PatchMesh** class edges table (**edges**) of the edge to find.

**Prototype:**

**void AddEdge(int index);**

**Remarks:**

Adds the specified edge to this vector table.

**Parameters:**

**int index**

The index in the **PatchMesh** class edges table (**edges**) of the edge to add.

**Prototype:**

**void DeleteEdge(int index);**

**Remarks:**

Deletes the edge specified by the index.

**Parameters:**

**int index**

The index in the **PatchMesh** class edges table (**edges**) of the edge to delete.

**Prototype:**

**void Transform(Matrix3 &tm);**

**Remarks:**

This method is available in release 3.0 and later only.

Transform the vertex by the specified matrix.

**Parameters:**

**Matrix3 &tm**

The matrix which transforms the point.

**Prototype:**

**void setHidden(BOOL sw = TRUE);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the hidden state of the vertex.

**Parameters:**

**BOOL sw = TRUE**

TRUE to set to hidden; FALSE for visible.

**Prototype:**

**BOOL IsHidden();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns TRUE if the vertex is hidden; otherwise FALSE.

**Prototype:**

**IOResult Save(ISave\* isave);**

**Remarks:**

This method is used internally to save the class data to disk storage.

**Prototype:**

**IOResult Load(ILoad\* iload);**

**Remarks:**

This method is used internally to load the class data from disk storage.

**Operators:**

**Prototype:**

**PatchVert& operator=(PatchVert& from);**

**Remarks:**

Assignment operator.

**Parameters:**

**PatchVert& from**

The patch vertex to copy from.

## Class PatchEdge

See Also: [Class PatchMesh](#), [Working with Patches](#).

class PatchEdge

### Description:

This class describes a patch edge using the vertices at the edge ends, and the indices of the patches sharing the edge. All methods of this class are implemented by the system.

### Data Members:

public:

**int v1;**

Index of the first vertex.

**int vec12;**

Vector from **v1** to **v2**.

**int vec21;**

Vector from **v2** to **v1**.

**int v2;**

Index of second vertex.

**IntTab patches;**

Index of the patches using this edge. If the edge is only used by one patch, patches[1] will be less than zero. Note: Previous to R4.0 two separate integer variables (**patch1** and **patch2**) were used.

**int aux1;**

This is used to track topology changes during editing (Edit Patch).

**int aux2;**

This is used to track topology changes during editing (PatchMesh).

### Methods:

### Prototype:

**PatchEdge();**

### Remarks:

Constructor. The data members are initialized as undefined.

**Prototype:**

**PatchEdge(PatchEdge &from);**

**Remarks:**

Constructor. The data members are initialized from the **PatchEdge** passed.

**Prototype:**

**PatchEdge(int v1, int vec12, int vec21, int v2, int p1, int p2, int  
aux1=-1, int aux2=-1);**

**Remarks:**

Constructor. The data members are initialized to the values passed.

**Prototype:**

**IOResult Save(ISave\* isave);**

**Remarks:**

This is used internally to save the data to the .MAX file.

**Prototype:**

**IOResult Load(ILoad\* iload);**

**Remarks:**

This is used internally to load the data from the .MAX file.

**Prototype:**

**void Dump();**

**Remarks:**

You may call this method to dump the patch edge structure via **DebugPrint()**. See [Debugging](#).

## Class TVPatch

See Also: [Class Patch](#), [Class PatchMesh](#), [Working with Patches](#).

class TVPatch

### Description:

This is a texture vertex patch structure. This is similar to the **TVFace** class used with a **Mesh**. All methods of this class are implemented by the system.

### Data Members:

public:

**int tv[4];**

Texture vertices. There are always four here, even for Tri Patches. These are indices in the PatchMesh's **tVerts** array.

**int handles[8];**

The UVW vertices for the handles.

**int interiors[4];**

The UVW interior handles.

### Methods:

#### Prototype:

**TVPatch();**

#### Remarks:

Constructor.

#### Prototype:

**TVPatch(TVPatch& fromPatch);**

#### Remarks:

Constructor. The **this** pointer is set to the **fromPatch**.

#### Prototype:

**void Init();**

#### Remarks:

Performs initialization by setting the texture vertices to 0.

**Prototype:**

```
void setTVerts(int *vrt, int count);
```

**Remarks:**

Sets the specified number of texture vertices.

**Parameters:**

**int \*vrt**

The array of verts to set.

**int count**

The number to set.

**Prototype:**

```
void setTVerts(int a, int b, int c, int d = 0);
```

**Remarks:**

Sets the texture vertices for a Quad Patch.

**Parameters:**

**int a, int b, int c, int d = 0**

The vertices to set: **tv[0]=a; tv[1]=b; tv[2]=c; tv[3]=d;**

**Prototype:**

```
int getTVert(int index);
```

**Remarks:**

Returns the texture vertex specified by the index.

**Parameters:**

**int index**

The index of the texture vertex to return.

**Prototype:**

```
int *getAllTVerts();
```

**Remarks:**

Returns a pointer to the array of texture vertices.

**Prototype:**

```
void setTHandles(int *vrt, int count);
```

**Remarks:**

Sets the specified number of texture (UVW) handles.

**Parameters:**

**int \*vrt**

The array of handles to set.

**int count**

The number to set.

**Prototype:**

```
void setTHandles(int a, int b, int c, int d, int e, int f, int g = 0, int h = 0);
```

**Remarks:**

Sets the texture (UVW) handles for a Quad Patch.

**Parameters:**

**int a, int b, int c, int d, int e, int f, int g = 0, int h = 0**

The handles to set: **handles[0]=a; handles[1]=b; handles[2]=c;**

**handles[3]=d; handles[4]=e; handles[5]=f; handles[6]=g;**

**handles[7]=h;**

**Prototype:**

```
void setTInteriors(int *vrt, int count);
```

**Remarks:**

Sets the specified number of interior texture (UVW) handles.

**Parameters:**

**int \*vrt**

The array of interior handles to set.

**int count**

The number to set.

**Prototype:**

**void setTInteriors(int a, int b, int c, int d = 0);**

**Remarks:**

Sets the interior texture (UVW) handles for a Quad Patch.

**Parameters:**

**int a, int b, int c, int d = 0**

The interior handles to set: **interiors[0]=a; interiors[1]=b;  
interiors[2]=c; interiors[3]=d;**

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

This method is used internally in saving to the MAX file.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

This method is used internally in loading from the MAX file.

**Operators:**

**Prototype:**

**TVPatch& operator=(const TVPatch& from);**

**Remarks:**

Assignment operator.

**Parameters:**

**TVPatch& from**

The texture vertex patch to copy from.

## Class Manipulator

See Also: [Class HelperObject](#), [Class SimpleManipulator](#), [Class IManipulatorMgr](#), [Class INode](#), [Class ViewExp](#), [Class IPoint2](#), [Class Box3](#)

```
class Manipulator : public HelperObject
```

### Description:

This class is available in release 4.0 and later only.

This is the base class for the development of Manipulator plug-ins. Shown are the methods that need to be implemented by the plug-in. Most of these methods are implemented by the **SimpleManipulator** class so most developers will not need to use these unless **SimpleManipulator** is not sufficient.

### Data Members:

protected:

**INode\* mpINode;**

The node being manipulated

### Methods:

public:

### Prototype:

**Manipulator(INode\* pINode);**

### Remarks:

Constructor. The node data member is initialized to the node passed.

### Prototype:

**virtual int HitTest(TimeValue t, INode\* pNode, int type, int crossing, int flags, IPoint2 \*pScreenPoint, ViewExp \*pVpt) = 0;**

### Remarks:

This method is called to determine if the specified screen point intersects the manipulator. The method returns nonzero if the item was hit; otherwise 0. This is like the normal **HitTest()** method in the **BaseObject** class. The difference is that it needs to log it hits in the viewport SubObjectHitList. It does this using the **ManipHitData** class defined in **Manipulator.h**.

**Parameters:****TimeValue t**

The time to perform the hit test.

**INode\* pNode**

A pointer to the node to test.

**int type**

The type of hit testing to perform. See [Hit Test Types](#) for details.

**int crossing**

The state of the crossing setting. If TRUE crossing selection is on.

**int flags**

The hit test flags. See [Hit Test Flags](#) for details.

**IPoint2 \*pScreenPoint**

Points to the screen point to test.

**ViewExp \*pVpt**

An interface that may be used to call methods associated with the viewports.

**Return Value:**

Nonzero if the item was hit; otherwise 0.

**Prototype:**

```
virtual int Display(TimeValue t, INode* pNode, ViewExp *pVpt,  
int flags) = 0;
```

**Remarks:**

This method lines the **BaseObject::Display()** method and displays the manipulator object.

**Parameters:****TimeValue t**

The time to display the object.

**INode\* pNode**

Points to the node that is being manipulated by the manipulator.

**ViewExp \*pVpt**

An interface that may be used to call methods associated with the viewports.

**int flags**

See [List of Display Flags](#).

**Return Value:**

The return value is not currently used.

**Prototype:**

```
virtual void GetLocalBoundingBox(TimeValue t, INode* inode,  
ViewExp* vp, Box3& box) = 0;
```

**Remarks:**

Used Internally.

Returns the object space bounding box of the manipulator in the object's local coordinates.

**Parameters:****TimeValue t**

The time to retrieve the bounding box.

**INode\* inode**

The node that is being manipulated by the manipulator.

**ViewExp\* vp**

An interface that may be used to call methods associated with the viewports.

**Box3& box**

The bounding box is returned here.

**Prototype:**

```
virtual bool AlwaysActive();
```

**Remarks:**

This method can be used to tell the manipulator management system that this manipulator is always active.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual TSTR& GetManipName() = 0;
```

**Remarks:**

This method returns the manipulator name string. The **SimpleManipulator**

class uses this method for the tooltip in the viewport.

**Prototype:**

**virtual DisplayState MouseEntersObject(TimeValue t, ViewExp\* pVpt, IPoint2& m, ManipHitData\* pHitData);**

**Remarks:**

This method gets called when the mouse first passes over a manipulator object. The return value is used to determine whether a redraw is needed or not. Normally manipulators display in a different color when the mouse is over them, so this should return **kFullRedrawNeeded**.

**Parameters:**

**TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Return Value:**

The display state indicating whether no redraw, a full redraw, or a post redraw is needed.

**Default Implementation:**

**{return kNoRedrawNeeded; }**

**Prototype:**

**virtual DisplayState MouseLeavesObject(TimeValue t, ViewExp\* pVpt, IPoint2& m, ManipHitData\* pHitData);**

**Remarks:**

This method gets called when the mouse leaves the manipulator object. The return value is used to determine whether a redraw is needed or not. Normally manipulators display in a different color when the mouse is over them, so this should return **kFullRedrawNeeded**.

**Parameters:****TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Return Value:**

The display state indicating whether no redraw, a full redraw, or a post redraw is needed.

**Default Implementation:**

```
{return kNoRedrawNeeded; }
```

**Prototype:**

```
virtual void OnMouseMove(TimeValue t, ViewExp* pVpt,  
IPoint2& m, DWORD flags, ManipHitData* pHitData);
```

**Remarks:**

This method gets called when the mouse is pressed down and moves within the manipulator context. It is the method that does the actual manipulator. It is up to the manipulator code to turn the mouse position into a new value for the parameter(s) being manipulated.

**Parameters:****TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**DWORD flags**

Not used, should be set to 0.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void OnButtonDown(TimeValue t, ViewExp* pVpt,  
IPoint2& m, DWORD flags, ManipHitData* pHitData);
```

**Remarks:**

This method gets called when the mouse buttons is first pressed down within the manipulator context.

**Parameters:**

**TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**DWORD flags**

Not used, should be set to 0.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void OnButtonUp(TimeValue t, ViewExp* pVpt, IPoint2&  
m, DWORD flags, ManipHitData* pHitData);
```

**Remarks:**

This method gets called when the mouse buttons is released within the manipulator context, and thus signals the end of the manipulation.

**Parameters:**

**TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**DWORD flags**

Not used, should be set to 0.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual INode* GetINode();
```

**Remarks:**

This method returns a pointer to the INode that is currently being manipulated.

**Default Implementation:**

```
{ return mpINode; }
```

**Prototype:**

```
virtual void DeleteThis();
```

**Remarks:**

This method deletes the manipulator instance.

**Default Implementation:**

```
{ delete this; }
```

## Class SimpleManipulator

See Also: [Class Manipulator](#), [Class ISimpleManipulator](#), [Class PolyShape](#), [Class GizmoShape](#), [Class ReferenceTarget](#), [Class IParamBlock2](#), [Class INode](#), [Class Mesh](#), [Class Point3](#), [Class Point2](#), [Class ViewExp](#), [Class IColorManager](#), [Class Ray](#), [Class ManipHitData](#), [List of Marker Types](#)

```
class SimpleManipulator: public Manipulator, public ISimpleManipulator
```

### Description:

This class is available in release 4.0 and later only.

The SimpleManipulator class provides a framework for implementing many common manipulators. It provides the following services:

- It supports an arbitrary number of gizmos made from PolyShape and/or Mesh objects.
- It creates and maintains tool tips in the viewport.
- It does hit testing, display and bounding box computations of the gizmos.
- It maintains an IParamBlock2 for the parameters of the Manipulator.

This class maintains a pointer to a parameter block. If the client of **SimpleManipulator** uses a single parameter block then **SimpleManipulator** can manage all the methods associated with SubAnims and References for the client.

If the client of **SimpleManipulator** maintains several parameter blocks then the client must implement the methods **NumSubs()**, **SubAnim(i)**, **SubAnimName(i)**, **NumRefs()**, **GetReference(i)** and **SetReference(i)** and call the **SimpleManipulator** methods when 'i' refers to the parameters maintained by **SimpleManipulator**.

Samples of Manipulators can be found in the SDK,  
**\MAXSDK\SAMPLES\MANIPULATORS.**

The Function Publishing interface to SimpleManipulators is defined as:  
**#define SIMPLE\_MANIP\_INTERFACE Interface\_ID(0x617c41d4, 0x6af06a5f)**

The following functions are not part of this class but are available for use with it

in the making of gizmo objects:

**Function:**

**Mesh\* MakeSphere(Point3& pos, float radius, int segments);**

**Remarks:**

This global function is available in release 4.0 and later only.

Creates a new mesh sphere at the given center position with the specified radius and segments count.

**Parameters:**

**Point3& pos**

The center point for the sphere in object space.

**float radius**

The radius for the sphere.

**int segments**

The number of segments for the sphere mesh.

**Return Value:**

A pointer to the sphere mesh. Developers are responsible for deleting this mesh when done.

**Function:**

**Mesh\* MakeTorus(Point3& pos, float radius, float radius2, int segs, int sides);**

**Remarks:**

This global function is available in release 4.0 and later only.

Create a mesh torus with the given center point, outer radius, inner radius, and segment counts along the two circular dimensions of the torus.

**Parameters:**

**Point3& pos**

The center point in object space.

**float radius**

The first radius.

**float radius2**

The second radius.

**int segs**

The segment count along the vertical circular dimension.

**int sides**

The segment count along the horizontal circular dimension.

**Return Value:**

A pointer to the torus mesh created. Developers are responsible for deleting this mesh when done with it.

**Function:**

**Mesh\* MakeBox(Point3& pos, float l, float w, float h, int lsegs, int wsegs, int hsegs);**

**Remarks:**

This global function is available in release 4.0 and later only.

Creates a mesh box with the given center point, length, width and height as well as segment parameters.

**Parameters:****Point3& pos**

The box is built from this position in size along +X, +Y and +Z. This coordinate is in object space.

**float l**

The length of the box.

**float w**

The width of the box.

**float h**

The height of the box.

**int lsegs**

The number of segments in the length dimension.

**int wsegs**

The number of segments in the width dimension.

**int hsegs**

The number of segments in the height dimension.

**Return Value:**

A pointer to the box mesh. Developers are responsible for deleting this mesh

when done.

**Function:**

**void AddCubeShape(PolyShape& shape, Point3& pos, float size);**

**Remarks:**

This global function is available in release 4.0 and later only.

Creates a new series of lines in the form of a cube and adds it to the specified PolyShape with the given position and side length size.

**Parameters:**

**PolyShape& shape**

The PolyShape to add the box shape to.

**Point3& pos**

The position for the center of the cube shape in object space.

**float size**

The size of one side of the cube in object space.

**Data Members:**

protected:

**int mDispSelectedIndex;**

The Index of manipulator that the mouse is over, for display.

**TSTR mToolTip;**

The tooltip text.

**float mGizmoScale;**

The scaling factor of the gizmo.

**IParamBlock2 \*mpPblock;**

The Parameter Block 2 for the manipulator.

**RefTargetHandle mhTarget;**

The handle to the manipulator reference target.

**MouseState mState;**

The state of the mouse, which is one of the following:

**kMouseIdle**

The mouse is idle, manipulator not active and the mouse is not over it.

**kMouseDragging**

The mouse is currently dragging the manipulator.

**kMouseOverManip**

The mouse is over the manipulator, but it is not being dragged.

**BOOL mActiveViewOnly;**

This flag defines whether the manipulator is shown in the active viewport only.

**Interval mValid;**

The validity interval of the reference.

**Methods:**

public:

**Prototype:**

**SimpleManipulator();**

**Remarks:**

Constructor.

**Prototype:**

**SimpleManipulator(INode\* pNode);**

**Remarks:**

Constructor.

**Prototype:**

**~SimpleManipulator();**

**Remarks:**

Destructor.

**Prototype:**

**void ClearPolyShapes();**

**Remarks:**

Implemented by the system.

Removes all of the current gizmos in the manipulator. This is normally called at the top of **UpdateShapes()** to clear out any previous gizmos before

creating new ones.

**Prototype:**

```
void AppendPolyShape(PolyShape* pPolyShape, DWORD flags = 0, Point3& unselColor = GetUIColor(COLOR_SEL_GIZMOS), Point3& selColor = GetSubSelColor());
```

**Remarks:**

Implemented by the system.

This method adds a new PolyShape gizmo to the manipulator. The shape is defined in the local coordinates of the node that owns the manipulator.

**Parameters:**

**PolyShape\* pPolyShape**

A pointer to the poly shape to add.

**DWORD flags = 0**

The flags can have one or more of the following values:

**kGizmoDontDisplay;**

Tells the gizmo not to display. It will still hit test it.

**kGizmoDontHitTest;**

Tells the gizmo not to hit test. It will still display.

**kGizmoScaleToViewport;**

Tells the gizmo to scale itself to have a constant size in the viewport. In this case, the system uses the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case. This flag only applies to mesh gizmo currently.

**Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

**Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

**Prototype:**

```
void AppendGizmo(GizmoShape* pGizmoShape, DWORD flags = 0, Point3& unselColor = GetUIColor(COLOR_SEL_GIZMOS), Point3& selColor = GetSubSelColor());
```

**Remarks:**

Implemented by the system.

This method adds a new GizmoShape to the manipulator. The shape is defined in the local coordinates of the node that owns the manipulator.

**Parameters:**

**GizmoShape\* pGizmoShape**

A pointer to the gizmo shape to add.

**DWORD flags = 0**

The flags can have one or more of the following values:

**kGizmoDontDisplay;**

Tells the gizmo not to display. It will still hit test it.

**kGizmoDontHitTest;**

Tells the gizmo not to hit test. It will still display.

**kGizmoScaleToViewport;**

Tells the gizmo to scale itself to have a constant size in the viewport. In this case, the system uses the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case. This flag only applies to mesh gizmo currently.

**Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

**Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

**Prototype:**

```
void AppendMesh(Mesh* pMesh, DWORD flags = 0, Point3&  
unselColor = GetUIColor(COLOR_SEL_GIZMOS), Point3&  
selColor = GetSubSelColor());
```

**Remarks:**

Implemented by the system.

This method adds a new Mesh to the manipulator. The shape is defined in the local coordinates of the node that owns the manipulator.

**Parameters:**

**Mesh\* pMesh**

A pointer to the mesh to add.

**DWORD flags = 0**

The flags can have one or more of the following values:

**kGizmoDontDisplay;**

Tells the gizmo not to display. It will still hit test it.

**kGizmoDontHitTest;**

Tells the gizmo not to hit test. It will still display.

**kGizmoScaleToViewport;**

Tells the gizmo to scale itself to have a constant size in the viewport. In this case, the system uses the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case. This flag only applies to mesh gizmo currently.

**Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

**Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

**Prototype:**

```
void AppendMarker(MarkerType markerType, Point3& position,  
DWORD flags = 0, Point3& unselColor =  
GetUIColor(COLOR_SEL_GIZMOS), Point3& selColor =  
GetSubSelColor());
```

**Remarks:**

Implemented by the system.

This method adds a new Marker to the manipulator. The shape is defined in the local coordinates of the node that owns the manipulator.

**Parameters:**

**MarkerType markerType**

The marker type for marker gizmos. See the [List of Marker Types](#)

**Point3& position**

The position of the marker.

**DWORD flags = 0**

The flags can have one or more of the following values:

**kGizmoDontDisplay;**

Tells the gizmo not to display. It will still hit test it.

**kGizmoDontHitTest;**

Tells the gizmo not to hit test. It will still display.

**kGizmoScaleToViewport;**

Tells the gizmo to scale itself to have a constant size in the viewport. In this case, the system uses the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case. This flag only applies to mesh gizmo currently.

**Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

**Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

**Prototype:**

```
void AppendText(TCHAR* pText, Point3& position, DWORD  
flags = 0, Point3& unselColor =  
GetUIColor(COLOR_SEL_GIZMOS), Point3& selColor =  
GetSubSelColor());
```

**Remarks:**

Implemented by the system.

This method adds a new Text to the manipulator. The shape is defined in the local coordinates of the node that owns the manipulator.

**Parameters:****TCHAR\* pText**

The text string to add.

**Point3& position**

The position of the text.

**DWORD flags = 0**

The flags can have one or more of the following values:

**kGizmoDontDisplay;**

Tells the gizmo not to display. It will still hit test it.

**kGizmoDontHitTest;**

Tells the gizmo not to hit test. It will still display.

**kGizmoScaleToViewport;**

Tells the gizmo to scale itself to have a constant size in the viewport. In this case, the system uses the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case. This flag only applies to mesh gizmo currently.

**Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

**Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

**Prototype:**

**TSTR& GetManipName();**

**Remarks:**

Implemented by the system.

This method returns the manipulator name.

**Prototype:**

**void SetGizmoScale(float gizmoScale);**

**Remarks:**

Implemented by the system.

This method allows you to set the scale of the gizmo.

**Parameters:**

**float gizmoScale**

The scale factor.

**Prototype:**

**TSTR& GetToolTip();**

**Remarks:**

Implemented by the system.

This method returns the tooltip string. Used internally.

**Default Implementation:**

```
{ return mToolTip; }
```

**Prototype:**

```
void SetToolTipWnd(HWND hWnd);
```

**Remarks:**

Implemented by the system.

Used internally.

**Prototype:**

```
void SetToolTipTimer(UINT timer);
```

**Remarks:**

Implemented by the system.

Used internally.

**Prototype:**

```
UINT GetToolTipTimer();
```

**Remarks:**

Implemented by the system.

Used internally.

**Prototype:**

```
HWND GetToolTipWnd();
```

**Remarks:**

Implemented by the system.

Used internally.

**Prototype:**

```
IParamBlock2* GetPBlock();
```

**Remarks:**

Implemented by the system.

This method returns a pointer to the parameter block.

**Default Implementation:**

```
{ return mpPblock; }
```

These must be implemented in the sub-class of SimpleManipulator

**Prototype:**

```
virtual void UpdateShapes(TimeValue t, TSTR& toolTip) = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method gets called whenever the manipulator needs to update its gizmos.

This is implemented by the manipulator to create the gizmos based on the current state of the node being manipulated.

**Parameters:**

**TimeValue t**

The time at which to update the shape.

**TSTR& toolTip**

The tool tip text to update.

**Prototype:**

```
virtual void ManipulatorSelected();
```

**Remarks:**

Implemented by the system.

Used internally.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
void SetManipTarget(RefTargetHandle hTarg);
```

**Remarks:**

Implemented by the system.

This tells the SimpleManipulator to make a reference to hTarg. This is normally called in the constructor of a manipulator to set a reference to whatever is being manipulated.

**Parameters:**

**RefTargetHandle hTarg**

The handle to the reference target.

**Prototype:**

**RefTargetHandle GetManipTarget();**

**Remarks:**

This method returns the handle to the manipulator reference target.

**Default Implementation:**

```
{ return mhTarget; }
```

**Prototype:**

**void SetMouseState(MouseState state);**

**Remarks:**

This method sets the state of the mouse.

**Parameters:**

**MouseState state**

One of the following values:

**kMouseIdle**

The mouse is idle, manipulator not active and the mouse is not over it.

**kMouseDragging**

The mouse is currently dragging the manipulator.

**kMouseOverManip**

The mouse is over the manipulator, but it is not being dragged.

**Default Implementation:**

```
{ mState = state; }
```

**Prototype:**

## **MouseState GetMouseState();**

### **Remarks:**

Implemented by the system.

This method returns the state of the mouse, which is one of the following values: **kMouseIdle**, **kMouseDragging**, or **kMouseOverManip**.

### **Default Implementation:**

```
{ return mState; }
```

### **Prototype:**

```
void OnButtonDown(TimeValue t, ViewExp* pVpt, IPoint2& m,  
DWORD flags, ManipHitData* pHitData);
```

### **Remarks:**

Implemented by the system.

This method gets called when the mouse buttons is pressed within the manipulator context. Used internally.

### **Parameters:**

#### **TimeValue t**

The time to display the object.

#### **ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

#### **IPoint2& m**

The location of the tooltip.

#### **DWORD flags**

Not used, should be set to 0.

#### **ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

### **Prototype:**

```
void OnMouseMove(TimeValue t, ViewExp* pVpt, IPoint2& m,  
DWORD flags, ManipHitData* pHitData);
```

### **Remarks:**

This method gets called when the mouse is pressed down and moves within

the manipulator context. It is the method that does the actual manipulator. It is up to the manipulator code to turn the mouse position into a new value for the parameter(s) being manipulated. It also updates the tooltip with the current value of the parameter.

**Parameters:**

**TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**DWORD flags**

Not used, should be set to 0.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Prototype:**

**void OnButtonUp(TimeValue t, ViewExp\* pVpt, IPoint2& m, DWORD flags, ManipHitData\* pHitData);**

**Remarks:**

Implemented by the system.

This method gets called when the mouse buttons is released within the manipulator context. Used internally.

**Parameters:**

**TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**DWORD flags**

Not used, should be set to 0.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Prototype:**

**DisplayState MouseEntersObject(TimeValue t, ViewExp\* pVpt, IPoint2& m, ManipHitData\* pHitData);**

**Remarks:**

Implemented by the system.

This method gets called when the mouse enters the manipulator object. Used internally.

**Parameters:**

**TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Return Value:**

The display state indicating whether no redraw, a full redraw, or a post redraw is needed.

**Prototype:**

**DisplayState MouseLeavesObject(TimeValue t, ViewExp\* pVpt, IPoint2& m, ManipHitData\* pHitData);**

**Remarks:**

Implemented by the system.

This method gets called when the mouse leaves the manipulator object. Used internally.

**Parameters:**

**TimeValue t**

The time to display the object.

**ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The location of the tooltip.

**ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

**Return Value:**

The display state indicating whether no redraw, a full redraw, or a post redraw is needed.

**Prototype:**

```
IPoint2& GetTipPos();
```

**Remarks:**

Implemented by the system.

This method returns the position of the tooltip. Used internally.

**Default Implementation:**

```
{ return mToolTipPos; }
```

**Prototype:**

```
void GetLocalViewRay(ViewExp* pVpt, IPoint2& m, Ray&  
viewRay);
```

**Remarks:**

This method is normally called from a manipulator's OnMouseMove method. It computes a ray that passes through the given mouse point in the given viewport. The result is in the local coordinates of the node owning the manipulator.

**Parameters:****ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The screen coordinate.

**Ray& viewRay**

The returned local view ray.

**Prototype:**

**Invalidate();**

**Remarks:**

This method invalidates the validity interval.

**Default Implementation:**

**{ mValid = NEVER; }**

**Prototype:**

**void UnRegisterViewChange();**

**Remarks:**

This method unregister the notifications so changes in the view are no longer registered.

**Prototype:**

**void SetResettingFlag(BOOL val);**

**Remarks:**

Used internally.

**Prototype:**

**BOOL GetResettingFlag();**

**Remarks:**

Used internally.

**Prototype:**

**void KillToolTip();**

**Remarks:**

This method will destroy the tooltip and its timer and cleans up.

**Prototype:**

**Point3 GetUnselectedColor();**

**Remarks:**

This method returns the color of the gizmo when unselected.

## Class ToneOperator

See Also: [Class SpecialFX](#), [Class ToneOperatorInterface](#), [Class IRendParams](#), [Class RendParams](#), [Class RenderGlobalContext](#), [Class RenderMapsContext](#), [Class ShadeContext](#), [Class Interval](#).

```
class ToneOperator : public SpecialFX
```

### Description:

This class is available in release 4.0 and later only.

This is the base class for the creation of Tone Operator plug-ins. A Tone Operator performs two functions:

1. It converts physically based values to RGB for filtering and display. The renderer calls the tone operator immediately after **Mtl::Shade** is called.
2. It balances physical and non-physical lighting.

The tone operator balances the physical and non-physical lighting by providing a scale relationship between them. The scale converts physical candelas to the non-physical value 1.0. Physically based objects in the 3ds max scene use this scale to convert their values for use by the renderer and materials. The tone operator then converts the scaled value to RGB for display.

An example of this is the tone operator for a radiosity plug-in. 3ds max works in a lighting space where values run from 0 to 1 and don't have any meaning. Pre-rendered Reflection maps, Refraction maps and self-illumination maps also use a 0 to 1 scale without any meaning. A radiosity plug-in introduces physical values to 3ds max that range from 0 to 90000 for the sun.

So the question is "How do we mix these values with physical values?" One solution is to use a scale. Physical values are scaled to "3ds max" values. Then they are processed by the shaders and materials, and then the scaled values are converted to RGB by ScaledToRGB.

So the PhysicalUnits, ScalePhysical and ScaleRGB are used to convert from 3ds max lighting values to physical lighting values. We can use this to balance 3ds max lights with physical lights, and to assign physical values to 3ds max lights when we want to use them in a radiosity solution.

The tone operator may include a UI that allows the user to set the scale, or it can set the scale apriori. The scale is also used for reflection maps, which are usually implemented in 32 bit bitmaps. If the scale is set too high, reflection maps can show banding because of rounding errors. If the scale is set too low, reflection

maps can wash out because of clipping values to 0...255.

The tone operator uses the standard Special Effects parameter dialog class for its user interface.

Note: **typedef SFXParamDlg ToneOpParamDlg;**

### **Methods:**

public:

### **Prototype:**

```
virtual ToneOpParamDlg *CreateParamDialog(IRendParams  
*ip);
```

### **Remarks:**

This method creates the rollup pages in the render panel that lets the user edit the tone operator's parameters. You can use

**IRendParams::AddRollupPage** and

**IRendParams::DeleteRollupPage** to manage your rollup pages directly.

Or, if your parameters are stored in a ParamBlock2 object, you can use

**CreateRParamMap2** and **DestroyRParamMap2** to manage the rollups.

You may return NULL, if no UI is required.

### **Parameters:**

**IRendParams \*ip**

Points to the render parameter dialog interface. You may call the methods of that class using this pointer.

### **Return Value:**

Pointer to the tone operator dialog.

### **Default Implementation:**

```
{ return NULL; }
```

### **Prototype:**

```
virtual void SetActive(bool active, TimeValue t);
```

### **Remarks:**

Implemented by the Plug-In.

This method indicates whether the tone operator is active. The default implementation does not use the **TimeValue t**. The result of the default

implementation can be retrieved using **SpecialFX::GetActive**. If you override this method and change the mechanism for storing this state, you should also override **SpecialFX::GetActive** so the correct state is returned.

**Parameters:**

**bool active**

A boolean indicating if the tone operator is active.

**TimeValue t**

The time at which the active check is made.

**Default Implementation:**

```
if (active ^ (TestAFlag(A_ATMOS_DISABLED) == 0)) {  
    if (active) {  
        ClearAFlag(A_ATMOS_DISABLED);  
    }  
    else {  
        SetAFlag(A_ATMOS_DISABLED);  
    }  
    NotifyDependents(FOREVER, PART_ALL,  
REFMSG_CHANGE);  
}
```

**Prototype:**

```
virtual BOOL SetDlgThing(ToneOpParamDlg* dlg);
```

**Remarks:**

Implemented by the Plug-In.

Implement this if you are using the ParamMap2 AUTO\_UI system and the atmosphere has secondary dialogs that don't have the effect as their 'thing'.

Called once for each secondary dialog for you to install the correct thing.

**Parameters:**

**ToneOpParamDlg\* dlg**

Points to the tone operator dialog.

**Return Value:**

Return TRUE if you process the dialog, FALSE otherwise.

**Prototype:**

```
virtual bool IsPhysicalSpace() const;
```

**Remarks:**

Implemented by the Plug-In.

Returns a boolean which indicates if this tone operator really maps physical values to RGB. This method is provided so shaders can determine whether the shading calculations are in physical or RGB space.

**Default Implementation:**

```
{ return true; }
```

**Prototype:**

```
virtual void Update(TimeValue t, Interval& valid);
```

**Remarks:**

This method is called once per frame when the renderer begins. This gives the tone operator the chance to cache any values it uses internally so they don't have to be computed on every pixel. But, this method should **not** be used to perform any very long tasks. This would be the likely method that caches the frames physical scaling value.

**Parameters:****TimeValue t**

The time at which the rendering is beginning.

**Interval& valid**

The validity interval for the update.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual bool BuildMaps(TimeValue t, RenderMapsContext& rmc);
```

**Remarks:**

This method is called for the operator to do any work it needs to do prior to rendering. You may use this method to perform a subrender to sample the

rendered output for histogramming or automatic exposure.

**Parameters:**

**TimeValue t**

The time at which the rendering is taking place.

**RenderMapsContext& rmc**

The context of the map rendering.

**Return Value:**

True means this method succeeded. False means it didn't. This method should return false if it the sub-render fails or if it can't allocate memory or some other error occurs. If BuildMaps returns false, the render is aborted.

**Default Implementation:**

```
{ return true; }
```

**Prototype:**

```
virtual void ScaledToRGB(float energy[3]) = 0;
```

**Remarks:**

This method maps a scaled energy value into RGB. This version converts a color value. The converted color value is stored in **energy**.

This method assumes that **Update()** has been called to cache the various values needed by the tone operator.

Note: By using a float array to pass in color values, we can use the same routine to handle the various classes used to store color information, for example, Color, AColor and Point3.

**Parameters:**

**float energy[3]**

The input energy value to convert. The converted color value is stored here as well. The red, green and blue components are stored in that order in the array. The valid ranges are -infinity to +infinity, but the returned value is clipped by the renderer very quickly to [0,1]. The tone operator can do it's own clipping, but it isn't required.

**Prototype:**

```
virtual float ScaledToRGB(float energy) = 0;
```

**Remarks:**

This method maps a scaled energy value to monochrome. The converted monochrome value is returned. This method assumes that **Update()** has been called to cache the various values needed by the tone operator.

**Parameters:****float energy**

The input energy value to convert.

**Prototype:**

```
virtual float GetPhysicalUnit(TimeValue t,Interval& valid = Interval(0,0)) const = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method returns the physical value that is scaled to 1.

**Parameters:****TimeValue t**

The time at which to return the value.

**Interval& valid = Interval(0,0)**

The validity interval for the value.

**Prototype:**

```
virtual void SetPhysicalUnit(float value, TimeValue t) = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method sets the physical value that is scale to 1. This is simply a programatic method to change the physical scaling of the tone operator. Valid values are (0,+infinity).

**Parameters:****TimeValue t**

The time at which to set the value.

**Interval& valid = Interval(0,0)**

The validity interval for the value.

**Prototype:**

```
virtual void ScalePhysical(float energy[3]) const = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method is used to scale a physical color value so it may be used in the renderer.

This method assumes that Update has been called to cache the various values needed by the tone operator.

Note: By using a float array to pass in color values, we can use the same routine to handle the various classes used to store color information, for example, Color, AColor and Point3.

**Parameters:**

**float energy[3]**

The input and output (converted) color value. The colors are stored as red=energy[0], green=energy[1], and blue=energy[2].

**Prototype:**

```
virtual float ScalePhysical(float energy) const = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method is used to scale a physical monochrome value so it may be used in the renderer.

This method assumes that Update has been called to cache the various values needed by the tone operator.

**Parameters:**

**float energy**

The input value to scale.

**Return Value:**

The scaled output value is returned.

**Prototype:**

```
virtual void ScaleRGB(float color[3]) const = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method is called to scale RGB values (the inverse of ScalePhysical()).

This method assumes that Update has been called to cache the various values needed by the tone operator.

**Parameters:**

**float color[3]**

The input values to scale and storage for the output scaled values as well. The colors are stored as red=energy[0], green=energy[1], and blue=energy[2]. The output values are in the range 0-1.

**Prototype:**

**virtual float ScaleRGB(float color) const = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to scale a monochrome value (the inverse of ScalePhysical()).

**Parameters:**

**float color**

The input value to scale.

**Return Value:**

The scaled output value is returned.

**Prototype:**

**bool GetProcessBackground();**

**Remarks:**

Returns the state of **A\_TONEOP\_PROCESS\_BG**, indicating whether the ToneOperator will be processing the background.

**Prototype:**

**void SetProcessBackground(bool active);**

**Remarks:**

This method allows you to set **A\_TONEOP\_PROCESS\_BG**.

**Parameters:**

**bool active**

TRUE to activate, FALSE to deactivate.

## Class IMultiPassCameraEffect

See Also: [Class ReferenceTarget](#), [Class CameraObject](#).

```
class IMultiPassCameraEffect : public ReferenceTarget
```

### Description:

This class is available in release 4.0 and later only.

The multipass camera effect allows modification of viewpoints & view directions or time for each pass of a multipass rendering. Algorithms such as Depth of Field, Scene motion blur can be implemented using multipass techniques.

Basically, a multipass camera effect is a plug-in to camera objects. It allows the renderer to query the camera for the view params for each pass of the rendering, & provides a dithered combining function to combine the bitmaps produced by each pass into the final bitmap. It also allows time to be manipulated for each rendering pass, providing effects such as motion blur.

### Methods:

public:

### Prototype:

```
virtual bool IsCompatible(CameraObject *pCameraObject) = 0;
```

### Remarks:

Some cameras are not compatible with some render effects, this method allows cameras to list compatible effects in the UI and as such allows the effect to declare its compatibility with the current camera object.

### Parameters:

**CameraObject \*pCameraObject**

A pointer to a camera object.

### Return Value:

TRUE if compatible, otherwise FALSE.

### Prototype:

```
virtual bool DisplayPasses(TimeValue renderTime) = 0;
```

### Remarks:

There is a UI option on multipass effects that indicates whether the renderer should display each pass as it is rendered. Note this is not used by viewport renderer, because of the hardware involvement. This method returns whether to display individual passes as they are computed.

**Parameters:**

**TimeValue renderTime**

The rendertime at which to check the display passes.

**Return Value:**

TRUE if display is on, otherwise FALSE.

**Prototype:**

**virtual int TotalPasses(TimeValue renderTime) = 0;**

**Remarks:**

The multipass effect also has a variable number of passes. This method tells the renderer how many passes to render per final output frame and as such returns the total number of passes to be rendered

**Parameters:**

**TimeValue renderTime**

The rendertime at which to check the display passes.

**Prototype:**

**virtual ViewParams \*Apply(INode \*pCameraNode,  
CameraObject \*pCameraObject, int passNum, TimeValue  
&overrideRenderTime) = 0;**

**Remarks:**

This method will modify the camera, camera node, or time value to affect each pass.

This is the modify function called for each pass of the frame. The effect can alter the camera node, camera object, or override the render time in the course of this call. 3ds max renderers take an optional parameter viewParams\* that when not NULL overrides the normal rendering camera. When this is called the override render time will be set to the current frame time. If the value is changed, this will be the time value used for the pass. Note that at the time that

apply is called, the renderer has not yet been called, hence it is possible, with care, to alter the scene in a general way, not just the camera & time parameters. Apply should return NULL if the normal unmodified camera is to be used.

**Parameters:**

**INode \*pCameraNode**

A pointer to the node of the camera.

**CameraObject \*pCameraObject**

A pointer to the camera object.

**int passNum**

The number of the pass.

**TimeValue &overrideRenderTime**

The time if you wish to override the render time.

**Return Value:**

The viewparams returned by apply which are supplied to the renderer.

**Prototype:**

```
virtual void AccumulateBitmap(Bitmap *pDest, Bitmap *pSrc, int  
passNum, TimeValue renderTime) = 0;
```

**Remarks:**

This method will blend each pass (**src**) into the final accumulator (**dest**).

After each pass is rendered, it needs to be combined into the final output bitmap. The current multipass effects use a dithered combiner, so that hard edges from the passes are more smoothly blended. There are many ways to do this, with varying quality, so this method allows different future implementations. Note that this is not used by the viewport renderer, as there's no way to tell the hardware to do this. Hardware is for fast & edgy, software is for slow & smooth.

**Parameters:**

**Bitmap \*pDest**

The destination bitmap.

**Bitmap \*pSrc**

The source bitmap.

**int passNum**

The number of the pass.

**TimeValue renderTime**

The render time.

**Prototype:**

**virtual void PostRenderFrame() = 0;**

**Remarks:**

This method is called after all passes have been rendered.

After all passes have been rendered & accumulated, this method will be called so that the effect can do any final cleanup. Currently unused, it can be ignored by multipass effects if they wish.

**Prototype:**

**virtual RefResult NotifyRefChanged(Interval changeInt,  
RefTargetHandle hTarget, PartID& partID, RefMessage  
message);**

**Remarks:**

This method is implemented to receive and respond to messages broadcast by all the dependants in the entire system.

**Parameters:**

**Interval changeInt**

This is the interval of time over which the message is active.

**RefTargetHandle hTarget**

This is the handle of the reference target the message was sent by. The reference maker uses this handle to know specifically which reference target sent the message.

**PartID& partID**

This contains information specific to the message passed in. Some messages don't use the partID at all. See the section [List of Reference Messages](#) for more information about the meaning of the partID for some common messages.

**RefMessage message**

The msg parameters passed into this method is the specific message which needs to be handled. See [List of Reference Messages](#).

**Return Value:**

The return value from this method is of type RefResult. This is usually **REF\_SUCCEED** indicating the message was processed. Sometimes, the return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

**Default Implementation:**

```
{ return REF_SUCCEED; }
```

**Prototype:**

```
virtual SClass_ID SuperClassID();
```

**Remarks:**

This method returns the super class ID of the creator of the clip object.

**Default Implementation:**

```
{ return MPASS_CAM_EFFECT_CLASS_ID; }
```

## Class IReshading

See Also: [Class IReshadeFragment](#), [Class ShadeContext](#), [Class IllumParams](#)

class IReshading

### Description:

This class is available in release 4.0 and later only.

This interface class is the interface that must be supported by materials and shaders to support interactive rendering in 3ds max. If this interface is not returned when requested through **GetInterface()** on the mtl or shader, then it is determined they do not support reshading. Any material that does not support reshading is evaluated only at preshading time.

In 3ds max, interactive rendering is implemented as fragment based caching scheme. It's very much like a giant multi-layer g-buffer with enough information stored with each fragment to re-run the material shading process without re-rendering, all stored in a compressed form.

The rendering process is divided into 2 parts: preshading and reshading. Preshading builds the scene and renders fragments to the compressed g-buffer, called the **rsBuffer**. To do this, it saves a minimal amount of information with each fragment, then gives each material and texture the opportunity to pre-shade itself.

Basically, the reshader populates a minimal **ShadeContext**, with the fragment center position in camera space (where shading is done), the shading normal, the sub-pixel mask, the coverage, pointers to the original material and **INode**, light lists, screen bounding box of object, the renderID, and screen integer x and y.

Any other values from the shade context that a material, shader, texture(future), atmosphere(future) or light(future) needs to save, it does so at pre-shade time, and retrieves the data at reshading time. So, for example, the multi-materials save which material is visible, anisotropic shaders save the Anisotropic reference vector.

### Methods:

public:

### Prototype:

**virtual ReshadeRequirements GetReshadeRequirements();**

**Remarks:**

This method is called by the interactive reshader after being notified that the material has changed. The return value indicates if the material is still valid, or needs to be pre-shaded or reshaded. The value should pertain only to the latest change to the material. If a material doesn't support reshading, it doesn't need to override this function -- any change will cause the nodes to which it's attached to be re- preShaded.

**Return Value:**

One of the following values;

**RR\_None**

No actions needs to be taken.

**RR\_NeedPreshade**

A pre-shade is needed.

**RR\_NeedReshade**

A re-shade is needed.

**Default Implementation:**

```
{ return RR_NeedPreshade; }
```

**Prototype:**

```
virtual void PreShade(ShadeContext& sc, IReshadeFragment* pFrag);
```

**Remarks:**

This method will pre-shade the object and cache any needed values in the fragment.

This method is called on the materials/shaders/textures as the reshading buffer is being built at the same point in the rendering process where the materials shade function would normally be called. Note that at this time the shade context is completely valid, as in a normal rendering. Any values the material wishes to cache can be attached to the reshading fragment passed in, and retrieved later at postShade time.

**Parameters:****ShadeContext& sc**

A reference to the shade context.

## **IReshadeFragment\* pFrag**

A pointer to the fragment.

### **Default Implementation:**

```
{ }
```

### **Prototype:**

```
virtual void PostShade(ShadeContext& sc, IReshadeFragment*  
pFrag, int& nextTexIndex, IllumParams* ip = NULL);
```

### **Remarks:**

This method will retrieve cached values and compute shade & transparency for the fragment.

This method is called for the material of each fragment as the reshading buffer is being traversed. Materials retrieve whatever they may have stored to complete the minimal shade context and compute a shade for the fragment.

**PostShade()** calls to shaders contain an additional parameter, the **IllumParams**, filled with the textured/blended but unshaded values for each texture. The shade context passed into **PostShade()** is the minimal shade context outlined above.

### **Parameters:**

#### **ShadeContext& sc**

A reference to the shade context.

#### **IReshadeFragment\* pFrag**

A pointer to the fragment.

#### **int& nextTexIndex**

A reference to the next texture index.

#### **IllumParams\* ip = NULL**

A pointer to the IllumParams containing textured/blended but unshaded values for each texture.

### **Default Implementation:**

```
{ }
```

## Class IInteractiveRender

See Also: [Class InterfaceServer](#), [Class IIRenderMgr](#), [Class INode](#), [Class ViewExp](#), [Class Bitmap](#), [Class DefaultLight](#), [Class IRenderProgressCallback](#), [Class Animatable](#) , [Class ActionTable](#)

```
class IInteractiveRender : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This is the abstract class (Interface) for a a renderer supporting interactive rendering.

With the likelihood of evolving rendering and shading techniques which are going to be markedly different from what is being used now, the 3ds max SDK provides the infrastructure to support interactive rendering. Since renderers are a plugin to 3ds max and since each renderer has a different set of resources and capabilities the interactive rendering and shading API has been made as general and independent as possible. The independence of the renderer also means that the renderer must do a fair amount of work that could be done by the system for all renderers.

Interactive rendering and shading is a separate interface which can be optionally supported by 3<sup>rd</sup> party renderers and can be obtained by using the

**GetInterface()** method on the renderer. **I\_RENDER\_ID** is passed as the ID parameter to a renderer's implementation of **Animatable::GetInterface()**.

The renderer returns a pointer to a class **IInteractiveRender** instance if it supports interactive rendering, otherwise the default implementation will return NULL, indicating that interactive rendering is not supported. This class,

**IInteractiveRender** is defined in

**\MAXSDK\INCLUDE\interactiveRendering.h.**

As it is, interactive renderers will reference any and all objects, as well as lights and materials in the scene that they can respond to changes from. Via the normal 3ds max notification system the interactive shader will then receive messages whenever one of these referenced objects changes and it must then decide how best to update the scene for the user.

### Methods:

public:

**Prototype:**

```
virtual void BeginSession() = 0;
```

**Remarks:**

This method initiates the interactive rendering and will reference objects in the scene so the renderer receives the proper change notifications. The interactive renderer will automatically update the scene when these changes are received.

This is called when the user first invokes interactive rendering. During the begin session call, the renderer should reference any object in the scene that it can respond to changes from. There are reference manager classes in the SDK to ease this referencing in the file **referenceManager.h**. many of the parameters of interactive reshading are set up prior to the call to **BeginSession()**, here is a code snippet from the interactive rendering manager:

```
mIRenderInterface->SetOwnerWnd(mpIMaxBitmapViewer-  
>GetHDisplayWindow());
```

```
mIRenderInterface->SetIIRenderMgr(this);
```

```
mIRenderInterface->SetBitmap(mpIMaxBitmapViewer-  
>GetBitmap());
```

```
mIRenderInterface->SetSceneINode(mpInterface-  
>GetRootNode());
```

```
mIRenderInterface->SetUseViewINode(false);
```

```
mIRenderInterface->SetViewINode(NULL);
```

```
mIRenderInterface->SetViewExp(mpViewExp);
```

```
mIRenderInterface->SetRegion(mSelectedRegion);
```

```
mIRenderInterface->SetDefaultLights(mDefaultLights,  
mNumDefaultLights);
```

```
mIRenderInterface->SetProgressCallback(  
    dynamic_cast<IRenderProgressCallback *>  
    (&mImageViewerCB)  
);
```

```
mIRenderInterface->BeginSession();
```

Consequently, **BeginSession()** can rely on all these local attributes being

valid.

**Prototype:**

**virtual void EndSession() = 0;**

**Remarks:**

This method will end an interactive rendering session and will remove all references placed on the scene.

**Prototype:**

**virtual void SetOwnerWnd(HWND hOwnerWnd) = 0;**

**Remarks:**

This method allows you to set the owner window, which could come in handy if you want to pass it to the renderer if necessary. The owner HWND is supplied to the interactive renderer so that it may receive window messages for the interactive window, update the window, etc.

**Parameters:**

**HWND hOwnerWnd**

A handle to the owner window.

**Prototype:**

**virtual HWND GetOwnerWnd() const = 0;**

**Remarks:**

This method allows you to retrieve the owner window.

**Prototype:**

**virtual void SetIIRenderMgr(IIRenderMgr \*pIIRenderMgr) = 0;**

**Remarks:**

This method sets a pointer to the controlling renderMgr, so that various states can be queried.

**Parameters:**

**IIRenderMgr \*pIIRenderMgr**

A pointer to the controlling interactive rendering manager.

**Prototype:**

```
virtual IIRenderMgr *GetIIRenderMgr(IIRenderMgr  
*pIIRenderMgr) const = 0;
```

**Remarks:**

This method allows you to retrieve a pointer to the controlling interactive render manager

**Parameters:**

**IIRenderMgr \*pIIRenderMgr**

A pointer to the render manager interface.

Used internally. This should always be set to NULL.

**Prototype:**

```
virtual void SetBitmap(Bitmap *pDestBitmap) = 0;
```

**Remarks:**

This method allows you to set the destination bitmap to be rendered and re-rendered to. This destination bitmap is persistent between update renderings, basically the renderer holds the bitmap and updates the screen while the manager holds the reference to the bitmap and controls its lifetime.

**Parameters:**

**Bitmap \*pDestBitmap**

A pointer to the destination bitmap to set.

**Prototype:**

```
virtual Bitmap *GetBitmap(Bitmap *pDestBitmap) const = 0;
```

**Remarks:**

This method allows you to retrieve the destination bitmap that's being rendered and re-rendered to.

**Parameters:**

**Bitmap \*pDestBitmap**

A pointer to the destination bitmap.

Used internally. This should always be set to NULL.

**Prototype:**

```
virtual void SetSceneINode(INode *pSceneINode) = 0;
```

**Remarks:**

This method allows you to set the scene root node. In general, the idea of interactive rendering is to start with a fixed scene and then respond to changes in that scene. This call sets the scene root node. All items stemming from this scene root that an interactive renderer can respond to changes in should be referenced.

**Parameters:**

```
INode *pSceneINode
```

A pointer to the scene root node.

**Prototype:**

```
virtual INode *GetSceneINode() const = 0;
```

**Remarks:**

This method returns a pointer to the scene's root node.

**Prototype:**

```
virtual void SetUseViewINode(bool bUseViewINode) = 0;
```

**Remarks:**

This method sets whether to use the ViewINode. When a separate camera node is needed instead of ViewExp, the interactive rendering manager will set the viewINode to the interactive renderer and set UseViewINode to TRUE.

**Parameters:**

```
bool bUseViewINode
```

If FALSE, ViewParams obtained from ViewExp should be used.

**Prototype:**

```
virtual bool GetUseViewINode() const = 0;
```

**Remarks:**

This method returns FALSE if the ViewParams obtained from ViewExp should be used. TRUE would indicate that this is not the case.

**Prototype:**

```
virtual void SetViewINode(INode *pViewINode) = 0;
```

**Remarks:**

This method allows you to set the View INode in case a separate camera node is needed instead of ViewExp.

**Parameters:**

**INode \*pViewINode**

A pointer to the view node.

**Prototype:**

```
virtual INode *GetViewINode() const = 0;
```

**Remarks:**

This method returns a pointer to the view node if this is used instead of ViewExp.

**Prototype:**

```
virtual void SetViewExp(ViewExp *pViewExp) = 0;
```

**Remarks:**

This method allows you to set the ViewExp. The ViewExp is the view specification for docked windows. The interactive renderer gets the view params out of the ViewExp.

**Parameters:**

**ViewExp \*pViewExp**

A pointer to the ViewExp.

**Prototype:**

```
virtual ViewExp *GetViewExp() const = 0;
```

**Remarks:**

This method returns a pointer to the ViewExp which is used for the view specification for docked windows.

**Prototype:**

**virtual void SetRegion(const Box2 &region) = 0;**

**Remarks:**

This method allows you to set the region of the bitmap to be rendered.

There are two standard interactive modes that should be supported in all interactive renderers: region rendering and selected object rendering, and these modes should ideally work in consort if at all possible: scenes are often very complex and the plugin renderer must be able to limit complexity to increase interactivity. Note that if **Box2::IsEmpty()** returns TRUE, it indicates to render entire bitmap.

**Parameters:**

**const Box2 &region**

A reference to the rectangular area.

**Prototype:**

**virtual const Box2 &GetRegion() const = 0;**

**Remarks:**

This method returns the region of the bitmap to be rendered. Note that if **Box2::IsEmpty()** returns TRUE, it indicates to render entire bitmap.

**Prototype:**

**virtual void SetDefaultLights(DefaultLight \*pDefLights, int numDefLights) = 0;**

**Remarks:**

This method allows you to set the default lights to be used in absence of scene lights. These lights will be used when no user specified lights are in the scene. This should be noted when the scene is traversed in begin session, and of course altered if new user lights are created.

**Parameters:**

**DefaultLight \*pDefLights**

A pointer to a default light source.

**int numDefLights**

The number of default lights.

**Prototype:**

```
virtual const DefaultLight *GetDefaultLights(int &numDefLights)  
const = 0;
```

**Remarks:**

This method returns a pointer to the default lights and the number of default lights which are used in absence of scene lights.

**Parameters:**

**int &numDefLights**

The number of default lights returned.

**Prototype:**

```
virtual void SetProgressCallback(IRenderProgressCallback  
*pProgCB) = 0;
```

**Remarks:**

This method allows you to set an interactive rendering progress callback object.

The Progress/Abort Callback should be called by the renderer ideally about every 100 milliseconds, but the actual range varies widely. The callback allows the manager to display rendering progress and/or abort a rendering.

**Parameters:**

**IRenderProgressCallback \*pProgCB**

A pointer to the interactive rendering progress callback object.

**Prototype:**

```
virtual const IRenderProgressCallback *GetProgressCallback()  
const = 0;
```

**Remarks:**

This method returns a pointer to the interactive rendering progress callback object.

**Prototype:**

```
virtual void Render(Bitmap *pDestBitmap) = 0;
```

**Remarks:**

This method renders the bitmap using default non-interactive rendering functionality. This is the only actual command to the interactive renderer.

**Parameters:****Bitmap \*pDestBitmap**

The destination bitmap to render to.

**Prototype:**

```
virtual ULONG GetNodeHandle(int x, int y) = 0;
```

**Remarks:**

This method returns the closest node handle for a given bitmap pixel location. This can be implemented with an item buffer, by using ray casting, or some other method and allows the interactive rendering manager to implement object selection.

**Parameters:****int x, int y**

The x and y coordinate of the bitmap pixel.

**Return Value:**

The node handle or 0 if there is no node.

**Prototype:**

```
virtual bool GetScreenBBox(Box2& sBBox, INode *pINode) = 0;
```

**Remarks:**

This method returns the screen bounding box of the corresponding INode, so the selection box corners can be drawn.

**Parameters:****Box2& sBBox**

The screen bounding box.

**INode \*pINode**

The INode for which you wish to retrieve the screen bounding box.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:****virtual ActionTableId GetActionTableId() = 0;****Remarks:**

This method returns the ActionTableId for any action items the renderer may implement. This method will return 0 if none are available. Action tables are used as context sensitive command system to generate quad menus and the like from the various objects in the scene.

**Prototype:****virtual ActionCallback \*GetActionCallback() = 0;****Remarks:**

This method returns a pointer to an ActionCallback for any action items the renderer may implement. This method will return NULL if none are available.

**Prototype:****virtual void \*GetInterface();****Remarks:**

This method provides a general extension mechanism, access to additional method interfaces.

**Default Implementation:****{ return NULL; }****Prototype:****virtual BOOL IsRendering() = 0;****Remarks:**

This method returns TRUE if the renderer is currently rendering, otherwise FALSE.

When the interactive rendering manager gets a message to shut down or abort an interactive rendering, there is a potential race condition between the interactive renderer shutting down and the shutting down of the manager itself, which deletes the renderer. Since it is the renderer itself that decides when, what and how to re-render the image, it's not clear to the manager whether a delete is safe. This method allows the manager to inquire whether

the renderer is recomputing the image. To abort a rendering, the progress/abort callback must be used. When the abort is complete, IsRendering will return FALSE.

## Class IMBOps

See Also: [Class FPStaticInterface](#), [Class Bitmap](#), [Class CheckAbortCallback](#).

class IMBOps: public FPStaticInterface

### Description:

This class is available in release 4.0 and later only.

This class is an interface for Image Motion Blur. This interface is implemented in the Effect plug-in MotionBlur.dlv, which must be present to use it. This interface does NOT support scripting, only direct calling. The sample code below shows how this is done:

### Sample Code:

```
ClassDesc2* mbcd = GET_MBLUR_CD;
if (mbcd) {
    IMBOps* imb = GetIMBInterface(mbcd);
    imb->ApplyMotionBlur(bm, &imbc, 1.2f);
}
```

### Methods:

public:

### Prototype:

```
virtual ULONG ChannelsRequired(ULONG flags=0)=0;
```

### Remarks:

Sets the channels required for the image motion blur.

### Parameters:

**ULONG flags=0**

The following flag may be set:

**IMB\_TRANSP**

Controls whether motion blur works through transparency. Setting it to 0 saves memory, runs faster.

### Prototype:

```
virtual int ApplyMotionBlur(Bitmap *bm, CheckAbortCallback
```

**\*progCallback=NULL, float duration=1.0f, ULONG flags=IMB\_TRANSP, Bitmap\* extraBM=NULL)=0;**

**Remarks:**

Applies the motion blur process to the specified bitmap.

**Parameters:**

**Bitmap \*bm**

The bitmap to apply the motion blur to.

**CheckAbortCallback \*progCallback=NULL**

A pointer to a callback, allowing an abort check during the progress.

**float duration=1.0f**

The motion blur duration.

**ULONG flags=IMB\_TRANSP**

The following flag may be set:

**IMB\_TRANSP**

Controls whether motion blur works through transparency. Setting it to 0 saves memory, runs faster.

**Bitmap\* extraBM=NULL**

If the **extraBM** bitmap is supplied, then that is used as the target color bitmap, but the gbuffer information still comes from the other, main, bitmap. this is used to apply motion blur to render elements.

**Return Value:**

TRUE if success, otherwise FALSE.

## Class IChkMtlAPI

See Also: [Class Mtl](#), [Class RenderInstance](#).

class IChkMtlAPI

### Description:

This class is available in release 4.0 and later only.

This class provides interface methods used to support indirect material referencing and enhanced face / material associations. Generally, these methods will need to be implemented by the plug-in using them and cannot be called from a standard library, since the information required is intimately associated with the geometry of the object.

All methods of this class are implemented by the Plug-In.

### Methods:

public:

### Prototype:

**virtual BOOL SupportsParticleIDbyFace();**

### Remarks:

Returns TRUE if the object can associate a particle number with a face number, and FALSE if not.

### Default Implementation:

**{ return FALSE; }**

### Prototype:

**virtual int GetParticleFromFace(int faceID);**

### Remarks:

Returns the particle to which the face identified by faceID belongs.

### Parameters:

**int faceID**

The ID of the face.

### Default Implementation:

**{ return 0; }**

**Prototype:**

**virtual BOOL SupportsIndirMtlRefs();**

**Remarks:**

Returns TRUE if the object can return a material pointer given a face being rendered, and FALSE if the object will be associated for that render pass with only the material applied to the node.

**Default Implementation:**

**{ return FALSE; }**

**Prototype:**

**virtual int NumberOfMtlsUsed();**

**Remarks:**

Returns the number of different materials used on the object. This number is used in enumerating the different materials via the methods **GetNthMtl()** and **GetNthMaxMtlID()** below.

**Default Implementation:**

**{ return 0; }**

the following methods are meaningful.

**Prototype:**

**virtual Mtl \*GetNthMtl(int n);**

**Remarks:**

If the method **SupportsIndirMtlRefs()** above returns TRUE then this method returns the different materials used on the object.

**Parameters:**

**int n**

The zero based index of the material used on the object.

**Return Value:**

**Default Implementation:**

**{ return NULL; }**

**Prototype:**

```
virtual int GetNthMaxMtlID(int n);
```

**Remarks:**

If the method **SupportsIndirMtlRefs()** above returns TRUE then this method returns the maximum material ID number used with the specified material on the object

**Parameters:**

**int n**

The zero based index of the material.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual Mtl *GetMaterialFromFace(int faceID);
```

**Remarks:**

If the method **SupportsIndirMtlRefs()** above returns TRUE then this method returns a pointer to the material associated with the face identified by faceID.

**Parameters:**

**int faceID**

The ID of the face to check.

**Default Implementation:**

```
{ return NULL; }
```

## Class IImageViewer

See Also: [Class DADMgr](#) , [Class BaseInterface](#), [Class IMaxBitmapViewer](#)

class IImageViewer

### Description:

This class is available in release 4.0 and later only.

This class represents an abstract interface class for an image viewer.

### Methods:

public:

### Prototype:

```
virtual void PostDisplayCallback::PostDisplayCB(HWND hWnd)  
= 0;
```

### Remarks:

The PostDisplayCB method of the PostDisplayCallback class is called after an image is displayed in the image viewer. Developers can use this to do any post-display related work.

### Parameters:

**HWND hWnd**

The handle of the image viewer window.

### Prototype:

```
virtual BaseInterface*  
PostDisplayCallback::GetInterface(Interface_ID id);
```

### Remarks:

This method provides a way to extend the class with interfaces.

### Parameters:

**Interface\_ID id**

The interface ID.

### Default Implementation:

```
{ return NULL; }
```

**Prototype:**

```
virtual LRESULT  
PreEventHandlerCallback::EventHandlerCB(HWND hWnd,  
UINT message, WPARAM wParam, LPARAM lParam, bool  
&propagate) = 0;
```

**Remarks:**

The EventHandlerCB method of the PreEventHandlerCallback class allows you to intercept window events prior to them being passed through.

**Parameters:****HWND hWnd**

The handle of the image viewer window.

**UINT message**

The message identifier.

**WPARAM wParam**

The WPARAM value.

**LPARAM lParam**

The LPARAM value.

**bool &propagate**

TRUE if the message and event are to be propagated, otherwise FALSE.

**Prototype:**

```
virtual BaseInterface*  
PreEventHandlerCallback::GetInterface(Interface_ID id);
```

**Remarks:**

This method provides a way to extend the class with interfaces.

**Parameters:****Interface\_ID id**

The interface ID.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

**virtual void Show() = 0;**

**Remarks:**

This method will show the image viewer. In the **IMaxBitmapViewer** class, the **Display()** method must initially be used to display the viewer; this method only displays the viewer after using **Hide()**.

**Prototype:**

**virtual void Hide() = 0;**

**Remarks:**

This method will hide the image viewer. In the **IMaxBitmapViewer** class, the **UnDisplay()** method should be used to close the viewer when done; use **Hide()** to temporarily hide the window.

**Prototype:**

**virtual HWND GetHDisplayWindow() = 0;**

**Remarks:**

This method returns the handle to the display window of the image viewer.

**Prototype:**

**virtual void SetPos(int x, int y, int w, int h) = 0;**

**Remarks:**

This method allows you to set the position of the image viewer window.

**Parameters:**

**int x, y**

The position of the window.

**int w, h**

The width and height of the window.

**Prototype:**

**virtual DisplayStyle GetDisplayStyle() const = 0;**

**Remarks:**

This method will return the display style used by the image viewer window,

which is either **IV\_FLOATING** or **IV\_DOCKED**.

**Prototype:**

**virtual void SetContextHelpId(DWORD helpID) = 0;**

**Remarks:**

This method allows you to set the context help identifier for the image viewer.

**Parameters:**

**DWORD helpID**

The help identifier.

**Prototype:**

**virtual DWORD GetContextHelpId() const = 0;**

**Remarks:**

This method returns the context help identifier.

**Prototype:**

**virtual void SetDADMgr(DADMgr \*pDADMgr) = 0;**

**Remarks:**

This method allows you to set the drag and drop manager which should be used for the image viewer.

**Parameters:**

**DADMgr \*pDADMgr**

A pointer to the drag and drop manager.

**Prototype:**

**virtual void**

**SetPreEventHandlerCallback(PreEventHandlerCallback\*  
pPreEventHandlerCB) = 0;**

**Remarks:**

This method allows you to set the pre-event handler callback method.

**Parameters:**

## **PreEventHandlerCallback\* pPreEventHandlerCB**

A pointer to the callback function.

### **Prototype:**

```
virtual PreEventHandlerCallback*  
GetPreEventHandlerCallback() const = 0;
```

### **Remarks:**

This method returns a pointer to the pre-event handler callback function.

### **Prototype:**

```
virtual void SetPostDisplayCallback(PostDisplayCallback*  
pPostDisplayCB) = 0;
```

### **Remarks:**

This method allows you to set the post display callback function.

### **Parameters:**

```
PostDisplayCallback* pPostDisplayCB  
A pointer to the callback function.
```

### **Prototype:**

```
virtual PostDisplayCallback* GetPostDisplayCallback() const = 0;
```

### **Remarks:**

This method returns a pointer to the post display callback function.

### **Prototype:**

```
virtual BaseInterface* GetInterface(Interface_ID id);
```

### **Remarks:**

This method provides a way to extend the class with interfaces.

### **Parameters:**

```
Interface_ID id  
The interface ID.
```

### **Default Implementation:**

```
{ return NULL; }
```

## Class IMenuManager

See Also: [Class IMenu](#), [Class IMenuBarContext](#), [Class IQuadMenuContext](#), [Class Interface](#).

class IMenuManager

### Description:

This class is available in release 4.0 and later only.

To get an interface for calling the methods of this class use

**Interface::GetMenuManager()**.

### Methods:

public:

### Prototype:

```
virtual bool RegisterMenu(IMenu* pMenu, DWORD flags = 0) = 0;
```

### Remarks:

This method allows you to add a menu to the manager.

### Parameters:

**IMenu\* pMenu**

Points to the menu to register.

**DWORD flags = 0**

Not used.

### Return Value:

Returns false if the menu is already registered; true if not.

### Prototype:

```
virtual bool UnRegisterMenu(IMenu* pMenu) = 0;
```

### Remarks:

This method allows you to remove a menu form the mananger.

### Parameters:

**IMenu\* pMenu**

Points to the menu to unregister.

**Return Value:**

FALSE if the menu was not registered, TRUE if successfully unregistered.

**Prototype:**

```
virtual IMenu* FindMenu(TCHAR* pTitle) = 0;
```

**Remarks:**

This method will return a pointer to a menu based on its name.

**Parameters:**

**TCHAR\* pTitle**

The name of the menu to return.

**Return Value:**

A pointer to the menu or NULL if the menu wasn't found.

**Prototype:**

```
virtual IQuadMenu* FindQuadMenu(TCHAR* pTitle) = 0;
```

**Remarks:**

This method will return a pointer to a quad menu based on its name.

**Parameters:**

**TCHAR\* pTitle**

The name of the menu to return.

**Return Value:**

A pointer to the quad menu or NULL if the menu wasn't found.

**Prototype:**

```
virtual bool RegisterMenuBarContext(MenuContextId contextId,  
TCHAR* pName) = 0;
```

**Remarks:**

This method allows you to register a new menu bar context

**Parameters:**

**MenuContextId contextId**

The menu context ID.

**TCHAR\* pName**

The name of the menu bar.

**Return Value:**

TRUE if the new menu is registered, FALSE if the menu was already registered.

**Prototype:**

```
virtual bool RegisterQuadMenuContext(MenuContextId  
contextId, TCHAR* pName) = 0;
```

**Remarks:**

This method allows you to register a new quad menu context.

**Parameters:**

**MenuContextId contextId**

The menu context ID.

**TCHAR\* pName**

The name of the quad menu.

**Return Value:**

TRUE if the new quad menu is registered, FALSE if the quad menu was already registered.

**Prototype:**

```
virtual int NumContexts() = 0;
```

**Remarks:**

This method returns the number of contexts registered.

**Prototype:**

```
virtual IMenuContext* GetContextByIndex(int index) = 0;
```

**Remarks:**

This method returns a pointer to a menu context by the specified index.

**Parameters:**

**int index**

The index of the menu context to retrieve.

**Prototype:**

```
virtual IMenuContext* GetContext(MenuContextId contextId) = 0;
```

**Remarks:**

This method returns a pointer to a menu context by the specified menu context ID. This method returns NULL if the context does not exist.

**Parameters:**

**MenuContextId contextId**

The menu context ID.

**Prototype:**

```
virtual void UpdateMenuBar() = 0;
```

**Remarks:**

This method can be called to update 3ds max' main menu bar after adding sub-menu's or menu items.

**Prototype:**

```
virtual BOOL LoadMenuFile(TCHAR* pMenuFile) = 0;
```

**Remarks:**

This method allows you to load a menu file from disk and automatically update the UI accordingly.

**Parameters:**

**TCHAR\* pMenuFile**

The path and filename of the menu file to load.

**Return Value:**

TRUE if the menu file was loaded, otherwise FALSE.

**Prototype:**

```
virtual BOOL SaveMenuFile(TCHAR* pMenuFile) = 0;
```

**Remarks:**

This method allows you to save a menu file to disk.

**Parameters:**

## **TCHAR\* pMenuFile**

The path and filename of the menu file to save.

### **Return Value:**

TRUE if the menu file was saved, otherwise FALSE.

### **Prototype:**

```
virtual TCHAR* GetMenuFile() = 0;
```

### **Remarks:**

This method returns the file name of the currently loaded and active menu file.

### **Prototype:**

```
virtual BOOL SetMainMenuBar(IMenu* pMenu) = 0;
```

### **Remarks:**

This method allows you to set the main menu bar.

### **Parameters:**

**IMenu\* pMenu**

A pointer to the menu you wish to set as the main menu bar.

### **Return Value:**

TRUE if it was set successfully.

### **Prototype:**

```
virtual IMenu* GetMainMenuBar() = 0;
```

### **Remarks:**

This method returns a pointer to the main menu bar.

### **Prototype:**

```
virtual BOOL  
SetViewportRightClickMenu(IQuadMenuContext::RightClickCont  
context, IQuadMenu* pQuadMenu) = 0;
```

### **Remarks:**

This method allows you to set the viewport right-click menu to the specified quad menu.

**Parameters:**

**IQuadMenuContext::RightClickContext context**

See the [List of Right-Click Contexts](#).

**IQuadMenu\* pQuadMenu**

A pointer to the quad menu you wish to set.

**Return Value:**

TRUE if it was set successfully.

**Prototype:**

**virtual IQuadMenu\***

**GetViewportRightClickMenu(IQuadMenuContext::RightClickContext context) = 0;**

**Remarks:**

This method returns a pointer to the current viewport right-click quad menu.

**Parameters:**

**IQuadMenuContext::RightClickContext context**

See the [List of Right-Click Contexts](#)..

**Prototype:**

**virtual bool GetShowAllQuads(IQuadMenu\* pQuadMenu) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method checks if the "Show All Quads" flag is set for a specific QuadMenu and will return TRUE if the flag is set or FALSE if the flag is not set.

**Parameters:**

**IQuadMenu\* pQuadMenu**

A pointer to the QuadMenu you wish to check the flag for.

**Prototype:**

**virtual void SetShowAllQuads(IQuadMenu\* pQuadMenu, bool showAll) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method sets the "Show All Quads" flag for a specific QuadMenu.

**Parameters:****IQuadMenu\* pQuadMenu**

A pointer to the QuadMenu you wish to set the flag for.

**bool showAll**

TRUE to set the flag to on, FALSE to set the flag off.

**Prototype:**

```
virtual TCHAR* GetQuadMenuName(IQuadMenu*  
pQuadMenu) = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the name given to a specific QuadMenu as a string.

**Parameters:****IQuadMenu\* pQuadMenu**

A pointer to the QuadMenu for which you wish to retrieve the name.

**Prototype:**

```
virtual void SetQuadMenuName(IQuadMenu* pQuadMenu,  
TCHAR* pName) = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the name of a specific QuadMenu.

**Parameters:****IQuadMenu\* pQuadMenu**

A pointer to the QuadMenu for which you wish to set the name.

**TCHAR\* pName**

The string containing the name for the QuadMenu.

## Class IColorManager

See Also: [Class FPStaticInterface Class Point3](#), [COLORREF](#), [List of Standard Color IDs](#), [Generate a Class\\_ID](#), [Class GUP](#), [Getting and Setting User Preferences](#).

```
class IColorManager : public FPStaticInterface
```

### Description:

This class is available in release 4.0 and later only.

This class is an interface to the Color Manager. Within 3ds max using the Customize pull down menu / Customize User Interface choice / Colors tab a user is able to alter the colors used for various UI elements. They can change the saturation, value and transparency of elements, and load and save color schemes.

Using the methods of this class developers can do the same (the 3ds max color manager uses this class internally). Developers can add their own named custom colors. Developers wanting to do this need to pick a random 32-bit **ColorId** to identify it. (Note: **typedef DWORD ColorId;**). The Class\_ID generator may be used for this where only one of the two DWORDS is used. See [Class Class\\_ID](#) for more details. The low integer ColorIds are reserved for 3ds max internal use. These colors should be registered on startup, so a Global Utility Plug-In (GUP) is the best way to handle this. See [Class GUP](#) for details. Methods that are marked as internal should not be used.

**The following global functions are not part of this class but are available for use:**

### Function:

```
IColorManager* GetColorManager();
```

### Remarks:

This global function is available in release 4.0 and later only.

Returns a pointer to the color manager with which you can call the methods of this class.

Note the following #define which may be used to simplify coding a bit:

```
#define ColorMan() (GetColorManager())
```

### Methods:

public:

**Prototype:**

**virtual bool UseStandardWindowsColors() = 0;**

**Remarks:**

Returns true if the standard windows colors are used and false if custom colors are used.

**Prototype:**

**virtual void SetUseStandardWindowsColors(bool useStandardColors) = 0;**

**Remarks:**

Sets whether standard windows colors are used or not. This allows the developer to tell the system to use standard windows colors, instead of the custom colors. This only affects calls to **CustSysColor()** and not **GetColor()**.

**Parameters:**

**bool useStandardColors**

Pass true to use the standard windows color and false to use the custom colors.

**Prototype:**

**virtual bool RegisterColor(ColorId id, TCHAR\* pName, TCHAR\* pCategory, COLORREF defaultValue) = 0;**

**Remarks:**

This method registers a new color with the system. For plug-in developers this should be done at startup using a Global Utility Plug-in which calls this method. See the Description section above for details. If developers want to add a color of their own, they need to pick a random 32-bit integer ColorId to identify it.

**Parameters:**

**ColorId id**

The ID of the color to register. This should be generated by the developer using a single DWORD from the output of the Class\_ID program. See [Class ID](#) for more details.

**TCHAR\* pName**

The name for the color.

**TCHAR\* pCategory**

The category for the color. If the name passed matches one of the existing 3ds max categories the color will be placed in there, otherwise a new one will be created.

**COLORREF defaultValue**

The default value for the color. This is the value that the color will be reset to when a 3ds max user presses "Reset" in the color customization dialog. See [COLORREF](#).

**Return Value:**

Returns false if the color is already registered; otherwise true.

**Prototype:**

```
virtual BOOL LoadColorFile(TCHAR* pFileName) = 0;
```

**Remarks:**

This method will load the specified color file from the current UI directory.

**Parameters:****TCHAR\* pFileName**

The filename of the color file to load.

**Return Value:**

TRUE if the load was successful, otherwise FALSE.

**Prototype:**

```
virtual BOOL SaveColorFile(TCHAR* pFileName) = 0;
```

**Remarks:**

This method will save the specified color file from the current UI directory.

**Parameters:****TCHAR\* pFileName**

The filename of the color file to save.

**Return Value:**

TRUE if the save process was successful, otherwise FALSE.

**Prototype:**

**virtual TCHAR\* GetColorFile() = 0;**

**Remarks:**

This method returns the file name of the current color file.

**Prototype:**

**virtual bool SetColor(ColorId id, COLORREF color) = 0;**

**Remarks:**

Sets the color value of the previously registered color whose ID is passed.

**Parameters:****ColorId id**

Specifies which color to set.

**COLORREF color**

The color value to set. See [COLORREF](#).

**Return Value:**

Returns true if the color was set and false if the id passed could not be found.

**Prototype:**

**virtual COLORREF GetColor(ColorId id) = 0;**

**Remarks:**

Returns the color value of the color whose ID is passed.

**Parameters:****ColorId id**

Specifies which color to get.

**Return Value:**

The color is returned or black (RGB(0,0,0)) if the ColorId passed was not found.

**Prototype:**

**virtual Point3 GetColorAsPoint3(ColorId id) = 0;**

**Remarks:**

This method returns the color associated with a specified color ID as a Point3.

**Parameters:**

**ColorId id**

The ID of the color you wish to get.

**Prototype:**

**virtual HBRUSH GetBrush(ColorId id) = 0;**

**Remarks:**

Returns a handle to the brush for the color whose id is specified. NULL is returned if the id passed is not found. Note that the color manager does resource management for brushes, so the value returned should not be deleted.

**Parameters:**

**ColorId id**

The color whose brush handle is returned.

**Prototype:**

**virtual TCHAR\* GetName(ColorId id) = 0;**

**Remarks:**

Returns the name of the color whose ID is passed.

**Parameters:**

**ColorId id**

The ID of the color.

**Prototype:**

**virtual TCHAR\* GetCategory(ColorId id) = 0;**

**Remarks:**

Returns the category string of the color whose ID is passed.

**Parameters:**

**ColorId id**

The ID of the color.

**Prototype:**

**virtual COLORREF CustSysColor(int which) = 0;**

**Remarks:**

This method takes the specified windows color definition (for example **COLOR\_BTNHIGHLIGHT**) and returns the 3ds max customized version of that color. Anyone writing a plug-in with custom windows code should use this call (and **CustSysColorBrush()** below) instead of the Win32 **GetSysColor()** and **GetSysColorBrush()** if they want to participate in the 3ds max custom color scheme.

Note the following #define which may be used to simplify coding:

```
#define GetCustSysColor(which) (ColorMan()->CustSysColor(which))
```

**Parameters:****int which**

Specifies the windows color definition. See [List of Standard Color IDs](#). For a full list of windows color definitions, please refer to the Win32 API, in particular the methods **GetSysColor()** and **SetSysColor()**.

**Prototype:**

```
virtual HBRUSH CustSysColorBrush(int which) = 0;
```

**Remarks:**

This method returns a handle identifying a logical brush that corresponds to the specified color index. Note that the color manager does resource management for brushes, so the value returned should not be deleted.

**Parameters:****int which**

Specifies the windows color definition. See [List of Standard Color IDs](#). For a full list of windows color definitions, please refer to the Win32 API, in particular the methods **GetSysColor()** and **SetSysColor()**.

Note the following #define which may be used to simplify coding a bit:

```
#define GetCustSysColorBrush(which) (ColorMan()->CustSysColorBrush(which))
```

**Prototype:**

```
virtual Point3 GetOldUIColor(int which) = 0;
```

**Remarks:**

Returns the specified color value for drawing various items in the viewports. This is the same as the previous GetUIColor() function.

**Parameters:****int which**

Specifies which color to retrieve. See [List of Viewport Drawing Color Indices](#).

**Return Value:**

The color as a Point3.

**Prototype:**

```
virtual COLORREF GetOldUIColorCOLORREF(int which) = 0;
```

**Remarks:**

This method returns the color associated with a user interface color as a COLORREF.

**Parameters:****int which**

The UI color index (see gfx.h).

**Prototype:**

```
virtual void SetOldUIColor(int which, Point3 *clr) = 0;
```

**Remarks:**

Sets the specified color value for drawing various items in the viewports. This is the same as the previous SetUIColor() function.

**Parameters:****int which**

Specifies which color to retrieve. See [List of Viewport Drawing Color Indices](#).

**Point3 \*clr**

Points to the color value to set.

**Prototype:**

```
virtual Point3 GetOldDefaultUIColor(int which) = 0;
```

**Remarks:**

Returns the default color used for drawing various items in the 3ds max user interface. The values returned are not affected by the user's color selections or those set by SetUIColor(). This is the same as the previous GetDefaultUIColor() function.

**Parameters:**

**int which**

Specifies which color to retrieve. See [List of Viewport Drawing Color Indices](#).

**Prototype:**

**virtual float GetIconColorScale(IconType type, IconColorScale which) = 0;**

**Remarks:**

Returns a floating point value (in the range 0.0f to 1.0f) that is one of the scale factors applied to the specified icon type. These scale values used to do image processing on the icons at start-up time.

**Parameters:**

**IconType type**

The icon type. One of the following values:

**kDisabledIcon**

The disabled icons.

**kEnabledIcon**

The enabled icons.

**IconColorScale which**

The icon color scale. One of the following values:

**kSaturationScale**

The saturation scale.

**kValueScale**

The value scale.

**kAlphaScale**

The alpha scale.

**Prototype:**

**virtual void SetIconColorScale(IconType type, IconColorScale**

**which, float value) = 0;**

**Remarks:**

Sets the specified scale factor for the icon type passed. The color manager maintains the values for the 3ds max icon image processing system. Developers can set values to scale the saturation, value and transparency for enabled and disabled icon images using this method.

**Parameters:**

**IconType type**

The icon type. One of the following values:

**kDisabledIcon**

The disabled icons.

**kEnabledIcon**

The enabled icons.

**IconColorScale which**

The icon color scale. One of the following values:

**kSaturationScale**

The saturation scale.

**kValueScale**

The value scale.

**kAlphaScale**

The alpha scale.

**float value**

The value to set (in the range 0.0f to 1.0f).

**Prototype:**

**virtual bool GetIconColorInvert(IconType type) = 0;**

**Remarks:**

Returns true if the invert flag is set for the specified icon type and false if not set.

**Parameters:**

**IconType type**

The icon type. One of the following values:

**kDisabledIcon**

The disabled icons.

**kEnabledIcon**

The enabled icons.

**Prototype:**

**virtual void SetIconColorInvert(IconType type, bool value) = 0;**

**Remarks:**

Sets the invert flag for the specified icon type to on or off.

**Parameters:**

**IconType type**

The icon type. One of the following values:

**kDisabledIcon**

The disabled icons.

**kEnabledIcon**

The enabled icons.

**bool value**

Pass true for inverted; false for not inverted.

**Prototype:**

**virtual BOOL SetIconFolder(TCHAR\* pFolder) = 0;**

**Remarks:**

This method takes the name of a folder that must be in 3ds max "UI" folder. If the folder exists, then it sets 3ds max icon folder to point to it, and redraws the UI with those new icons. Warning: All of the 3ds max standard icon BMP files must exist in that folder. If any of the standard files are missing, icons will appear blank in the UI. All the icons files needed live in the **UIIcons** folder, which is the default icon folder.

**Parameters:**

**TCHAR\* pFolder**

The icon folder to set.

**Prototype:**

**virtual void InitSystemColors() = 0;**

**Remarks:**

This method is used internally to initialize the colors used by the system to their default values. This should not be called by third party developers.

**Prototype:**

**virtual IColorManager\* Copy() = 0;**

**Remarks:**

This method is used internally. It makes a copy of the color database that the UI changes, and then it copies it back to the original to commit the changes. Third party developers won't need to call this.

**Prototype:**

**virtual TCHAR\* GetFileName() = 0;**

**Remarks:**

Returns the file name of the currently loaded color file.

**Prototype:**

**virtual COLORREF GetDefaultColor(ColorId id) = 0;**

**Remarks:**

Returns the default color for the specified ID. The default color is the value passed as **defaultValue** in **RegisterColor()**, regardless if a **SetColor()** has been done subsequently. This is used by the UI when the user presses "Reset" to reset a color to its default value.

**Parameters:**

**ColorId id**

The ID of the color.

**Prototype:**

**virtual COLORREF GetOldUIColorCOLORREF(int which) = 0;**

**Remarks:**

Returns the specified color value for drawing various items in the viewports as

a COLORREF.

**Parameters:**

**int which**

Specifies which color to retrieve. See [List of Viewport Drawing Color Indices](#).

**Prototype:**

**virtual void RepaintUI(RepaintType type) = 0;**

**Remarks:**

This method allows you to issue a repaint of the user interface.

**Parameters:**

**RepaintType type**

The type of repaint you wish to issue; **kRepaintAll**, **kRepaintTrackBar**, **kRepaintTimeBar**.

**The following global functions are used internally and should not be called by plug-in developers**

**IColorManager\* CreateColorManager(TCHAR\* pDefaultColorFile);**

**void DeleteColorManager(IColorManager\* pColorMan);**

**void SaveColors();**

## Class IDragAndDropMgr

See Also: [Class FPStaticInterface](#), [Class DragAndDropHandler](#), [Class URLTab](#)

class IDragAndDropMgr : public FPStaticInterface

### Description:

This class is available in release 4.0 and later only.

The Drag and Drop system is managed through a Core FP interface (DND\_MGR\_INTERFACE), defined by this class. It provides control over the DnD system, manages handler registration and exposes some useful utility methods for downloading URL's, simulating drops, etc.

The DragAndDropMgr supports multiple DragAndDropHandlers registered against a single HWND window. This is to allow new components and 3rd-party developers to incrementally add handlers for new dropTypes to common windows such as viewports that the default handlers don't know how to handle.

The **IDragAndDropMgr::EnableDandD(HWND hwnd, BOOL flag, DragAndDropHandler\* handler)** method can be called multiple times on the same window with different handler instances. The DnDMgr keeps track of all the DragAndDropHandlers on each window and will call their DnD event methods as needed, in order of registration, until one of them returns **S\_OK**. For example, on a viewport, the DefaultDragAndDropHandler is registered by default. The event methods (such as DragEnter, DragOver, Drop, etc.) return **E\_FAIL** if the dropping IDataObject or DropType is not recognized by them. Then, if a new component registers its own handler to deal with some new droptype it adds, the default handler will fail to recognize the new droptype and so its handler will be called to process the new drop type.

Note that this means DragAndDropHandler event methods must correctly return **S\_OK** or **E\_FAIL** depending on whether they handle the IDataObject or DropType currently dropping, so that the appropriate handler is found and called by the DnDMgr.

For an example, please refer to

**\\MAXSDK\\SAMPLES\\HOWTO\\DND\_TEST.**

### Function:

**IDragAndDropMgr\* GetDragAndDropMgr();**

### Remarks:

This function, which is not part of the class, allows you to retrieve a pointer to the Drag and Drop Manager interface. This will return a pointer to the DnD manager interface from the core interface.

**Methods:**

public:

**Prototype:**

**virtual void EnableDandD(BOOL flag)=0;**

**Remarks:**

This method allows you to globally enable or disable the DnD interface.

**Parameters:**

**BOOL flag**

TRUE to enable, FALSE to disable.

**Prototype:**

**virtual BOOL IsEnabled()=0;;**

**Remarks:**

This method returns TRUE if the global DnD interface is enabled, otherwise FALSE.

**Prototype:**

**virtual BOOL EnableDandD(HWND hwnd, BOOL flag,  
DragAndDropHandler\* handler = NULL)=0;;**

**Remarks:**

This method allows you to enable DnD for a given window (and its children). If no custom DragAndDropHandler is supplied, a default one is used that will accept dropped scene files for opening and scripts for running.

**Parameters:**

**HWND hwnd**

A handle to the window you wish to enable or disable DnD for.

**BOOL flag**

TRUE to enable, FALSE to disable.

### **DragAndDropHandler\* handler = NULL**

A pointer to a custom DnD handler, or NULL to accept a default one.

### **Return Value:**

TRUE if the method was successful, otherwise FALSE.

### **Prototype:**

**virtual BOOL DropPackage(HWND hwnd, POINT& point, URLTab& package)=0;**

### **Remarks:**

This method allows the simulation of a package of files into a window at a given point. A package of files, specified as a list of URL strings is the common form of **DropType** data from **iDrop** sources and files dragged from the Windows desktop. The entire package is downloaded, as needed, but only the first file in the list is actually dropped into 3ds max. The other files in the package are presumed to be support files, such as texmaps or xref sources, for the main drop file. After the drop, the URL strings in the URLTab are converted to fully-specified path names to local file copies, if any had to be downloaded from the web.

### **Parameters:**

#### **HWND hwnd**

A handle to the window. If this is set to NULL, the default 3ds max window is used.

#### **POINT& point**

The point at which to drop.

#### **URLTab& package**

A reference to the local copies of the URL strings.

### **Return Value:**

TRUE if the drop was successful, otherwise FALSE.

### **Prototype:**

**virtual BOOL DownloadPackage(URLTab& package, TCHAR\* directory, HWND hwnd = NULL, bool showProgress = false)=0;**

### **Remarks:**

This method serves as a utility function that can be used to download a package of URLs to the specified directory. If the **hwnd** argument is supplied, any progress or other messages are centered over that window.

**Parameters:**

**URLTab& package**

A reference to the local copies of the URL strings.

**TCHAR\* directory**

The directory path string to download to.

**HWND hwnd = NULL**

A handle to the window. If this is set to NULL, the default window is used.

**bool showProgress = false**

The download progress dialog can be displayed by passing true.

**Return Value:**

TRUE if the download was successful, otherwise FALSE.

**Prototype:**

**virtual TCHAR\* GetDownloadDirectory()=0;**

**Remarks:**

This method returns the fully-specified path to the directory in which package drops are downloaded.

**Prototype:**

**virtual int NumHandlers(HWND hwnd)=0;**

**Remarks:**

This method returns the number of handlers associated with the given window.

**Parameters:**

**HWND hwnd = NULL**

A handle to the window.

**Prototype:**

**virtual DragAndDropHandler\* GetHandler(HWND hwnd, int i)=0;**

**Remarks:**

This method returns a pointer to a specified DnD handler of a specified window.

**Parameters:**

**HWND hwnd = NULL**

A handle to the window.

**int i**

The I-th handler.

**Prototype:**

**virtual bool DownloadUrlToDisk(HWND hwnd, TCHAR\* url, TCHAR\* fileName, DWORD dlgflags=0)=0;**

**Remarks:**

This method allows you to download the file referenced by the URL to disk.

**Parameters:**

**HWND hwnd = NULL**

A handle to the window.

**TCHAR\* url**

The URL string of the file to download.

**TCHAR\* fileName**

The filename string of the URL to store on disk.

**DWORD dlgflags=0**

Additional controls to the download behavior. Currently only one flag is supported, **DOWNLOADDLG\_NOPLACE**, which hides an option in the progress dialog that allows the user to place (move) a dropped object immediately after being dropped.

**Return Value:**

TRUE if the download was successful, otherwise FALSE.

**Prototype:**

**virtual INode\* ImportContextNode()=0;**

**Remarks:**

This method returns a pointer to the import context node.

## Class IParamWireMgr

See Also: [Class FPStaticInterface](#), [Class Control](#), [Class ReferenceTarget](#)

```
class IParamWireMgr : public FPStaticInterface
```

### Description:

This class is available in release 4.0 and later only.

This class represents the interface that provides general access to the parameter wiring functions. You can obtain a pointer to the Parameter Wire Manager interface using; **IParamWireMgr\* GetParamWireMgr()**. This macro will return

```
(IParamWireMgr*)GetCOREInterface(PARAMWIRE_MGR_INTEI
```

All methods of this class are Implemented by the System.

### Methods:

public:

### Prototype:

```
virtual void StartParamWire()=0;
```

### Remarks:

This method will launch the parameter wiring UI mode.

### Prototype:

```
virtual void OpenEditor()=0;
```

### Remarks:

This method will open up the parameter wiring dialog on the selected objects.

### Prototype:

```
virtual void EditParams(ReferenceTarget* leftParent, int  
leftSubNum, ReferenceTarget* rightParent, int rightSubNum)=0;
```

### Remarks:

This method allows you to edit the left- and right-hand parameters and opens the parameter wiring dialog using the provided parameters.

### Parameters:

**ReferenceTarget\* leftParent**

A pointer to the left-hand reference target.

**int leftSubNum**

The sub-animatable of the left-hand reference target.

**ReferenceTarget\* rightParent**

A pointer to the right-hand reference target.

**int rightSubNum**

The sub-animatable of the right-hand reference target.

**Prototype:**

```
virtual void EditControllers(Control* leftWire, Control*  
rightWire)=0;
```

**Remarks:**

This method allows you to setup the two controllers for the left- and right-hand to edit.

**Parameters:****Control\* leftWire**

A pointer to the controller for the left-hand wire.

**Control\* rightWire**

A pointer to the controller for the right-hand wire.

**Prototype:**

```
virtual void EditController(Control* wire)=0;
```

**Remarks:**

This method is identical to the **EditControllers()** but accepts a single wire controller for the left-hand. This method effectively calls **EditControllers(wire, NULL)**.

**Parameters:****Control\* wire**

A pointer to the controller being edited.

**Prototype:**

```
virtual bool Connect(ReferenceTarget* fromParent, int  
fromSubNum, ReferenceTarget* toParent, int toSubNum,  
TCHAR* toExpr)=0;
```

**Remarks:**

This method allows you to set up a one-way wire.

**Parameters:**

**ReferenceTarget\* fromParent**

A pointer to the reference target to wire from.

**int fromSubNum**

The sub-animatable to wire from.

**ReferenceTarget\* toParent**

A pointer to the reference target to wire to.

**int toSubNum**

The sub-animatable to wire to.

**TCHAR\* toExpr**

A string containing the expression on the "to wire".

**Return Value:**

TRUE if the connection can be made, otherwise FALSE.

**Prototype:**

```
virtual bool Connect2Way(ReferenceTarget* leftParent, int  
leftSubNum, ReferenceTarget* rightParent, int rightSubNum,  
TCHAR* leftExpr, TCHAR* rightExpr=NULL)=0;
```

**Remarks:**

This method allows you to set up a two-way wire.

**Parameters:**

**ReferenceTarget\* leftParent**

A pointer to the left-hand reference target.

**int leftSubNum**

The sub-animatable of the left-hand reference target.

**ReferenceTarget\* rightParent**

A pointer to the right-hand reference target.

**int rightSubNum**

The sub-animatable of the right-hand reference target.

**TCHAR\* leftExpr**

A string containing the expression for the left-hand target.

**TCHAR\* rightExpr = NULL**

A string containing the expression for the right-hand target.

**Return Value:**

TRUE if the connection can be made, otherwise FALSE.

**Prototype:**

```
virtual bool Disconnect(Control* wireController)=0;
```

**Remarks:**

This method allows you to disconnect a one-way wire.

**Parameters:****Control\* wireController**

A pointer to the wire controller you wish to disconnect.

**Return Value:**

TRUE if the disconnect was successful, otherwise FALSE.

**Prototype:**

```
virtual bool Disconnect2Way(Control* wireController1, Control*  
wireController2)=0;
```

**Remarks:**

This method allows you to disconnect a two-way wire.

**Parameters:****Control\* wireController1**

A pointer to the first wire controller you wish to disconnect.

**Control\* wireController2**

A pointer to the second wire controller you wish to disconnect.

**Return Value:**

TRUE if the disconnect was successful, otherwise FALSE.

## Class NodeDisplayCallback

See Also: [Class InterfaceServer](#), [Class INodeDisplayControl](#), [Class INode](#), [Class ViewExp](#), [Class IPoint2](#).

```
class NodeDisplayCallback : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

A callback to allow plug-ins that aren't actually objects (such as utilities) to control a Node's display.

This class enables you to display extra information on top of a node in a viewport. Once activated, a plug-in will control the display (on/off) of a node's world space representation as well as add data in a viewport on a per node basis. This approach allows you to replace the drawing code of every node without adding modifiers on top of each of them.

### Methods:

public:

### Prototype:

```
virtual void StartDisplay(TimeValue t, ViewExp *vpt, int flags)=0;
```

### Remarks:

This method is called just before 3ds max draws the nodes in the scene.

### Parameters:

#### **TimeValue t**

The time at which the nodes are being drawn.

#### **ViewExp \*vpt**

Points to an interface for the viewport the node is being drawn in.

#### **int flags**

These flags are used internally.

### Prototype:

```
virtual void EndDisplay(TimeValue t, ViewExp *vpt, int flags)=0;
```

### Remarks:

This method is called just after 3ds max draws the nodes in the scene.

**Parameters:****TimeValue t**

The time at which the nodes were drawn.

**ViewExp \*vpt**

Points to an interface for the viewport the node is being drawn in.

**int flags**

These flags are used internally.

**Prototype:**

```
virtual bool Display(TimeValue t, ViewExp *vpt, int flags, INode *node)=0;
```

**Remarks:**

This method is called for every node to allow it display itself.

**Parameters:****TimeValue t**

The time at which the node is to be drawn.

**ViewExp \*vpt**

Points to an interface for the viewport in which the node is being drawn in.

**int flags**

The display flags, which are;

**USE\_DAMAGE\_RECT**

If this flag is set, only the damaged area needs to be displayed. The damaged rectangle may be retrieved using **INode::GetDamagedRect()**. See [Class INode](#).

**DISP\_SHOWSUBOBJECT**

This indicates if an item should display its sub-object selection state. The system will set this flag if the item is selected, the user is in the modify branch, and the item is in sub-object selection mode.

**INode \*node**

Points to the node being drawn.

**Return Value:**

TRUE if displayed, otherwise FALSE.

**Prototype:**

```
virtual bool SuspendObjectDisplay(TimeValue t, INode *node)=0;
```

**Remarks:**

This method is called to determine if the node mesh should be displayed. It should return true; otherwise return false.

**Parameters:****TimeValue t**

The time at which to check if the node should be displayed.

**INode \*node**

The node to check.

**Prototype:**

```
virtual void AddNodeCallbackBox(TimeValue t, INode *node,  
ViewExp *vpt, Box3& box)=0;
```

**Remarks:**

This method will ask the callback to participate in the bounding box calculation.

**Parameters:****TimeValue t**

The time at which to calculate the bounding box.

**INode \*node**

The node to calculate the bounding box for.

**ViewExp \*vpt**

Points to an interface for the viewport in which the node is being drawn in.

**Box3& box**

A reference to the bounding box.

**Prototype:**

```
virtual bool HitTest(TimeValue t, INode *node, int type, int  
crossing, int flags, IPoint2 *p, ViewExp* vpt)=0;
```

**Remarks:**

This method hit tests the callback's mesh.

**Parameters:****TimeValue t**

The time at which to hit test.

**INode \*node**

A pointer to the node to test.

**int type**

The type of hit testing to perform. See [Hit Test Types](#) for details.

**int crossing**

The state of the crossing setting. If TRUE crossing selection is on.

**int flags**

The hit test flags. See [Hit Test Flags](#) for details.

**IPoint2 \*p**

The screen point to test.

**ViewExp\* vpt**

An interface pointer that may be used to call methods associated with the viewports.

**Return Value:**

TRUE if the item was hit, otherwise FALSE.

**Prototype:**

```
virtual void Activate()=0;
```

**Remarks:**

This method is called when the callback gets activated. It is up to the callback to invalidate the screen.

**Prototype:**

```
virtual void Deactivate()=0;
```

**Remarks:**

This method is called when the callback is deactivated.

**Prototype:**

```
virtual TSTR GetName() const = 0;
```

**Remarks:**

This method returns the name of the callback which is used for display in the menu. The user must delete the string returned.

## Class XTCTObject

See Also: [Class InterfaceServer](#), [Class Class\\_ID](#), [Class ModContext](#), [Class ObjectState](#), [Class INode](#), [Class Object](#), [Class Modifier](#), [Class GraphicsWindow](#), [Class FPInterface](#), [List of Channel Bits.](#), [Class XTCTContainer](#), [Class IXTCAccess](#)

```
class XTCTObject : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This is the base class for an Extension Channel plug-in. These plug-ins are used to allow a developer defined object to flow down the geometry pipeline. This class provides an interface to the extension object. This is the virtual base class that developers can derive objects from, that should be inserted into the extension channel of the object flowing up the stack. Extension Channels will expand the geometry pipeline by allowing one to add a custom object to the pipeline object that can flow down the pipeline. This object will get notified whenever something in the pipeline changes. For example, if you want to indicate when a certain object becomes invalid for export to their game engine, invalid skin-vertex assignments, bound patches etc. By inserting an Extension Channel Object (XTCTObject, for short) into the pipeline you can accomplish this, by constantly checking the structure of the object and displaying wrong faces/vertices etc. in the viewport.

You can specify which other channels it depends on using **DependsOn()**. The extension object has callback methods that get called before and after a modifier modifies a channel that the extension object depends on using **PreChanChangedNotify()** and **PostChanChangedNotify()**. The extension object can declare additional channels that it modifies using **ChannelsChanged()**, so that it can make any changes to the mesh before and after the modification by the modifier.

In general, the Extension Channel is a transient data structure that gets recreated on every pipeline evaluation. So the object that adds an extension channel to the modifier stack automatically makes it persistent. However, when the user collapses the stack, the user might want the Extension Channel to be preserved as well. In order to accomplish that, please refer to the [Class BaseObject](#) and the methods **NotifyPreCollapse()** and **NotifyPostCollapse()**. These methods

will be called by the collapse code. It will give the modifier or BaseObject, that adds an XTC object to the stack the possibility to apply a modifier, that inserts these XTC objects onto the stack after the collapse. Through this mechanism, the XTC will survive a stack collapse. The Pre and Post notifications will be called through a pipeline enumeration downstream (for more info see [Class GeomPipelineEnumProc](#)). Developers, who are collapsing the stack programmatically, have to call this method. In case this method is not called, the XTC objects will by default be copied as well, since they are part of the object in the wsCache. However, they won't survive a save/load operation. In addition to all this, XTC objects also have the possibility to display their data in the viewports. Any Extension Channel Object can disable the display of the object itself and take over the entire display itself, by returning true in the method **SuspendObjectDisplay()**.

Note: Modifiers which change the type of object that flows up the stack have to copy the Extension Channel from the old object into the new one using **CopyAdditionalChannels()** (e.g. the extrude modifier has to copy the XTC from the incoming spline to the Mesh, Patch or NURBS object).

Note: Compound objects have to merge the Extension Channel of the branched pipelines into the resulting pipeline. This is in general a simple copy of the Extension Channel Object into the new Extension Channel. When the CompoundObject evaluates its branches it would call **CopyAdditionalChannels(os->obj)**, so that the Extension Channels of the branches are copied over. In the **ConvertToType()** method it then has to copy the Extension Channels from itself to the converted object using **obj->CopyAdditionalChannels(this)**.

Also note that the Extension Channel itself is implemented in Class Object. This means, that it will be available for all pipeline objects that get implemented in 3ds max. For additional methods related to extension objects see the methods in Class Object -> [Extension Channel Access](#).

All methods of this class are implemented by the plug-in. Default implementations are shown.

## **Methods:**

public:

## **Prototype:**

**virtual Class\_ID ExtensionID()=0;**

**Remarks:**

This method returns the unique identifier for the object.

**Prototype:**

**virtual XTCTObject \*Clone()=0;**

**Remarks:**

This method is called to create a cloned copy of the object. The object should create a copy of itself and return a pointer to it.

**Prototype:**

**virtual ChannelMask DependsOn();**

**Remarks:**

This method returns a ChannelMask which specifies the channels that the XTCTObject depends on. If a modifier changes a channel that a XTCTObject depends on, its **PreChanChangedNotify()** and **PostChanChangedNotify()** methods will be called.

**Return Value:**

See the [List of Channel Bits](#).

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

**virtual ChannelMask ChannelsChanged();**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns a ChannelMask which specifies the channels that the extension object changes in the **PreChanChangedNotify()** and **PostChanChangedNotify()** methods.

**Return Value:**

See the [List of Channel Bits](#).

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual ChannelMask ChannelsUsed();
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns a ChannelMask which specifies the channels that the extension object uses in the **PreChanChangedNotify()** and **PostChanChangedNotify()** methods.

**Return Value:**

See the [List of Channel Bits](#).

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual int Display(TimeValue t, INode* inode, ViewExp *vpt, int flags, Object *pObj);
```

**Remarks:**

If an XTCObject wants to display itself in the viewport it can overwrite this method.

**Parameters:**

**TimeValue t**

The time at which the object is to be displayed.

**INode\* inode**

Points to the node for the object.

**ViewExp \*vpt**

Points to the viewport interface for the object.

**int flags**

See [List of Display Flags](#).

**Object \*pObj**

Points to the object that the extension object is a part of.

**Return Value:**

The return value is not currently used.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual void PreChanChangedNotify(TimeValue t, ModContext  
&mc, ObjectState* os, INode *node, Modifier *mod, bool  
bEndOfPipeline);
```

**Remarks:**

This method is called before a modifier is applied that changes a channel that the XTCTObject depends on.

**Parameters:**

**TimeValue t**

The time at which the channel will be modified.

**ModContext &mc**

The modifier context.

**ObjectState\* os**

The objectstate of the object.

**INode \*node**

A pointer to the node.

**Modifier \*mod**

A pointer to the modifier being applied.

**bool bEndOfPipeline**

TRUE to indicate that this is the last change before the wsCache.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void PostChanChangedNotify(TimeValue t, ModContext  
&mc, ObjectState* os, INode *node, Modifier *mod, bool  
bEndOfPipeline);
```

**Remarks:**

This method will be called after a modifier is applied that changes a channel that the XTC object depends on.

**Parameters:**

**TimeValue t**

The time at which the channel will be modified.

**ModContext &mc**

The modifier context.

**ObjectState\* os**

The objectstate of the object.

**Inode \*node**

A pointer to the node.

**Modifier \*mod**

A pointer to the modifier being applied.

**bool bEndOfPipeline**

TRUE to indicate that this is the last change before the wsCache.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BOOL SuspendObjectDisplay();
```

**Remarks:**

If the XTCObject returns TRUE from this method the object is not displayed in the viewport; if FALSE is returned the **Display()** method will be called to display the object.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void DeleteThis()=0;
```

**Remarks:**

This method is called to delete the extension object.

**Prototype:**

```
virtual void MaybeEnlargeViewportRect(GraphicsWindow *gw,  
Rect &rect);
```

**Remarks:**

This method allows the object to enlarge its viewport rectangle if it wants to. The system will call this method for all XTCObjects when calculating the viewport rectangle; the XTCObject can enlarge the rectangle if desired.

**Parameters:**

**GraphicsWindow \*gw**

Points to the GraphicsWindow instance associated with the viewport the object is displayed in.

**Rect &rect**

The viewport rectangle for the object which may be modified.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual bool RemoveXTCObjectOnMergeBranches(Object  
*obFrom, Object *obTo);
```

**Remarks:**

By default the existing XTCObjects will be deleted if a branch updates. In case the XTCObject wants to do more intelligent branching (not simply delete and add), it can return false from this method so that it can later (see **MergeXTCObject()** below) copy the data from this and other branches into an existing XTCObject.

**Parameters:**

**Object \*obFrom**

Points to the source object.

**Object \*obTo**

Points to the destination object.

**Return Value:**

Returns true if the object will be deleted; false to do more processing via MergeXTCObject.

**Default Implementation:**

```
{ return true; }
```

**Prototype:**

```
virtual bool MergeXTCObject(Object *obFrom, Object *obTo, int  
prio, int branchID);
```

**Remarks:**

The default implementation just adds the XTCObject to the to object. In case the XTCObject should do a more intelligent merge with already existing XTCObjects in the obTo, it has to overwrite this method.

**Parameters:**

**Object \*obFrom**

Points to the source object.

**Object \*obTo**

Points to the destination object.

**int prio**

The priority to set.

**int branchID**

The branch identifier to set.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Default Implementation:**

```
{ obTo->AddXTCObject(this,prio,branchID); return true;}
```

**Prototype:**

```
virtual bool RemoveXTCObjectOnBranchDeleted(Object  
*ComObj,int branchID, bool branchWillBeReordered);
```

**Remarks:**

In case a branch of a compound object is deleted the XTCObject will be asked if the XTCObject should be deleted as well. In case the XTCObject represents a merge of all branches the XTCObject might want to return false to this method and reassign itself to another branch, so that the merged information is not lost.

**Parameters:**

**Object \*ComObj**

A pointer to the compound object.

**int branchID**

The branch identifier to set.

**bool branchWillBeReordered**

TRUE if the branch should be reordered, otherwise FALSE.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Default Implementation:**

```
{ return true; }
```

## Class GeomPipelineEnumProc

See Also: [Class InterfaceServer](#), [Class INode](#), [Class Object](#), [Class Modifier](#), [Class IDerivedObject](#), [Class ModContext](#), [Class NotifyCollapseEnumProc](#).

```
class GeomPipelineEnumProc : public InterfaceServer
```

### Description:

This class and its associated global functions are available in release 4.0 and later only.

This is the callback object for the global geometry pipeline enumeration functions. The single **proc()** method of this class is called as the enumeration takes place.

The following functions are not methods of this class but are available for use with it to begin the enumeration:

These all start a pipeline enumeration down the pipeline towards the baseobject and over the baseobjects' branches in case it is a compound object. A pipeline enumeration can be started from a Node, an Object or from a Modifier.

### Function:

```
int EnumGeomPipeline(GeomPipelineEnumProc *gpep, INode  
*start, bool includeEmptyDOs = false);
```

### Remarks:

The global function begins an enumeration of the geometry pipeline using the specified node.

### Parameters:

**GeomPipelineEnumProc \*gpep**

Points to the callback object to process the enumeration.

**INode \*start**

Points to the node to start the enumeration.

**bool includeEmptyDOs = false**

In case the flag includeEmptyDOs is declared as true, the proc will be called even for DerivedObjects, that don't contain any modifiers. In that case the object pointer will be NULL, the derObj pointer will contain the DerivedObject and the index will be -1.

### Return Value:

One of the following values:

**PIPE\_ENUM\_CONTINUE**

**PIPE\_ENUM\_STOP**

**Function:**

```
int EnumGeomPipeline(GeomPipelineEnumProc *gpep, Object  
*start, bool includeEmptyDOs = false);
```

**Remarks:**

The global function begins an enumeration of the geometry pipeline using the specified object.

**Parameters:**

**GeomPipelineEnumProc \*gpep**

Points to the callback object to process the enumeration.

**Object \*start**

Points to the object to start the enumeration.

**bool includeEmptyDOs = false**

In case the flag includeEmptyDOs is declared as true, the proc will be called even for DerivedObjects, that don't contain any modifiers. In that case the object pointer will be NULL, the derObj pointer will contain the DerivedObject and the index will be -1.

**Return Value:**

One of the following values:

**PIPE\_ENUM\_CONTINUE**

**PIPE\_ENUM\_STOP**

**Function:**

```
int EnumGeomPipeline(GeomPipelineEnumProc *gpep,  
IDerivedObject *start, int modIndex = 0, bool includeEmptyDOs  
= false);
```

**Remarks:**

The global function begins an enumeration of the geometry pipeline using the specified derived object and modifier index. The caller of this method has to provide the IDerivedObject the Modifier is applied to and the index of the

Modifier in the IDerivedObject. Developers can use the method **Modifier::GetIDerivedObject()** in order to get the IDerivedObject and the index, given a modifier and a ModContext.

**Parameters:**

**GeomPipelineEnumProc \*gpep**

Points to the callback object to process the enumeration.

**IDerivedObject \*start**

Points to the derived object to start the enumeration.

**int modIndex = 0**

The zero based index of the modifier in the derived object to start with.

**bool includeEmptyDOs = false**

In case the flag includeEmptyDOs is declared as true, the proc will be called even for DerivedObjects, that don't contain any modifiers. In that case the object pointer will be NULL, the derObj pointer will contain the DerivedObject and the index will be -1.

**Return Value:**

One of the following values:

**PIPE\_ENUM\_CONTINUE**

**PIPE\_ENUM\_STOP**

**Methods:**

public:

**Prototype:**

**virtual PipeEnumResult proc(ReferenceTarget \*object, IDerivedObject \*derObj, int index)=0;**

**Remarks:**

This is the callback procedure for pipeline enumeration. The ReferenceTarget passed to the proc can be a Node, Modifier or Object. In case it is a Modifier the parameter **derObj** contains the DerivedObject and the index is the index of this modifier in the DerivedObject. In all other cases **derObj** is NULL and index is 0.

**Parameters:**

**ReferenceTarget \*object**

Points to the item in the geometry pipeline. This can be a Node, Modifier or Object.

**IDerivedObject \*derObj**

If **object** above is a Modifier this points to the derived object.

**int index**

If **object** above is a Modifier this is the index of this modifier in the DerivedObject.

**Return Value:**

One of the following values which determines how the enumeration proceeds:

**PIPE\_ENUM\_CONTINUE**

Specifies to continue the enumeration.

**PIPE\_ENUM\_STOP**

Specifies to halt the enumeration.

## Class IFaceDataChannel

See Also: [Class IDataChannel](#), [Class IFaceDataChannelsEnumCallBack](#) , [Class IFaceDataMgr](#)

```
class IFaceDataChannel : public IDataChannel
```

### Description:

This class is available in release 4.0 and later only.

This class represents the face-data channel interface and as such is an abstraction of a collection of data objects that is associated with faces of 3ds max objects. 3ds max objects that have face-data channels call the methods of this interface when those faces change in some way. The data channels can then react to the changes to the faces. You can use the macro

**GetFaceDataChannelInterface(obj)** to obtain a pointer to this interface.

Currently in version 4.0 only Meshes support face-data channels.

### Methods:

public:

### Face Operations and Events

The face operations and events are called by the owner of face-data channels when its faces change in some way. It's up to the face-data channel to do whatever it wants to do on these notification methods.

### Prototype:

```
virtual BOOL FacesCreated(ULONG at, ULONG num) = 0;
```

### Remarks:

This method is called when **num** new faces are created at the index **at** in the object's list of faces.

### Parameters:

**ULONG at**

The index in the object's array of faces where the new faces are inserted.

**ULONG num**

The number of new faces which are created.

### Return Value:

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual BOOL FacesClonedAndAppended(BitArray& set) = 0;**

**Remarks:**

This method is called when the owner object has cloned some of its faces and appended these to its list of faces. The bits in the set array correspond to the cloned faces.

**Parameters:**

**BitArray& set**

The array of bits. Note that this array has as many bits as there are faces in the owner object.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual BOOL FacesDeleted(BitArray& set) = 0;**

**Remarks:**

This method is called when faces were deleted in the owner object. The bits in the set array correspond to the deleted faces.

**Parameters:**

**BitArray& set**

The array of bits. Note that this array has as many bits as there are faces in the owner object.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual BOOL FacesDeleted(ULONG from, ULONG num) = 0;**

**Remarks:**

This method is called when faces were deleted in the owner object. This method allows for a more efficient deletion of a range of data objects than the

previous BitArray based one.

**Parameters:**

**ULONG from**

The index in the object's array of faces. Faces starting at this index were deleted.

**ULONG num**

The number of faces that were deleted.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual void AllFacesDeleted() = 0;**

**Remarks:**

This method is called when all faces in the owner object are deleted

**Prototype:**

**virtual BOOL FaceCopied(ULONG from, ULONG to) = 0;**

**Remarks:**

This method is called when a face has been copied from index **from** in the owner object's array of faces to the face at index **to**.

**Parameters:**

**ULONG from**

The index of the source face.

**ULONG to**

The index of the destination face.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual BOOL FaceInterpolated(ULONG numSrc, ULONG\* srcFaces, float\* coeff, ULONG targetFace) = 0;**

**Remarks:**

This method is called when a new face has been created in the owner object based on data interpolated from other faces.

**Parameters:**

**ULONG numSrc**

The number of faces used in the interpolation.

**ULONG\* srcFaces**

The array of **numSrc** face indices into the owner object's face array. These faces were used when creating the new face.

**float\* coeff**

The array of **numSrc** coefficients used in the interpolation.

**ULONG targetFac**

The index in the owner object's array of faces of the newly created face.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Geometry pipeline (stack) methods**

These methods are called when the owner object is flowing up the pipeline (stack). They must be implemented to ensure that the face-data channel flows up the pipeline correctly. The owner object expects the face-data to do exactly what the names of these methods imply. These can be seen as commands that are given by the owner object to the face-data channel

**Prototype:**

**virtual IFaceDataChannel\* CreateChannel( ) = 0;**

**Remarks:**

This method will allocate an empty data-channel.

**Prototype:**

**virtual IFaceDataChannel\* CloneChannel( ) = 0;**

**Remarks:**

The data-channel needs to allocate a new instance of itself and fill it with copies of all data items it stores. Note: This method makes it more efficient to clone the whole data-channel.

**Prototype:**

```
virtual BOOL AppendChannel(const IFaceDataChannel*  
fromChan) = 0;
```

**Remarks:**

The data-channel needs to append the data objects in the **fromChan** to itself.

**Parameters:**

**const IFaceDataChannel\* fromChan**

The channel containing the data objects to append.

**Return Value:**

TRUE if successful, otherwise FALSE.

## Class IFaceDataMgr

See Also: [Class BaseInterface](#), [Class IDataChannel](#), [Class IFaceDataChannel](#)

```
class IFaceDataMgr : public BaseInterface
```

### Description:

This class is available in release 4.0 and later only.

This class represents an Interface for managing face-data channels. Objects that wish to have face-data channels should implement this interface. If this interface needs to be changed, a new one should be derived from it and changed (IFaceDataMgr2) and ensuring that objects support face-data implementations of both old and new interfaces.

A "naive" extension of Mesh and MNMesh with a GetInterface method are available but there's no support for interface lifetime management. The Mesh and MNMesh have full control over the lifetime of the face-data manager and as such, clients should not cache an IFaceDataMgr interface aquired from Mesh or MNMesh.

Please note that only **Meshes** supports the **IFaceDataMgr** interface. The **MNMesh** (and thus polygons) does not support it. As a consequence, if you write a modifier that applies face data to a meshes, and then the pipeline transforms this mesh into a poly or patch, the face data is lost at the top of the stack.

### Methods:

public:

#### Geometry pipeline (stack) methods

Modifiers and procedural objects should call these methods to add, remove, or retrieve a face-data channel on an object (mesh, patch, poly).

### Prototype:

```
virtual ULONG NumFaceDataChans() const = 0;;
```

### Remarks:

This method returns the number of face-data channels.

### Prototype:

**virtual IFaceDataChannel\* GetFaceDataChan(const Class\_ID& ID) const = 0;**

**Remarks:**

This method returns a pointer to the face-data channel.

**Parameters:**

**const Class\_ID& ID**

The class ID of the channel you wish to retrieve.

**Prototype:**

**virtual BOOL AddFaceDataChan(IFaceDataChannel\* pChan) = 0;**

**Remarks:**

This method adds a face-data channel to the object.

**Parameters:**

**IFaceDataChannel\* pChan**

A pointer to the face-data channel.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual BOOL RemoveFaceDataChan(const Class\_ID& ID) = 0;**

**Remarks:**

This method removes a face-data channel from the object.

**Parameters:**

**const Class\_ID& ID**

The class ID of the channel you wish to remove.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Geometry pipeline (stack) methods**

The system (3ds max) should call these methods to manage the face-data channels when the object flows up the stack

**Prototype:**

```
virtual BOOL AppendFaceDataChan(const IFaceDataChannel*  
pChan) = 0;
```

**Remarks:**

This method appends a face-data channel to the object.

**Parameters:**

```
const IFaceDataChannel* pChan  
The face-data channel to append.
```

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

```
virtual BOOL CopyFaceDataChans(const IFaceDataMgr* pFrom)  
= 0;
```

**Remarks:**

This method adds or appends face-data channels from the **from** object, to this object. If the channel already exists on this object, it's appended otherwise it gets added.

**Parameters:**

```
const IFaceDataMgr* pFrom  
The face-data channel to copy from.
```

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

```
virtual void RemoveAllFaceDataChans() = 0;
```

**Remarks:**

This method removes all face-data channels from this object.

**Prototype:**

```
virtual BOOL  
EnumFaceDataChans(IFaceDataChannelsEnumCallBack& cb,
```

**void\* pContext) const = 0;**

**Remarks:**

This method provides a mechanism for executing an operation for all face-data-channels on this object: For all face-data-channels calls **IFaceDataEnumCallBack::proc()** with a pointer to that face-data-channel and a context data

**Parameters:**

**IFaceDataChannelsEnumCallBack& cb**

A pointer to the face-data channel enumerator callback.

**void\* pContext**

A pointer to the context data.

**Return Value:**

FALSE if the callback returns FALSE for any of the face-data channels.

**Prototype:**

**virtual IOResult Save(ISave\* isave) = 0;**

**Remarks:**

Saves the face-data to the max file.

**Return Value:**

See Also: [List of IO Results](#).

**Prototype:**

**virtual IOResult Load(ILoad\* iload) = 0;**

**Remarks:**

Loads the face-data from the max file.

**Return Value:**

See Also: [List of IO Results](#).

**Prototype:**

**Interface\_ID GetID();**

**Remarks:**

This method returns the interface ID of the object.

**Default Implementation:**

```
{ return FACEDATAMGR_INTERFACE; }
```

## Class ObjectConverter

See Also: [Class InterfaceServer](#), [Class Class\\_ID](#), [Class Object](#), [Class TriObject](#), [Class PatchObject](#).

```
class ObjectConverter : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This virtual class is implemented by applications that want to supply a conversion method from one object type to another. A typical use would be to support conversion of a native 3ds max type (such as TriObject) to a plug-in's object type. There are a set of global functions that can be used with this class. These are documented at the bottom of the topic. One of these is called to register the ObjectConverter with the system.

Note that the registered object converters are called from the methods:

**Object::CanConvertToType** and **Object::ConvertToType**.

So for individual objects to support these, they'll need to add the line

```
if (Object::CanConvertToType(obtype)) return 1;
```

to the end of their CanConvertToType methods and

```
if (Object::CanConvertToType(obtype))
```

```
return Object::ConvertToType(t, obtype);
```

to the end of their ConvertToType methods.

### Methods:

public:

#### Prototype:

```
virtual Class_ID ConvertsFrom()=0;
```

#### Remarks:

This method returns the Class ID of the object this converter converts from.

#### Prototype:

```
virtual Class_ID ConvertsTo()=0;
```

#### Remarks:

This method returns the Class ID of the object this converter converts to.

**Prototype:**

```
virtual Object *Convert(Object *from)=0;
```

**Remarks:**

This method actually performs the conversion, creating and returning a new object with the class ID specified in **ConvertsTo()**.

**Parameters:**

```
Object *from
```

Points to the object to convert.

**Prototype:**

```
virtual void DeleteThis();
```

**Remarks:**

This should delete the ObjectConverter if necessary.

**Default Implementation:**

```
{}
```

**The following global functions are not part of this class but are available for use:**

**Function:**

```
bool RegisterObjectConverter(ObjectConverter *conv);
```

**Remarks:**

This global function is available in release 4.0 and later only.

Registers an object converter with the system.

**Parameters:**

```
ObjectConverter *conv
```

Points to the ObjectConverter instance to register.

**Return Value:**

Returns true if the converter could be added; false if not.

**Function:**

**int CanConvertTriObject(Class\_ID to);**

**Remarks:**

This global function is available in release 4.0 and later only.

Indicates if a TriObject can convert to the specified class ID.

Note: this actually checks if an Editable Mesh object can convert to the specified type, since an Editable Mesh is what you get when you call **CreateNewTriObject ()**.

This method may be used in an object's **CanConvertToType()** and **ConvertToType()** methods. If your object supports conversion to a TriObject, but doesn't support conversion to the given class ID, you can use this method to find out if TriObjects can be used as an "intermediary". If so, you can construct a temporary TriObject, convert it to the given class ID, and call the temporary TriObject's **DeleteThis()** method.

**Parameters:**

**Class\_ID to**

The Class ID to convert to.

**Return Value:**

Nonzero if the TriObject can be converted to the specified objec type; otherwise zero.

**Function:**

**int CanConvertPatchObject(Class\_ID to);**

**Remarks:**

This global function is available in release 4.0 and later only.

Indicates if a PatchObject can convert to the specified class ID.

**Parameters:**

**Class\_ID to**

The Class ID to convert to.

**Return Value:**

Nonzero if the PatchObject can be converted to the specified objec type; otherwise zero.

**Function:**

**int CanConvertSplineShape(Class\_ID to);**

**Remarks:**

This global function is available in release 4.0 and later only.  
Indicates if a SplineObject can convert to the specified class ID.

**Parameters:**

**Class\_ID to**

The Class ID to convert to.

**Return Value:**

Nonzero if the SplineObject can be converted to the specified object type; otherwise zero.

**Function:**

**void RegisterStaticEditTri(Object \*triob);**

**Remarks:**

This global function is available in release 4.0 and later only.  
This method will register the object passed as the editable tri object.

**Parameters:**

**Object \*triob**

The object to register as the editable tri object.

**Function:**

**void RegisterCollapseType(Class\_ID cid, TSTR name, bool canSelfConvert=false);**

**Remarks:**

This global function is available in release 4.0 and later only.  
Registers a class ID of an object that the user can collapse other objects to.  
The type will only appear if the current object returns nonzero from **CanConvertTo(cid)**.

**Parameters:**

**Class\_ID cid**

The class ID the object will collapse to.

**TSTR name**

The name of the collapse-to object type (such as "Editable Poly").

**bool canSelfConvert=false**

Indicates whether an object should be allowed to collapse to itself. (false is generally preferred, so that the collapse-to menu only has relevant entries.)

## Explicit Conversion Functions

See Also: [Class Mesh](#), [Class PatchMesh](#), [Class MNMesh](#) , [Class ObjectConverter](#)

The following functions are available to apply explicit conversions of object types.

### Function:

```
void ConvertMeshToPatch (Mesh &m, PatchMesh &pm, DWORD flags=0);
```

### Remarks:

This method is available in release 4.0 and later only.

This function converts a mesh to a patch mesh.

### Parameters:

#### **Mesh &m**

The mesh to convert.

#### **PatchMesh &pm**

The patchmesh to convert into.

#### **DWORD flags=0**

The conversion flags;

#### **CONVERT\_KEESEL**

Translate subobject selections, so selected vertices, edges, or surface areas (faces or patches) in the input model is still selected in the output.

#### **CONVERT\_USESOFTSEL**

Translate soft selections. Also, in the case of patches turning into meshes with more vertices, soft selections are used to interpolate between selected patch corners and nonselected ones.

#### **CONVERT\_PATCH\_USEQUADS**

When converting to a Patch model, create quad patches for any quads in the input mesh or polymesh. (Otherwise, turns everything into triangle patches).

#### **CONVERT\_NO\_RELAX**

Indicates in patch-to-mesh conversions that the Relax parameters in the patch should be ignored.

**Function:**

```
void ConvertPatchToMesh (PatchMesh &pm, Mesh &m, DWORD  
flags=0);
```

**Remarks:**

This method is available in release 4.0 and later only.

This function converts a patchmesh to a mesh.

**Parameters:****PatchMesh &pm**

The patchmesh to convert.

**Mesh &m**

The mesh to convert into..

**DWORD flags=0**

The conversion flags;

**CONVERT\_KEESEL**

Translate subobject selections, so selected vertices, edges, or surface areas (faces or patches) in the input model is still selected in the output.

**CONVERT\_USESOFTSEL**

Translate soft selections. Also, in the case of patches turning into meshes with more vertices, soft selections are used to interpolate between selected patch corners and nonselected ones.

**CONVERT\_PATCH\_USEQUADS**

When converting to a Patch model, create quad patches for any quads in the input mesh or polymesh. (Otherwise, turns everything into triangle patches).

**CONVERT\_NO\_RELAX**

Indicates in patch-to-mesh conversions that the Relax parameters in the patch should be ignored.

**Function:**

```
void ConvertPolyToPatch (MNMESH & from, PatchMesh & to,  
DWORD flags=0);
```

**Remarks:**

This method is available in release 4.0 and later only.

This function converts a poly object to a patch mesh.

**Parameters:**

**MNMesh & from**

The poly mesh to convert.

**PatchMesh & to**

The patchmesh to convert into.

**DWORD flags=0**

The conversion flags;

**CONVERT\_KEESEL**

Translate subobject selections, so selected vertices, edges, or surface areas (faces or patches) in the input model is still selected in the output.

**CONVERT\_USESOFTSEL**

Translate soft selections. Also, in the case of patches turning into meshes with more vertices, soft selections are used to interpolate between selected patch corners and nonselected ones.

**CONVERT\_PATCH\_USEQUADS**

When converting to a Patch model, create quad patches for any quads in the input mesh or polymesh. (Otherwise, turns everything into triangle patches).

**CONVERT\_NO\_RELAX**

Indicates in patch-to-mesh conversions that the Relax parameters in the patch should be ignored.

**Function:**

**void ConvertPatchToPoly (PatchMesh & from, MNMesh & to, DWORD flags=0);**

**Remarks:**

This method is available in release 4.0 and later only.

This function converts a patch mesh to a poly object.

**Parameters:**

**PatchMesh & to**

The patchmesh to convert into.

**MNMesh & from**

The poly mesh to convert.

**DWORD flags=0**

The conversion flags;

**CONVERT\_KEESEL**

Translate subobject selections, so selected vertices, edges, or surface areas (faces or patches) in the input model is still selected in the output.

**CONVERT\_USESOFTSEL**

Translate soft selections. Also, in the case of patches turning into meshes with more vertices, soft selections are used to interpolate between selected patch corners and nonselected ones.

**CONVERT\_PATCH\_USEQUADS**

When converting to a Patch model, create quad patches for any quads in the input mesh or polymesh. (Otherwise, turns everything into triangle patches).

**CONVERT\_NO\_RELAX**

Indicates in patch-to-mesh conversions that the Relax parameters in the patch should be ignored.

## Class ISubObjType

See Also: [Class InterfaceServer](#), [Class BaseObject](#), [Class MaxIcon](#), [Class GenSubObjType](#).

```
class ISubObjType : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

Developers have to return a class derived from this class with implementations for all the methods when implementing sub-objects for obejcts and modifiers (see **BaseObject::GetSubObjType()**).

### Methods:

public:

### Prototype:

```
virtual MaxIcon *GetIcon()=0;
```

### Remarks:

Returns a pointer to the icon for this sub-object type. This icon appears in the stack view beside the name.

### Prototype:

```
virtual TCHAR *GetName()=0;
```

### Remarks:

Returns the name of this sub-object type that appears in the stack view.

## Class GenSubObjType

See Also: [Class ISubObjType](#), [Class BaseObject](#), [Class MaxIcon](#).

```
class GenSubObjType : public ISubObjType
```

### Description:

This class is available in release 4.0 and later only.

This class provides a generic implementation for sub-object types. Instead of having to create a sub-class of **ISubObjType** the constructors of this class may be used to initialize private data members of the class. Then implementations of the **GetName()** and **GetIcon()** methods of **ISubObjType** are provided which simply return the data members.

This SubObjectType will either use the **subObjectIcons\_16i.bmp** and **SubObjectIcons\_16a.bmp** bitmaps in the **UICONS** directory (for the **GenSubObjType(int idx)** constructor), or any other bmp file that is specified in the **GenSubObjType(TCHAR \*nm, TCHAR\* pFilePrefix, int idx)** constructor. The bitmap files have to reside in the **UICONS** directory.

All methods of this class are implemented by the System.

### Methods:

public:

### Prototype:

```
GenSubObjType(TCHAR *nm, TCHAR* pFilePrefix, int idx) :  
name(nm), mIcon(NULL), mIdx(idx), mFilePrefix(pFilePrefix);
```

### Remarks:

Constructor. The private data members are inialized to the values passed and the corresponding Get methods of this class will return these data members.

### Parameters:

**TCHAR \*nm**

The name for this sub-object type.

**TCHAR\* pFilePrefix**

The BMP imagelist file name prefix for this sub-object type. This is the file name, without the extension, and with the assumption that the file is in the **ui\icons** directory. For example specifying **\_T("SubObjectIcons")** for this

parameter indicates the file **UICONS\SubObjectIcons\_16i.bmp** if the small icons are in use and **SubObjectIcons\_24i.bmp** if the large icons are in use.

**int idx**

This is the one based index into the image list of the icon to use.

**Prototype:**

```
GenSubObjType(int idx) : mIcon(NULL), mIdx(idx),  
mFilePrefix(_T("SubObjectIcons"));
```

**Remarks:**

This constructor assumes that the icons are in either **UICONS\SubObjectIcons\_16i.bmp** or **SubObjectIcons\_24i.bmp** depending on which size icons are in use by the system. In this case only the index into the image list is required.

**Parameters:**

**int idx**

This is the one based index into the image list of the icon to use.

**Prototype:**

```
~GenSubObjType();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetName(TCHAR *nm);
```

**Remarks:**

Sets the name for this sub-object type.

**Parameters:**

**TCHAR \*nm**

The name to set.

**Prototype:**

**TCHAR \*GetName();**

**Remarks:**

Returns the name for this sub-object type. This is the implementation of the ISubObjType method.

**Prototype:**

**MaxIcon \*GetIcon();**

**Remarks:**

Returns the icon for this sub-object type. This is the implementation of the ISubObjType method.

## Class MaxIcon

See Also: [Class InterfaceServer](#), [Class ICustButton](#), [Class MaxBmpFileIcon](#).

class MaxIcon : public InterfaceServer

### Description:

This class is available in release 4.0 and later only.

This abstract class represents an icon image for toolbar buttons, icons in list boxes, etc. The class is based on Win32 ImageLists. MaxIcons must provide an image list and index into the list for both large (24x24) and small (16x15) icons.

### Methods:

public:

### Prototype:

**virtual HIMAGELIST GetDefaultImageList() = 0;**

### Remarks:

Returns the handle to the image list for the size of icons that the user has chosen.

### Prototype:

**virtual HIMAGELIST GetSmallImageList() = 0;**

### Remarks:

Returns the image list for small icons.

### Prototype:

**virtual HIMAGELIST GetLargeImageList() = 0;**

### Remarks:

Returns the image list for large icons.

### Prototype:

**virtual int GetSmallImageIndex(bool enabledVersion = true,  
COLORREF backgroundColor =  
GetCustSysColor(COLOR\_BTNFACE)) = 0;**

**Remarks:**

Returns the zero based index into the image list for the small version of this particular icon.

**Parameters:**

**bool enabledVersion = true**

Pass true to get the index of the enabled version of the icon; false to get the disabled version.

**COLORREF backgroundColor =  
GetCustSysColor(COLOR\_BTNFACE)**

The background color to use for the icon.

Specifies the windows color definition. See [List of Standard Color IDs](#). For a full list of windows color definitions, please refer to the Win32 API, in particular the methods **GetSysColor()** and **SetSysColor()**.

**Prototype:**

```
virtual int GetLargeImageIndex(bool enabledVersion = true,  
COLORREF backgroundColor =  
GetCustSysColor(COLOR_BTNFACE)) = 0;
```

**Remarks:**

Returns the zero based index into the image list for the large version of this particular icon.

**Parameters:**

**bool enabledVersion = true**

Pass true to get the enabled version of the icon or false to get the disabled version.

**COLORREF backgroundColor =  
GetCustSysColor(COLOR\_BTNFACE)**

The background color to use for the icon.

Specifies the windows color definition. See [List of Standard Color IDs](#). For a full list of windows color definitions, please refer to the Win32 API, in particular the methods **GetSysColor()** and **SetSysColor()**.

**Prototype:**

```
int GetDefaultImageIndex(bool enabledVersion = true,  
COLORREF backgroundColor =  
GetCustSysColor(COLOR_BTNFACE));
```

**Remarks:**

Returns the zero based index into the image list for the default version of this particular icon.

**Parameters:**

**bool enabledVersion = true**

Pass true to get the enabled version of the icon or false to get the disabled version.

**COLORREF backgroundColor =  
GetCustSysColor(COLOR\_BTNFACE)**

The background color to use for the icon.

Specifies the windows color definition. See [List of Standard Color IDs](#). For a full list of windows color definitions, please refer to the Win32 API, in particular the methods **GetSysColor()** and **SetSysColor()**.

**Prototype:**

```
virtual bool UsesAlphaMask() = 0;
```

**Remarks:**

This method returns true if the icons has an alpha mask that needs to be blended with the background color and false if it doesn't use an alpha mask.

## Class PolyObject

See Also: [Class GeomObject.](#), [Class MNMesh](#)

class PolyObject: public GeomObject

### Description:

This class is available in release 4.0 and later only.

3ds max 4 introduces a new type of pipeline object, the polygon-based mesh object, or **PolyObject** for short. This object is based on the **MNMesh** which has been present in the SDK for some time and used for 3ds max effects like MeshSmooth, Boolean, and Connect.

PolyObjects are more restricted than TriObjects in that they only support "manifold topology". That is to say, you can't create "rat's nest" meshes out of polygon meshes. Each edge in a polygon mesh can be referenced only once on each "side", with a well defined "outside" and "inside" direction for every element of polygons. In TriObjects, you could create a mesh with 5 vertices and 3 faces: (0,1,2), (0,1,3), and (0,1,4). This would be illegal in PolyObjects because the edge (0,1) is referenced in the (0->1) direction by three different faces. (0,1,2), (1,0,3) would be a legal pair of faces, however, because the (0,1) edge is referenced only once in each direction. This implies also that the two faces have consistent normals - there's a well-defined inside and outside at the edge. It's impossible to have neighboring faces have inconsistent normals in a PolyObject, which is why the FlipNormals method in Editable Poly only works on entire elements.

PolyObjects have a complete edge list present virtually all the time. They also have full topological links - the edges reference the vertices and faces they use, the faces reference the vertices and edges, and the vertices also reference the edges and faces. This is somewhat wasteful of memory, but it makes it very easy to navigate the mesh and do complex algorithms like subdivision or Booleans. (Of course this also means there's more to keep track of in these operations.)

Mesh objects flowing up the pipeline should be freely convertible between TriObjects and PolyObjects. Virtually all data should be preserved in converting back and forth between these types. Exceptions include PolyObject edge data, such as crease values, which are lost upon conversion to TriObjects (since TriObjects have no edge lists).

Virtually all the public methods of PolyObject are based on the equivalent

methods in TriObject. class PolyObject provides implementations of all the required methods of Animatable, ReferenceMaker, ReferenceTarget, Base Object, Object, and GeomObject. All methods of this class are implemented by the system.

#### **Data Members:**

public:

**MNMesh mm;**

See [class MNMesh](#) for information about manipulating the mesh.

**The following global functions are not part of this class but are available for use:**

#### **Function:**

**ClassDesc\* GetPolyObjDescriptor();**

#### **Remarks:**

Gets the class descriptor for the PolyObject.

#### **Function:**

**void RegisterEditPolyObjDesc(ClassDesc\* desc);**

#### **Remarks:**

Allows a plugin to register an Editable Poly object. (This is done by epoly.dlo in the standard 3ds max distribution.) This is the object which is collapsed to when the stack is collapsed. The default if no such object is registered is to simply collapse to a PolyObject (which has no UI parameters).

#### **Parameters:**

**ClassDesc\* desc**

A pointer to the class descriptor.

#### **Function:**

**ClassDesc\* GetEditPolyObjDesc();**

#### **Remarks:**

This method is available in release 4.0 and later only.

Gets the class descriptor for the currently registered Editable Poly object.

**Function:**

**PolyObject \*CreateEditablePolyObject();**

**Remarks:**

This method is available in release 4.0 and later only.

Returns an Editable Poly object from the currently registered EPoly descriptor, cast as a PolyObject.

**Methods:**

public:

**Prototype:**

**DWORD GetSubSelState();**

**Remarks:**

This method returns the selection levels defined for class MNMesh:

**MNM\_SL\_OBJECT** (0), **MNM\_SL\_VERTEX** (1),  
**MNM\_SL\_EDGE** (2), and **MNM\_SL\_FACE** (3).

**Prototype:**

**void SetSubSelState(DWORD s);**

**Remarks:**

This method allows you to set the selection levels defined for class MNMesh:

**MNM\_SL\_OBJECT** (0), **MNM\_SL\_VERTEX** (1),  
**MNM\_SL\_EDGE** (2), and **MNM\_SL\_FACE** (3).

**Parameters:**

**DWORD s**

The selection level to set.

**Prototype:**

**BOOL CheckObjectIntegrity();**

**Remarks:**

Uses the **MNMesh::CheckAllData()** method, which uses **DebugPrint()** to give details about any errors that it finds.

**Prototype:**

**MNMesh& GetMesh();**

**Remarks:**

Accessor for the MNMesh mm data member.

**Displacement mapping subdivision methods in PolyObject**

Displacement mapping in materials can be enhanced by a subdivision algorithm which subdivides as needed based on the map. This algorithm is implemented in class TriObject, but can be accessed by PolyObjects since they need to convert to TriObjects for rendering anyway. The following methods are used to access and set the displacement mapping parameters.

**Prototype:**

**BOOL CanDoDisplacementMapping();**

**Remarks:**

Implementation of a class GeomObject method which indicates whether or not this object supports displacement mapping. It returns true as long as **GetDisplacementDisable()** returns false.

**Prototype:**

**void SetDisplacementApproxToPreset(int preset);**

**Remarks:**

Sets displacement subdivision parameters to match one of the standard displacement approximation presets.

**Parameters:**

**int preset**

Should be 0 for low, 1 for medium, or 2 for high.

**Prototype:**

**void SetDisplacementDisable (bool disable);**

**Remarks:**

Disables displacement subdivision (without altering the parameters).

**Parameters:**

**bool disable**

TRUE to disable; FALSE to enable.

**Prototype:**

**void SetDisplacementParameters (TessApprox & params);**

**Remarks:**

Sets most of the displacement parameters. See [class TessApprox](#) for details.

**Parameters:**

**TessApprox & params**

The tessellation approximation data.

**Prototype:**

**void SetDisplacementSplit (bool split);**

**Remarks:**

Controls displacement subdivision splitting.

**Parameters:**

**bool split**

TRUE to set; FALSE to unset.

**Prototype:**

**void SetDisplacement (bool displace);**

**Remarks:**

Turns displacement on or off.

**Parameters:**

**bool displace**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**bool GetDisplacementDisable () const;**

**Remarks:**

Indicates whether displacement subdivision is currently disabled.

**Prototype:**

**TessApprox GetDisplacementParameters () const;**

**Remarks:**

Accesses most of the displacement parameters. See [class TessApprox](#) for details.

**Prototype:**

**bool GetDisplacementSplit () const;**

**Remarks:**

Indicates whether displacement subdivision splitting is on.

**Prototype:**

**bool GetDisplacement () const;**

**Remarks:**

Indicates whether displacement subdivision is on.

**Prototype:**

**TessApprox &DispParams();**

**Remarks:**

Accessor method for the displacement subdivision parameters that can be quicker to use than **SetDisplacementParameters()** and **GetDisplacementParameters()**.

**Prototype:**

**BOOL IsPointSelected(int i)**

**Remarks:**

This method is available in release 4.2 and later only.

Returns TRUE if the 'i-th' point is selected; otherwise FALSE.

**Parameters:**

**int i**

The zero based index of the point to check.

**Prototype:**

**float PointSelection(int i);**

**Remarks:**

This method is available in release 4.2 and later only.

Returns a floating point weighted point selection.

**Parameters:**

**int i**

The zero based index of the point to check.

**Return Value:**

Returns 1.0f if selected and 0.0f if not.

**Prototype:**

**float BOOL PolygonCount(TimeValue t, int& numFaces, int& numVerts);**

**Remarks:**

This method is available in release 4.2 and later only.

Retreives the number of faces and vertices of the polyginal mesh representation of the object. Note: Plug-In developers should use the global function GetPolygonCount(Object\*, int&, int&) to retrieve the number f vertices and faces in an arbitrary object.

**Parameters:**

**TimeValue t**

The time at which to compute the number of faces and vertices.

**int& numFaces**

The number of faces is returned here.

**int& numVerts**

The number of vertices is returned here.

**Default Implementation:**

```
{ return TRUE; }
```

## Class EPoly

See Also: [Class FPMixinInterface](#), [Class MNMesh](#), [Class IParamBlock2](#), [List of Edge Data Channels](#), [List of Vertex Data Channels](#)

```
class EPoly : public FPMixinInterface
```

### Description:

This class is available in release 4.0 and later only.

The EPoly class is the main interface to the Editable Poly Object. This class is a virtual class with no data members. More details can be found in the SDK samples under `\MAXSDK\SAMPLES\MESHEDITABLEPOLY`.

[Cache and update methods](#)

[Selection and component flag access](#)

[UI Button and Command Mode methods](#)

[Transform Methods](#)

[Slice plane accessors](#)

[Component data access methods](#)

[Displacement approximation methods](#)

[Drag operations](#)

[Regular Operations](#)

### Prototype:

```
FPInterfaceDesc *GetDesc();
```

### Remarks:

This method returns a description of the published function interface (for scripter access to EPoly functions).

### Prototype:

```
virtual MNMesh *GetMeshPtr();
```

### Remarks:

This method returns a pointer to the Editable Poly's **MNMesh**. See [class MNMesh](#) for fun things to do with an MNMesh.

### Default Implementation:

```
{ return NULL; }
```

### Prototype:

```
virtual IParamBlock2 *getParamBlock();
```

### Remarks:

Gets a pointer to the Editable Poly's parameter block, which can be used to get or set Editable Poly UI parameters. (See class IParamBlock2 for more information on access methods.) Here is a list of the available parameters, as defined in the ePolyParameters enum near the top of **iEPoly.h**. (Please see the Editable Poly documentation in the normal 3ds max User Reference file for more information on these parameters.)

#### Selection dialog parameters:

**ep\_by\_vertex**: select by vertex (checkbox)

**ep\_ignore\_backfacing**: Ignore backfacing (checkbox)

**ep\_show\_normals**: Currently unused.

**ep\_normal\_size**: Currently unused.

#### Soft Selection dialog parameters:

**ep\_ss\_use**: Use soft selection (checkbox)

**ep\_ss\_retro**: Currently unused

**ep\_ss\_edist\_use**: Use edge-based distances to compute soft selection (checkbox)

**ep\_ss\_edist**: Maximum number of edges to traverse in computing edge-based distances (int spinner - range: 1-999999)

**ep\_ss\_affect\_back**: Affect backfacing in soft selection (checkbox)

**ep\_ss\_falloff**: Falloff value for soft selection (float spinner)

**ep\_ss\_pinch**: Pinch value for soft selection (float spinner)

**ep\_ss\_bubble**: Bubble value for soft selection (float spinner)

#### Edit Geometry dialog parameters:

**ep\_extrusion\_type**: Type of face extrusion. (Radio.) Values:

**0**: Extrude by group (cluster)

**1**: Extrude by local normals

**2:** Extrude by polygon - each polygon extrudes separately.

**ep\_split:** Controls whether the Cut algorithm splits the mesh open.  
(Checkbox)

**ep\_refine\_ends:** Currently unused

**ep\_weld\_threshold:** Threshold for welding selected vertices. (float spinner)

**ep\_weld\_pixels:** Pixel threshold for Target welding (int spinner)

**Subdivide dialog parameters:**

**ep\_ms\_smoothness:** Smoothness value for MeshSmooth type subdivision  
(float spinner, range 0-1)

**ep\_ms\_smooth:** Currently unused.

**ep\_ms\_sep\_smooth:** Separate by smoothing groups for MeshSmooth type subdivision (checkbox)

**ep\_ms\_sep\_mat:** Separate by material IDs for MeshSmooth type subdivision (checkbox)

**ep\_tess\_type:** Tessellation type (Radio) Values:

**0:** by edge

**1:** by face.

**ep\_tess\_tension:** Tessellation tension (float spinner)

**Surface Properties dialog parameters (object level):**

**ep\_surf\_subdivide:** Apply NURMS-style MeshSmooth subdivision to polymesh. (checkbox)

**ep\_surf\_subdiv\_smooth:** Apply MeshSmooth smoothing group algorithm after subdividing (checkbox)

**ep\_surf\_ig\_sel:** Currently unused.

**ep\_surf\_iter:** Number of iterations of subdivision (int spinner)

**ep\_surf\_thresh:** "Smoothness" threshold for adaptive subdivision (float spinner, range 0-1)

**ep\_surf\_riter:** Render value of iterations (int spinner)

**ep\_surf\_rthresh:** Render value of smoothness threshold (float spinner, range 0-1)

**ep\_surf\_use\_riter:** Use Render iterations value when rendering (checkbox)

**ep\_surf\_use\_rthresh:** Use Render smoothness threshold when rendering (checkbox)

**ep\_surf\_sep\_smooth:** Separate by smoothing groups in subdivision (checkbox)

**ep\_surf\_sep\_mat:** Separate by materials in subdivision (checkbox)

**ep\_surf\_update:** Update type for subdivision (radio). Values:

0: Update always

1: Update when rendering

2: Update manually

**Surface Properties dialog parameters (vertex level):**

**ep\_vert\_sel\_color:** Target color for select-by-color (color swatch)

**ep\_vert\_selc\_r:** Tolerance of red values in select-by-color. (int spinner, range 0-255)

**ep\_vert\_selc\_g:** Tolerance of green values in select-by-color. (int spinner, range 0-255)

**ep\_vert\_selc\_b:** Tolerance of blue values in select-by-color. (int spinner, range 0-255)

**ep\_vert\_color\_selby:** Which kind of color to select by in select-by-color. Radio values:

0: Select by regular vertex color

1: Select by vertex illumination

**Surface Properties dialog parameters (face level):**

**ep\_face\_smooth\_thresh:** Autosmooth threshold (float spinner, angle units).

**Subdivision Displacement parameters (object level):**

**ep\_sd\_use:** Apply subdivision displacement (checkbox)

**ep\_sd\_split\_mesh:** Split the mesh (checkbox)

**ep\_sd\_method:** Subdivision displacement method (radio). Values:

0: Regular method

1: Spatial method

2: Curvature method

3: Use both spatial & curvature methods.

**ep\_sd\_tess\_steps:** Tessellation steps (for regular method) (int spinner)

**ep\_sd\_tess\_edge:** Edge size (for spatial method) (float spinner)

**ep\_sd\_tess\_distance:** Distance (for curvature method) (float spinner)

**ep\_sd\_tess\_angle:** Angle value (for curvature method) (float spinner)

**ep\_sd\_view\_dependent:** View dependency (checkbox)

**Advanced Subdivision Displacement parameters:**

**ep\_asd\_style:** Subdivision style. Radio values:

**0:** Grid-based

**1:** Tree-based

**2:** Delauney algorithm.

**ep\_asd\_min\_iters:** Minimum number of iterations (grid or tree style) (int spinner)

**ep\_asd\_max\_iters:** Maximum number of iterations (grid or tree style) (int spinner)

**ep\_asd\_max\_tris:** Maximum number of triangles (Delauney style) (int spinner)

**Default Implementation:**

```
{ return NULL; }
```

## Cache and update methods

**Prototype:**

```
virtual void LocalDataChanged(DWORD parts);
```

**Remarks:**

This method is used to indicate to the EPoly that some parts of its mesh have changed. This is automatically handled by most EPoly methods; you only need to use it if you're directly manipulating the mesh yourself.

**Parameters:**

**DWORD parts**

Parts of the mesh that have been changed, such as **PART\_GEOM**, **PART\_TOPO**, etc.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void InvalidateSoftSelectionCache();
```

**Remarks:**

This method invalidates soft selection values in the mesh and in any cached data.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void InvalidateDistanceCache();
```

**Remarks:**

This method invalidates pre-computed distances on which soft selection values are based (as well as soft selection values) - note this is automatically done when you call **LocalDataChanged (PART\_GEOM)**. This should also be done if the soft selection parameters are changed.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void RefreshScreen();
```

**Remarks:**

This is a handy method that does a simple call to **ip->RedrawViewports**. Also updates the named selection dropdown list. Also, if the EPoly project has been compiled as a debug build, it will verify that the MNMesh is free of errors using the **MNMesh::CheckAllData** method.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual bool Editing();**

**Remarks:**

This method indicates if the Editable Poly object is currently being edited in the modifier panel (and has its UI present).

**Default Implementation:**

```
{ return FALSE; }
```

## Selection and component flag access

**Prototype:**

```
virtual int GetEPolySelLevel();
```

**Remarks:**

This method returns the EPoly selection level, as defined by the **ePolySelLevel enum**: One of the following values; **EP\_SL\_OBJECT**, **EP\_SL\_VERTEX**, **EP\_SL\_EDGE**, **EP\_SL\_BORDER**, **EP\_SL\_FACE**, **EP\_SL\_ELEMENT**.

**Default Implementation:**

```
{ return EP_SL_OBJECT; }
```

**Prototype:**

```
virtual int GetMNSelLevel();
```

**Remarks:**

This method returns the MNMesh's selection level, as defined by the **PMeshSelLevel enum** in **MNMesh.h**: one of the following; **MNM\_SL\_OBJECT**, **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, **MNM\_SL\_FACE**. (Note that the Editable Poly selection levels **EP\_SL\_BORDER** and **EP\_SL\_ELEMENT** are considered varieties of **MNM\_SL\_EDGE** and **MNM\_SL\_FACE** selection types, respectively.)

**Default Implementation:**

```
{ return MNM_SL_OBJECT; }
```

**Prototype:**

**virtual void SetEPolySelLevel(int level);**

**Remarks:**

This method sets the Editable Poly selection level.

**Parameters:**

**int level**

Possible values are defined by the **ePolySelLevel enum**: one of **EP\_SL\_OBJECT**, **EP\_SL\_VERTEX**, **EP\_SL\_EDGE**, **EP\_SL\_BORDER**, **EP\_SL\_FACE**, **EP\_SL\_ELEMENT**.

**Default Implementation:**

{ }

**Prototype:**

**virtual bool EpGetVerticesByFlag(BitArray & vset, DWORD flags, DWORD fmask=0x0);**

**Remarks:**

This method fills in a **BitArray** depending on whether or not each **MNVert** in the **MNMesh** has a particular flag or set of flags set or cleared.

Example: for instance to set the **BitArray** according to selected vertices, you'd just call **EpGetVerticesByFlag (vset, MN\_SEL)**. But to find vertices which do not have the **MN\_DEAD** flag set, but which do have the **MN\_WHATEVER** flag set, you'd call **EpGetVerticesByFlag (vset, MN\_WHATEVER, MN\_WHATEVER|MN\_DEAD)**.

**Parameters:**

**BitArray & vset**

The array for output to be stored in. The **vset** will be set to size of the number of verts in the mesh.

**DWORD flags**

The flags we're looking for in the vertices

**DWORD fmask=0x0**

The mask of flags we're checking. This is automatically or'd with "flags".

**Return Value:**

TRUE if successful, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpGetEdgesByFlag(BitArray & eset, DWORD flags,  
    DWORD fmask=0x0);
```

**Remarks:**

This method fills in a **BitArray** depending on whether or not each **MNEdge** in the **MNMesh** has a particular flag or set of flags set or cleared.

Example: for instance to set the **BitArray** according to selected edges, you'd just call **EpGetEdgesByFlag (eset, MN\_SEL)**. But to find edges which do not have the **MN\_DEAD** flag set, but which do have the **MN\_WHATEVER** flag set, you'd call **EpGetEdgesByFlag (eset, MN\_WHATEVER, MN\_WHATEVER|MN\_DEAD)**.

**Parameters:****BitArray & vset**

The array for output to be stored in. eset will be set to size of the number of edges in the mesh.

**DWORD flags**

The flags we're looking for in the edges

**DWORD fmask=0x0**

The mask of flags we're checking. This is automatically or'd with "flags".

**Return Value:**

TRUE if successful, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpGetFacesByFlag(BitArray & fset, DWORD flags,  
    DWORD fmask=0x0);
```

**Remarks:**

This method fills in a **BitArray** depending on whether or not each **MNFace**

in the **MNMesh** has a particular flag or set of flags set or cleared.

Example: for instance to set the **BitArray** according to selected faces, you'd just call **EpGetFacesByFlag (fset, MN\_SEL)**. But to find faces which do not have the **MN\_DEAD** flag set, but which do have the **MN\_WHATEVER** flag set, you'd call **EpGetFacesByFlag (fset, MN\_WHATEVER, MN\_WHATEVER|MN\_DEAD)**.

**Parameters:**

**BitArray & vset**

The array for output to be stored in. fset will be set to size of the number of faces in the mesh.

**DWORD flags**

The flags we're looking for in the faces

**DWORD fmask=0x0**

The mask of flags we're checking. This is automatically or'd with "flags".

**Return Value:**

TRUE if successful, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void EpSetVertexFlags(BitArray &vset, DWORD flags,  
DWORD fmask=0x0, bool undoable=true);
```

**Remarks:**

This method sets **MNVert** flags based on a **BitArray**.

Example: to hide vertices specified by the **BitArray**, you'd just call **EpSetVertexFlags (vset, MN\_HIDE)**. To unhide the vertices, you'd use **EpSetVertexFlags (vset, 0, MN\_HIDE)**.

**Parameters:**

**BitArray & vset**

This bitarray indicates which vertices should have their flags modified.

**DWORD flags**

The flags to set.

**DWORD fmask=0x0**

The flag mask - if it includes bits not in "flags", those bits are cleared in the specified vertices.

**bool undoable=true**

If (**undoable && theHold.Holding()**), a restore object for this flag change will be added to the current undo stack.

**Default Implementation:**

```
{ return; }
```

**Prototype:**

```
virtual void EpSetEdgeFlags(BitArray &eset, DWORD flags,  
    DWORD fmask = 0x0, bool undoable=true);
```

**Remarks:**

This method sets **MNEdge** flags based on a **BitArray**.

Example: to select edges specified by the **BitArray**, you'd just call **EpSetEdgeFlags (eset, MN\_SEL)**. To clear selection on the edges, you'd use **EpSetEdgeFlags (eset, 0, MN\_SEL)**.

**Parameters:****BitArray & vset**

This bitarray indicates which edges should have their flags modified.

**DWORD flags**

The flags to set.

**DWORD fmask=0x0**

The flag mask - if it includes bits not in "flags", those bits are cleared in the specified edges.

**bool undoable=true**

If (**undoable && theHold.Holding()**), a restore object for this flag change will be added to the current undo stack.

**Default Implementation:**

```
{ return; }
```

**Prototype:**

**virtual void EpSetFaceFlags(BitArray &fset, DWORD flags, DWORD fmask = 0x0, bool undoable=true);**

**Remarks:**

This method sets **MNFace** flags based on a **BitArray**.

Example: to hide faces specified by the **BitArray**, you'd just call **EpSetFaceFlags (fset, MN\_HIDE)**. To unhide the faes, you'd use **EpSetFaceFlags (fset, 0, MN\_HIDE)**.

**Parameters:**

**BitArray & vset**

This bitarray indicates which faces should have their flags modified.

**DWORD flags**

The flags to set.

**DWORD fmask=0x0**

The flag mask - if it includes bits not in "flags", those bits are cleared in the specified faces.

**bool undoable=true**

If (**undoable && theHold.Holding()**), a restore object for this flag change will be added to the current undo stack.

**Default Implementation:**

```
{ return; }
```

## **UI Button and Command Mode methods**

These methods allow the developer to simulate pushing the buttons available in the UI. Some buttons in Editable Poly, such as "Delete", complete an operation when pressed. Others, such as "Create", enter into an interactive command mode while they're depressed.

**Prototype:**

**virtual void EpActionButtonOp(int opcode);**

**Remarks:**

Completes the action corresponding to the specified UI button.

**Parameters:**

## **int opcode**

The list of "button operations" is defined by the **epolyButtonOp enum**.

Select dialog button operations:

**epop\_hide**: Hide current selection

**epop\_unhide**: Unhide current selection

**epop\_ns\_copy**: Copy named selection (brings up UI)

**epop\_ns\_paste**: Paste named selection (may bring up UI)

Edit Geometry dialog button operations:

**epop\_cap**: Cap currently selected borders

**epop\_delete**: Delete current selection.

**epop\_detach**: Detach current selection.

**epop\_attach\_list**: Attach any number of nodes using the attach by name dialog.

**epop\_split**: Split currently selected edges.

**epop\_break**: Break currently selected vertices

**epop\_collapse**: Collapse current selection

**epop\_reset\_plane**: Reset the slice plane

**epop\_slice**: Slice

**epop\_weld\_sel**: Weld current vertex or edge selection

**epop\_create\_shape**: Create a shape from current edge selection (brings up UI)

**epop\_make\_planar**: Make current selection planar

**epop\_align\_grid**: Align current selection to construction grid

**epop\_align\_view**: Align current selection at right angles to view.

**epop\_remove\_iso\_verts**: Remove isolated vertices

Subdivide dialog button operations:

**epop\_meshtsmooth**: Subdivide by MeshSmooth (NURMS Style)

**epop\_tessellate**: Subdivide by Tessellation

Surface Properties dialog button operations:

Object Level:

**epop\_update**: Update MeshSmooth subdivision

Vertex Level:

**epop\_selby\_vc:** Select by vertex color

Face Level:

**epop\_retriangulate:** Retriangulate currently selected faces

**epop\_flip\_normals:** Flip normals on currently selected elements

**epop\_selby\_matid:** Select faces by Material ID (brings up UI)

**epop\_selby\_smg:** Select faces by smoothing groups (brings up UI)

**epop\_autosmooth:** Autosmooth currently selected faces

**epop\_clear\_smg:** Clear all smoothing groups on currently selected faces.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EpActionToggleCommandMode(int mode);
```

**Remarks:**

If the user is currently in the specified command mode, this method causes them to exit it. If the user is not, this method will enter it.

**Parameters:**

**int mode**

The list of command modes is defined by the **epolyCommandMode enum** and is given here for reference. (Their names are self-explanatory.)

**epmode\_create\_vertex**

**epmode\_create\_edge**

**epmode\_create\_face**

**epmode\_divide\_edge**

**epmode\_divide\_face**

**epmode\_extrude\_vertex** (note: currently inactive)

**epmode\_extrude\_edge** (note: currently inactive)

**epmode\_extrude\_face**

**epmode\_chamfer\_vertex**  
**epmode\_chamfer\_edge**  
**epmode\_bevel**  
**epmode\_sliceplane**  
**epmode\_cut\_vertex**  
**epmode\_cut\_edge**  
**epmode\_cut\_face**  
**epmode\_weld**  
**epmode\_edit\_tri**

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EpActionEnterPickMode(int mode);
```

**Remarks:**

This method enters the specified pick mode, which is like a command mode but relates to picking nodes.

**Parameters:**

**int mode**

Currently there is only one pick mode supported by EPoly, which is defined in the **epolyPickMode enum: epmode\_attach**, which allows the user to pick a node to attach to this Editable Poly object.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EpActionExitCommandModes();
```

**Remarks:**

This method exits from any command mode the system currently may be in.

**Default Implementation:**

{ }

## Transform Methods

### Prototype:

**virtual void MoveSelection(int level, TimeValue t, Matrix3& partm, Matrix3& tmAxis, Point3& val, BOOL localOrigin);**

### Remarks:

This method moves the current selection (including any soft selection) by the specified amount in the specified coordinate system.

### Parameters:

**int level**

The **enum ePolySelLevel**, being one of the following values;  
**EP\_SL\_OBJECT, EP\_SL\_VERTEX, EP\_SL\_EDGE, EP\_SL\_BORDER, EP\_SL\_FACE, EP\_SL\_ELEMENT.**

**TimeValue t**

The time at which to apply the move operation.

**Matrix3& partm**

The parent transformation matrix.

**Matrix3& tmAxis**

The transformation axis.

**Point3& val**

The vector describing the translation.

**BOOL localOrigin**

TRUE to move based on the local origin, otherwise FALSE.

### Default Implementation:

{ }

### Prototype:

**virtual void RotateSelection(int level, TimeValue t, Matrix3& partm, Matrix3& tmAxis, Quat& val, BOOL localOrigin);**

### Remarks:

This method rotates the current selection (including any soft selection) by the specified amount in the specified coordinate system.

**Parameters:**

**int level**

The **enum ePolySelLevel**, being one of the following values;  
**EP\_SL\_OBJECT, EP\_SL\_VERTEX, EP\_SL\_EDGE,**  
**EP\_SL\_BORDER, EP\_SL\_FACE, EP\_SL\_ELEMENT.**

**TimeValue t**

The time at which to apply the rotate operation.

**Matrix3& partm**

The parent transformation matrix.

**Matrix3& tmAxis**

The transformation axis.

**Quat& val**

The rotation quaternion.

**BOOL localOrigin**

TRUE to rotate based on the local origin, otherwise FALSE.

**Default Implementation:**

{ }

**Prototype:**

**virtual void ScaleSelection(int level, TimeValue t, Matrix3& partm, Matrix3& tmAxis, Point3& val, BOOL localOrigin);**

**Remarks:**

This method scales the current selection (including any soft selection) by the specified amount in the specified coordinate system.

**Parameters:**

**int level**

The **enum ePolySelLevel**, being one of the following values;  
**EP\_SL\_OBJECT, EP\_SL\_VERTEX, EP\_SL\_EDGE,**  
**EP\_SL\_BORDER, EP\_SL\_FACE, EP\_SL\_ELEMENT.**

**TimeValue t**

The time at which to apply the rotate operation.

**Matrix3& partm**

The parent transformation matrix.

**Matrix3& tmAxis**

The transformation axis.

**Point3& val**

The scaling value.

**BOOL localOrigin**

TRUE to scale based on the local origin, otherwise FALSE.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void ApplyDelta(Tab<Point3> & delta, EPoly *epol,  
TimeValue t);
```

**Remarks:**

This method applies a geometric "delta" vector to the current mesh at the specified time. (Note: if t!=0 and the system's animate feature is on, this will set keys.)

**Parameters:**

**Tab<Point3> & delta**

The table of geometry delta vectors.

**EPoly \*epol**

A pointer to the editable poly object to apply to.

**TimeValue t**

The time at which to apply the geometric delta.

**Default Implementation:**

```
{ }
```

**Slice plane accessors**

**Prototype:**

**virtual void EpResetSlicePlane();**

**Remarks:**

This method resets the slice plane.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual void EpGetSlicePlane(Point3 & planeNormal, Point3 & planeCenter, float \*planeSize=NULL);**

**Remarks:**

This method returns the slice plane, as defined by its normal, center, and size. (Size is irrelevant for slicing, but defines the size of the slice gizmo the user sees.)

**Parameters:**

**Point3 & planeNormal**

The plane normal vector.

**Point3 & planeCenter**

The plane center.

**float \*planeSize=NULL**

The size of the plane.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual void EpSetSlicePlane(Point3 & planeNormal, Point3 & planeCenter, float planeSize);**

**Remarks:**

This method sets the slice plane to have the specified normal, center, and size. (Unlike in **EpGetSlicePlane()**, size is not an optional argument here.)

**Parameters:**

**Point3 & planeNormal**

The plane normal vector.

## **Point3 & planeCenter**

The plane center.

## **float planeSize**

The size of the plane.

### **Default Implementation:**

```
{ }
```

## **Component data access methods**

### **Prototype:**

```
virtual Color GetVertexColor(bool *uniform=NULL, int *num=NULL, int mp=0, DWORD flag=MN_SEL, TimeValue t=0);
```

### **Remarks:**

This method obtains the vertex color for the flagged vertices in the indicated map channel.

### **Parameters:**

**bool \*uniform=NULL**

If non-NULL, the bool this pointer points to is set to true if all flagged vertices have the identical color, and false otherwise. (It's set to true if there are 0 vertices.)

**int \*num=NULL**

If non-NULL, the number of vertices currently flagged is computed and stored here.

**int mp=0**

The map channel we're using. Most vertex color applications use the standard vertex color channel, 0. However, you can also use this method with the Illumination channel (**MAP\_SHADING** = -1) or the alpha channel (**MAP\_ALPHA** = -2) - or even with a regular UVW map channel (1-99).

**DWORD flag=MN\_SEL**

This indicates the vertices we look at. If left at the default, selected vertices' colors are analyzed. If flag were to equal **MN\_WHATEVER**, then vertices with the **MN\_WHATEVER** flag would have their colors analyzed.

**TimeValue t=0**

This is not currently used.

**Return Value:**

The color of the flagged vertices, or black (0,0,0) if the vertices' colors are not the same.

**Default Implementation:**

```
{ return Color(1,1,1); }
```

**Prototype:**

```
virtual void SetVertexColor(Color clr, int mp=0, DWORD  
flag=MN_SEL, TimeValue t=0);
```

**Remarks:**

This method sets the vertex color for the flagged vertices in the indicated map channel.

**Parameters:****Color clr**

The color to set the vertices to.

**int mp=0**

The map channel we're using. Most vertex color applications use the standard vertex color channel, 0. However, you can also use this method with the Illumination channel (**MAP\_SHADING** = -1) or the alpha channel (**MAP\_ALPHA** = -2) - or even with a regular UVW map channel (1-99).

**DWORD flag=MN\_SEL**

This indicates which vertices we set. If left at the default, selected vertices' colors are set. If flag were to equal **MN\_WHATEVER**, then vertices with the **MN\_WHATEVER** flag would have their colors set.

**TimeValue t=0**

This is not currently used.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual Color GetFaceColor(bool *uniform=NULL, int *num=NULL, int mp=0, DWORD flag=MN_SEL, TimeValue t=0);
```

**Remarks:**

This method obtains the face color for the flagged faces in the indicated map channel.

**Parameters:**

**bool \*uniform=NULL**

If non-NULL, the bool this pointer points to is set to true if all flagged faces have the identical color, and false otherwise. (It's set to true if there are 0 flagged faces.)

**int \*num=NULL**

If non-NULL, the number of faces currently flagged is computed and stored here.

**int mp=0**

The map channel we're using. Most vertex color applications use the standard vertex color channel, 0. However, you can also use this method with the Illumination channel (**MAP\_SHADING** = -1) or the alpha channel (**MAP\_ALPHA** = -2) - or even with a regular UVW map channel (1-99).

**DWORD flag=MN\_SEL**

This indicates which vertices we set. If left at the default, selected vertices' colors are set. If flag were to equal **MN\_WHATEVER**, then vertices with the **MN\_WHATEVER** flag would have their colors set.

**TimeValue t=0**

This is not currently used.

**Return Value:**

The color of the flagged faces, or black (0,0,0) if the faces' colors are not the same.

**Default Implementation:**

```
{ return Color(1,1,1); }
```

**Prototype:**

```
virtual void SetFaceColor(Color clr, int mp=0, DWORD
```

**flag=MN\_SEL, TimeValue t=0);**

**Remarks:**

This method sets the vertex color for the flagged faces in the indicated map channel.

**Parameters:**

**Color clr**

The color to set the faces to.

**int mp=0**

The map channel we're using. Most vertex color applications use the standard vertex color channel, 0. However, you can also use this method with the Illumination channel (**MAP\_SHADING** = -1) or the alpha channel (**MAP\_ALPHA** = -2) - or even with a regular UVW map channel (1-99).

**DWORD flag=MN\_SEL**

This indicates which vertices we set. If left at the default, selected vertices' colors are set. If flag were to equal **MN\_WHATEVER**, then vertices with the **MN\_WHATEVER** flag would have their colors set.

**TimeValue t=0**

This is not currently used.

**Default Implementation:**

{ }

**Prototype:**

**virtual float GetVertexDataValue(int channel, int \*numSel, bool \*uniform, DWORD vertexFlags, TimeValue t);**

**Remarks:**

This method obtains floating-point vertex data from the flagged vertices in the specified vertex data channel.

**Parameters:**

**int channel**

The vertex data channel we're querying. See the [List of Vertex Data Channels](#) (which are defined in **mesh.h**).

**int \*numSel**

If non-NULL, this is filled in with the current number of flagged vertices.

**bool \*uniform**

If non-NULL, this is set to indicate whether the currently flagged vertices have uniform values or not.

**DWORD vertexFlags**

Indicates which vertices to evaluate. (Use **MN\_SEL** to get vertex data from selected vertices.)

**TimeValue t**

This is not currently used.

**Return Value:**

The vertex data value for the flagged vertices. If the vertices' values vary, the first value found is returned.

**Default Implementation:**

```
{ return 1.0f; }
```

**Prototype:**

```
virtual float GetEdgeDataValue(int channel, int *numSel, bool *uniform, DWORD edgeFlags, TimeValue t);
```

**Remarks:**

This method obtains floating-point edge data from the flagged edges in the specified edge data channel.

**Parameters:**

**int channel**

The edge data channel we're querying. See the [List of Edge Data Channels](#) (which are defined in **mnmesh.h**).

**int \*numSel**

If non-NULL, this is filled in with the current number of flagged edges.

**bool \*uniform**

If non-NULL, this is set to indicate whether the currently flagged edges have uniform values or not.

**DWORD edgeFlags**

Indicates which edges to evaluate. (Use **MN\_SEL** to get vertex data from selected vertices.)

### **TimeValue t**

This is not currently used.

### **Return Value:**

The edge data value for the flagged edges. If the edges' values vary, the first value found is returned.

### **Default Implementation:**

```
{ return 1.0f; }
```

### **Prototype:**

```
virtual void SetVertexDataValue(int channel, float w, DWORD  
vertexFlags, TimeValue t);
```

### **Remarks:**

This method sets floating-point vertex data for the flagged vertices in the specified vertex data channel.

### **Parameters:**

#### **int channel**

The vertex data channel we're modifying. See the [List of Vertex Data Channels](#) (which are defined in **mesh.h**).

#### **float w**

The value to set the flagged vertices to.

#### **DWORD vertexFlags**

Indicates which vertices to modify. (Use **MN\_SEL** to set vertex data in selected vertices.)

#### **TimeValue t**

This is not currently used.

### **Default Implementation:**

```
{ }
```

### **Prototype:**

```
virtual void SetEdgeDataValue(int channel, float w, DWORD  
edgeFlags, TimeValue t);
```

### **Remarks:**

This method sets floating-point edge data for the flagged edges in the specified edge data channel.

**Parameters:**

**int channel**

The edge data channel we're modifying. See the [List of Edge Data Channels](#) (which are defined in **mnmesh.h**).

**float w**

The value to set the flagged edges to.

**DWORD edgeFlags**

Indicates which edges to modify. (Use **MN\_SEL** to set edge data in selected edges.)

**TimeValue t**

This is not currently used.

**Default Implementation:**

{ }

**Prototype:**

**virtual void ResetVertexData(int channel);**

**Remarks:**

This method resets all vertex data in the specified channel. For instance, **ResetEdgeData (VDATA\_WEIGHT)** would reset all vertex weights to 1.

**Parameters:**

**int channel**

The vertex data. See the [List of Vertex Data Channels](#)

**Default Implementation:**

{ }

**Prototype:**

**virtual void ResetEdgeData(int channel);**

**Remarks:**

This method resets all edge data in the specified channel. For instance, **ResetEdgeData (EDATA\_CREASE)** would reset all edge crease values

to 0.

**Parameters:**

**int channel**

The edge data channel. See the [List of Edge Data Channels](#)

**Return Value:**

**Default Implementation:**

{ }

**Prototype:**

**virtual void BeginPerDataModify(int mnSelLevel, int channel);**

**Remarks:**

This method is used in combination with **EndPerDataModify** to store undo information for any vertex or edge data modification.

**Parameters:**

**int mnSelLevel**

Set to one of **MNM\_SL\_VERTEX** or **MNM\_SL\_EDGE** for vertex or edge data respectively.

**int channel**

Indicates the channel of vertex or edge data we're modifying. For instance, **BeginPerDataModify (MNM\_SL\_EDGE, EDATA\_CREASE)** would be used before modifying edge crease information.

**Default Implementation:**

{ }

**Prototype:**

**virtual bool InPerDataModify();**

**Remarks:**

This method returns true if we're between **BeginPerDataModify** and **EndPerDataModify** calls.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void EndPerDataModify(bool success);
```

**Remarks:**

This method completes the undo object corresponding to the vertex or edge data modifications made since the related **BeginPerDataModify** call.

**Parameters:**

**bool success**

If FALSE, the system restores the original vertex colors and throws away the undo object.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void BeginVertexColorModify(int mp=0);
```

**Remarks:**

This method is used in combination with **EndVertexColorModify** to store undo information for any vertex or edge data modification.

**Parameters:**

**int mp=0**

The map channel we're using. Most vertex color applications use the standard vertex color channel, 0. However, you can also use this method with the Illumination channel (**MAP\_SHADING** = -1) or the alpha channel (**MAP\_ALPHA** = -2) - or even with a regular UVW map channel (1-99).

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual bool InVertexColorModify();
```

**Remarks:**

This method returns true if we're between **BeginVertexColorModify** and **EndVertexColorModify** calls.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void EndVertexColorModify(bool success);
```

**Remarks:**

This method completes the undo object corresponding to the vertex color modifications made since the related **BeginVertexColorModify** call.

**Parameters:**

**bool success**

If FALSE, the system restores the original vertex colors and throws away the undo object.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int GetMatIndex(bool *determined, DWORD  
flag=MN_SEL);
```

**Remarks:**

This method obtains the material index for the selected faces.

**Parameters:**

**bool \*determined**

The bool this points to (which should not be NULL) is filled with:

**FALSE** if there are no selected faces or if selected faces have different material indices

**TRUE** if at least one face is selected and all selected faces have the same material ID.

**DWORD flag=MN\_SEL**

Indicates which faces should have their material IDs set.

**Return Value:**

The material index of the selected faces, cast as an int.

**Default Implementation:**

```
{ determined=false; return 0; }
```

**Prototype:**

```
virtual void SetMatIndex(int index, DWORD flag=MN_SEL);
```

**Remarks:**

This method sets the material index for the flagged faces.

**Parameters:**

**int index**

The material index to set flagged faces to.

**DWORD flag=MN\_SEL**

Indicates which faces should have their material IDs set.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void GetSmoothingGroups(DWORD faceFlag, DWORD  
*anyFaces, DWORD *allFaces=NULL);
```

**Remarks:**

This method obtains smoothing group information for the specified faces.

**Parameters:**

**DWORD faceFlag**

Indicates which faces to read smoothing group information from. If this value is 0, all faces are read.

**DWORD \*anyFaces**

**DWORD \*allFaces=NULL**

These two parameters are where the output is stored. "**anyFaces**" has bits set that are present in any of the faces' smoothing groups. "**allFaces**", if non-NULL, has bits set that are present in all of the faces' smoothing groups. In other words, **anyFaces** or's together the faces' groups, while **allFaces** and's

them together.

**Default Implementation:**

```
{ if (anyFaces) *anyFaces = 0; if (allFaces) *allFaces = 0; }
```

**Prototype:**

```
virtual void SetSmoothBits(DWORD bits, DWORD bitmask,  
DWORD flag);
```

**Remarks:**

This method sets (or clears) smoothing group bits in the specified faces.

**Parameters:**

**DWORD bits**

The smoothing group bits to set in flagged faces.

**DWORD bitmask**

The smoothing group bits to clear in flagged faces

**DWORD flag**

Indicates which faces to set smoothing group information in. If this value is 0, all faces are modified.

**Default Implementation:**

```
{ }
```

## Displacement approximation methods

**Prototype:**

```
virtual void SetDisplacementParams();
```

**Remarks:**

This method copies displacement parameters from pblock to polyobject.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void UpdateDisplacementParams();
```

**Remarks:**

This method copies displacement parameters from polyobject to pblock.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void UseDisplacementPreset(int presetNumber);
```

**Remarks:**

This method engages a displacement approximation preset.

**Parameters:**

**int presetNumber**

The presetNumber values are either 0 (low), 1 (medium), or 2 (high).

**Default Implementation:**

```
{ }
```

## Drag operations

**Prototype:**

```
virtual void EpfnBeginExtrude(int msl, DWORD flag, TimeValue t);
```

**Remarks:**

This method is called at the beginning of an interactive extrusion operation. Performs the topological extrusion.

**Parameters:**

**int msl**

Indicates the MNMesh-based selection level we're extruding. (Currently, this must be **MNM\_SL\_FACE**.)

**DWORD flag**

Indicates the faces we're extruding.

**TimeValue t**

The current time.

**Default Implementation:**

{ }

**Prototype:**

**virtual void EpfndEndExtrude(bool accept, TimeValue t);**

**Remarks:**

This method is called at the end of an interactive extrusion operation. Completes the RestoreObjects and finalizes the geometric edit.

**Parameters:**

**bool accept**

If TRUE, end extrude normally. If FALSE, cancel the extrusion completely (undoing the original topological extrusion).

**TimeValue t**

The current time.

**Default Implementation:**

{ }

**Prototype:**

**virtual void EpfndDragExtrude(float amount, TimeValue t);**

**Remarks:**

This method is used to drag the current extrusion to the amount specified. May be called multiple times in one session between **EpfndBeginExtrude** and **EpfndEndExtrude**.

**Parameters:**

**float amount**

The (absolute) amount of the extrusion

**TimeValue t**

The current time

**Default Implementation:**

{ }

**Prototype:**

**virtual void EpfndBeginBevel(int msl, DWORD flag, bool**

**doExtrude, TimeValue t);**

**Remarks:**

This method is called at the beginning of an interactive bevel or outline operation. Performs the topological extrusion if necessary, and prepares certain cached data.

**Parameters:**

**int msl**

Indicates the MNMesh-based selection level we're beveling. (Currently, this must be **MNM\_SL\_FACE**.)

**DWORD flag**

Indicates the faces we're extruding.

**bool doExtrude**

Indicates whether or not a topological extrusion should be done for this bevel. (For instance, leaving this at false you to do "outlining", or to adjust a previous bevel.)

**TimeValue t**

The current time.

**Default Implementation:**

{ }

**Prototype:**

**virtual void EpfndBevel(bool accept, TimeValue t);**

**Remarks:**

This method is called at the end of an interactive bevel. Completes the **RestoreObjects** and finalizes the geometric edit.

**Parameters:**

**bool accept**

If TRUE, end bevel normally. If FALSE, cancel the bevel completely (undoing any earlier topological extrusion).

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EpfndragBevel(float outline, float height, TimeValue t);
```

**Remarks:**

This method is used to drag the current bevel to the outline and height specified. May be called multiple times in one session between **EpfndragBevel** and **EpfndragBevel**.

**Parameters:**

**float outline**

The (positive or negative) outline amount for the bevel.

**float height**

The (positive or negative) height of the bevel.

**TimeValue t**

The current time

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EpfndragChamfer(int msl, TimeValue t);
```

**Remarks:**

This method is called at the beginning of an interactive chamfer operation. Performs the topological changes and prepares certain cached data.

**Parameters:**

**int msl**

Indicates the MNMesh-based selection level we're chamfering. (Either **MNM\_SL\_VERTEX** or **MNM\_SL\_EDGE**.)

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

**virtual void EfnEndChamfer(bool accept, TimeValue t);**

**Remarks:**

This method is called at the end of an interactive chamfer. Completes the **RestoreObjects** and finalizes the geometric edits.

**Parameters:****bool accept**

If TRUE, end chamfer normally. If FALSE, cancel the chamfer completely (undoing the earlier topological changes).

**TimeValue t**

The current time.

**Default Implementation:**

{ }

**Prototype:**

**virtual void EfnDragChamfer(float amount, TimeValue t);**

**Remarks:**

This method is used to drag the current chamfer to the amount specified. May be called multiple times in one session between **EfnBeginChamfer** and **EfnEndChamfer**.

**Parameters:****float amount**

The amount of the chamfer.

**TimeValue t**

The current time

**Default Implementation:**

{ }

**Regular operations****Prototype:**

**virtual bool EfnHide(int msl, DWORD flags);**

**Remarks:**

This method Hides flagged components.

**Parameters:****int msl**

MNMesh selection level - should be either **MNM\_SL\_VERTEX** or **MNM\_SL\_FACE**.

**DWORD flags**

Indicates which components to hide. For instance, **MN\_SEL** would cause it to hide selected vertices or faces.

**Return Value:**

TRUE if components were hidden, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EfnUnhideAll(int msl);
```

**Remarks:**

This method unhides all components at the specified level.

**Parameters:****int msl**

MNMesh selection level - should be either **MNM\_SL\_VERTEX** or **MNM\_SL\_FACE**.

**Return Value:**

TRUE if components were unhidden, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void EfnNamedSelectionCopy(TSTR setName);
```

**Remarks:**

This method copies the named selection specified to the named selection copy/paste buffer.

**Parameters:****TSTR setName**

The name of the selection set.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EpfNamedSelectionPaste(bool useDlgToRename);
```

**Remarks:**

This method pastes selection from named selection copy/paste buffer.

**Parameters:****bool useDlgToRename**

Only matters if there is a name conflict with an existing named selection. If true, the system should throw up an interactive dialog for the user to rename the selection. If false, 3ds max should use string techniques to rename the selection (by adding or increasing a number, etc.)

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int EpfCreateVertex(Point3 pt, bool pt_local=false, bool select=true);
```

**Remarks:**

This method creates a new vertex in the mesh.

**Parameters:****Point3 pt**

The location of the new vertex in object or world space.

**bool pt\_local=false**

If true, the point passed is assumed to be in object space. If false, the point is assumed to represent world space, and the object space location must be computed by the method.

**bool select=true**

Indicates if the new vertex should be selected (have its **MN\_SEL** flag set).

**Return Value:**

The index of the new vertex, or -1 to indicate failure to create the vertex.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

```
virtual int EpfnCreateEdge(int v1, int v2, bool select=true);
```

**Remarks:**

This method creates a new edge, dividing a polygon into two smaller polygons.

**Parameters:**

**int v1, v2**

The endpoint vertices for this edge. These vertices must have at least one face in common, or the creation will fail.

**bool select=true**

Indicates if the new edge should be selected (have its **MN\_SEL** flag set).

**Return Value:**

The index of the edge created.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

```
virtual int EpfnCreateFace(int *v, int deg, bool select=true);
```

**Remarks:**

This method creates a new face on a set of vertices.

**Parameters:**

**int \*v**

An array of vertices for this new face. Note that each vertex must be an "open" vertex - it either must be on no edges or faces, or it must be part of a border (i.e. it's on more edges than faces).

**int deg**

The degree of the new face - and the size of the "v" array.

**bool select=true**

Indicates if the new face should be selected (have its **MN\_SEL** flag set).

**Return Value:**

The index of the face created.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

```
virtual bool EPFN_CapHoles(int msl=MNM_SL_EDGE, DWORD  
targetFlags=MN_SEL);
```

**Remarks:**

This method caps the indicated holes.

**Parameters:**

**int msl=MNM\_SL\_EDGE**

MNMesh-based selection level, one of **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**. The holes are border loops which can be identified by containing selected edges, using selected vertices, or being touched by selected faces.

**DWORD targetFlags=MN\_SEL**

The flags we're looking for (in the vertex, edge, or face levels, according to **msl**) to identify the holes we should cap.

**Return Value:**

TRUE if any hole was successfully capped, otherwise FALSE

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EPFN_Delete(int msl, DWORD delFlag=MN_SEL, bool  
delIsoVerts=false);
```

**Remarks:**

This method deletes the specified components (and any other components

dependent on them).

**Parameters:**

**int msl**

MNMesh-based selection level for deletion to occur on; one of **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**. Note that edge deletion joins, rather than removes, the neighboring faces. Border edges cannot be deleted. Deleting an edge between two quads makes a hexagon. Etc.

**DWORD delFlag=MN\_SEL**

The flag indicating components to delete.

**bool delIsoVerts=false**

If deleting faces, this indicates whether vertices that are left isolated by the face deletion should also be deleted. (Note that in the reverse situation, faces dependent on deleted vertices are always deleted.)

**Return Value:**

TRUE if components were deleted, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void EPFNAttach(INode *node, INode *myNode, TimeValue t);
```

**Remarks:**

This method attaches the specified object to this mesh. The object given is first converted to a polymesh (if needed), then attached as an element in the Editable Poly, then the original is deleted.

**Parameters:**

**INode \*node**

A pointer to the node we want to attach.

**INode \*myNode**

A pointer to this Editable Poly's node (used to match the attached object to our object space).

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EPFNMultiAttach(INodeTab &nodeTab, INode  
*myNode, TimeValue t);
```

**Remarks:**

This method attaches a bunch of nodes to this mesh. The objects in the nodes are converted to polymeshes if needed, then attached as elements in this Editable Poly. (Then the originals are deleted.)

**Parameters:**

**INodeTab &nodeTab**

A table of nodes we want to attach.

**INode \*myNode**

A pointer to this Editable Poly's node (used to match the attached objects to our object space).

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual bool EPFNDetachToElement(int msl, DWORD flag, bool  
keepOriginal);
```

**Remarks:**

This method detaches part of PolyMesh to a separate element.

**Parameters:**

**int msl**

Indicates the MNMesh-based selection level, one of **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag**

Flag indicates which components should be detached. For instance, **MNM\_SL\_VERTEX** and **MN\_SEL** means selected vertices.

**bool keepOriginal**

If TRUE, the original components are left intact and a new element is cloned instead.

**Return Value:**

TRUE if elements are detached, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnDetachToObject(TSTR name, int msl, DWORD  
flag, bool keepOriginal, INode *myNode, TimeValue t);
```

**Remarks:**

This method detaches part of PolyMesh to a separate object.

**Parameters:**

**TSTR name**

The desired name for the new node.

**int msl**

Indicates the MNMesh-based selection level, one of **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag**

Flag indicates which components should be detached. For instance, **MNM\_SL\_VERTEX** and **MN\_SEL** means selected vertices.

**bool keepOriginal**

If TRUE, the original components are left intact and the new object is cloned from them.

**INode \*myNode**

A pointer to this EPoly's node (for transform and other node property access).

**TimeValue t**

The current time.

**Return Value:**

TRUE if parts were detached, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnSplitEdges(DWORD flag=MN_SEL);
```

**Remarks:**

This method "splits" edges by breaking vertices on two or more flagged edges into as many copies as needed. In this way, any path of flagged edges becomes two open seams.

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which edges should be split. (Left at the default, selected edges are split.)

**Return Value:**

TRUE if any topological changes happened, FALSE if nothing happened.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnBreakVerts(DWORD flag=MN_SEL);
```

**Remarks:**

This method breaks vertices into separate copies for each face using them. For example, breaking one corner of a box polymesh turns it into 3 vertices, one for each side that met at that vertex.

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which vertices should be broken. (Left at the default, selected vertices are broken.)

**Return Value:**

TRUE if any topological changes happened, FALSE if nothing happened.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual int EpfndivideFace(int face, Tab<float> &bary, bool  
select=true);
```

**Remarks:**

This method divides the face into triangles meeting at a point described by generalized barycentric coordinates on the face. An n-sided polygon will become n triangles using this technique.

**Parameters:**

**int face**

The face to divide.

**Tab<float> &bary**

A table of floats, of the same size as the face's degree, indicating the contribution of each of the face's vertices to the division point. They should all sum to 1, indicating that the division point is a linear combination of the vertices. If they are all equal (1/n), the center of the face will be used.

**bool select=true**

Indicates whether the new triangles should have the MN\_SEL flag set or not.

**Return Value:**

The index of the new vertex, or -1 for failure.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

```
virtual int EpfndivideEdge(int edge, float prop, bool select=true);
```

**Remarks:**

This method divides an edge in two, creating a new vertex.

**Parameters:**

**int edge**

The edge to divide.

**float prop**

The proportion along the edge for the new vertex, going from 0 at the v1 end to 1 at the v2 end. For instance, a prop of .35 means that the new point will be located at;

$$.65*(v[e[edge].v1].p) + .35*(v[e[edge].v2].p)$$

**bool select=true**

Indicates if the new vertex should be selected. (The new edge picks up its selection flag from the old edge.)

**Return Value:**

The index of the new vertex, or -1 for failure.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

```
virtual bool EpfnCollapse(int msl, DWORD flag);
```

**Remarks:**

This method will collapse the current selection, turning each cluster (in edge or face level) or all selected points into a single point. Some restrictions inherent in a 3ds max polygon-based mesh may prevent a complete collapse, if the result would have an illegal geometry.

**Parameters:**

**int msl**

MNMesh-based selection level, one of **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag**

The flag on the components we wish to collapse.

**Return Value:**

TRUE if any changes occurred, FALSE if nothing happened.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void EpfnExtrudeFaces(float amount, DWORD flag,
```

**TimeValue t);**

**Remarks:**

This method extrudes the flagged faces by the specified amount. Note that this method uses the parameter block value for **ep\_extrusion\_type**. Values can be:

**0:** Extrude by group (cluster)

**1:** Extrude by local normals

**2:** Extrude by polygon - each polygon extrudes separately.

**Parameters:**

**float amount**

The height of the extrusion. Can be positive or negative.

**DWORD flag**

Indicates which faces should be beveled.

**TimeValue t**

The current time.

**Default Implementation:**

{ }

**Prototype:**

**virtual void EfnBevelFaces(float height, float outline, DWORD flag, TimeValue t);**

**Remarks:**

This method bevels the flagged faces by the specified height and outline. Note that this method uses the parameter block value for **ep\_extrusion\_type** in making the extrusion component. Values can be:

**0:** Extrude by group (cluster)

**1:** Extrude by local normals

**2:** Extrude by polygon - each polygon extrudes separately.

**Parameters:**

**float height**

The height of the desired bevel. Can be positive or negative.

**float outline**

The amount of the outlining in the bevel. Positive amounts make the selected region larger; negative amounts make it smaller.

**DWORD flag**

Indicates which faces should be beveled.

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EfnChamferVertices(float amount, TimeValue t);
```

**Remarks:**

This method chamfers the flagged vertices by the specified amount.

**Parameters:****float amount**

The amount of the chamfer.

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EfnChamferEdges(float amount, TimeValue t);
```

**Remarks:**

This method chamfers the flagged edges by the specified amount.

**Parameters:****float amount**

The amount of the chamfer.

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual bool EpfnSlice(Point3 planeNormal, Point3 planeCenter,  
bool flaggedFacesOnly=false, DWORD faceFlags=MN_SEL);
```

**Remarks:**

This method slices the mesh with the specified plane.

**Parameters:**

**Point3 planeNormal**

**Point3 planeCenter**

The definition of the plane, by the normal and the "center" (which can be any point in the plane).

**bool flaggedFacesOnly=false**

If set, only the flagged faces should be sliced. If false, all faces should be sliced.

**DWORD faceFlags=MN\_SEL**

Indicates which faces should be sliced, if **flaggedFacesOnly** is TRUE.

**Return Value:**

TRUE if something has been sliced, or FALSE if nothing happened.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnInSlicePlaneMode();
```

**Remarks:**

This method indicates whether the Editable Poly is currently in Slice Plane mode.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

**virtual int EpfncutVertex(int startv, Point3 destination, Point3 projDir);**

**Remarks:**

This method cuts from one vertex to another. Note that this algorithm uses the parameter block value for **ep\_split**, which controls whether the cut algorithm splits the mesh open.

**Parameters:**

**int startv**

The starting vertex for the cut

**Point3 destination**

The location of the ending vertex for the cut.

**Point3 projDir**

The direction of the "view". (projDir and the vector between the two vertices define the plane that the cut occurs in.)

**Return Value:**

The destination vertex, or -1 if the cut was unable to be completed for some reason.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

**virtual int EpfncutEdge(int e1, float prop1, int e2, float prop2, Point3 projDir);**

**Remarks:**

This method cuts from one edge to another. Note that this algorithm uses the parameter block value for **ep\_split**, which controls whether the cut algorithm splits the mesh open.

**Parameters:**

**int e1**

The edge index along that edge for the start of the cut.

**float prop1**

The edge proportion along that edge for the start of the cut.

**int e2**

The edge index along that edge for the end of the cut.

**float prop2**

The edge proportion along that edge for the end of the cut.

**Point3 projDir**

The direction of the "view". (projDir and the vector between the two endpoints define the plane that the cut occurs in.)

**Return Value:**

The vertex created at the end of the cut, or -1 if the cut was unable to be completed for some reason.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

```
virtual int EpfncutFace(int f1, Point3 p1, Point3 p2, Point3  
projDir);
```

**Remarks:**

This method cuts from one face to another, subdividing the start and end faces as needed for precisely matching the given start and end points. Note that this algorithm uses the parameter block value for **ep\_split**, which controls whether the cut algorithm splits the mesh open.

**Parameters:**

**int f1**

The face we should start on.

**Point3 p1**

The point (on face f1) for the start of the cut.

**Point3 p2**

The point at the end of the cut.

**Point3 projDir**

The direction of the "view". (projDir and the vector between the two endpoints define the plane that the cut occurs in.)

**Return Value:**

The vertex created at the end of the cut, or -1 if the cut was unable to be

completed for some reason.

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

```
virtual bool EpgnWeldVerts(int vert1, int vert2, Point3  
destination);
```

**Remarks:**

This method welds the specified vertices together (if possible), and puts the result at the location specified.

**Parameters:**

**int vert1, vert2**

The two vertices we wish to weld. Note that these vertices must be "border" vertices, that is, they must be used by some open (one-sided) edges.

**Point3 destination**

The desired location for the result. Usually this is **v[vert1].p**, **v[vert2].p**, or the average of the two.

**Return Value:**

Indicates if any welding successfully occurred. If FALSE, nothing happened.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpgnWeldEdges(int edge1, int edge2);
```

**Remarks:**

This method welds the first edge to the second edge, leaving the result at the location of the second edge.

**Parameters:**

**int edge1, edge2**

The two edges we wish to weld. Note that these must be open (one-sided) edges.

**Return Value:**

Indicates if any welding successfully occurred. (If false, nothing happened.)

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpcfWeldFlaggedVerts(DWORD flag);
```

**Remarks:**

This method welds all flagged vertices together to their average location.

**Parameters:**

**DWORD flag**

Indicates which vertices should be welded. Note that these vertices must be "border" vertices, that is, they must be used by some open (one-sided) edges.

**Return Value:**

Indicates if any welding successfully occurred. If FALSE, nothing happened.)

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpcfWeldFlaggedEdges(DWORD flag);
```

**Remarks:**

This method welds flagged edges together.

**Parameters:**

**DWORD flag**

Indicates which edges should be welded. Note that these edges must be open (one-sided).

**Return Value:**

Indicates if any welding successfully occurred. (If false, nothing happened.)

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

**virtual bool EpcfCreateShape(TSTR name, bool smooth, INode \*myNode, DWORD edgeFlag=MN\_SEL);**

**Remarks:**

This method creates a new shape object from flagged edges in the polymesh.

**Parameters:**

**TSTR name**

The desired name for the new node containing the shape.

**bool smooth**

If TRUE, the new shape should be a smooth curve. If FALSE, it should be a linear shape which exactly follows the edges.

**INode \*myNode**

A pointer to the node of the PolyMesh (used to obtain transform and other node level information for the new shape node).

**DWORD edgeFlag=MN\_SEL**

Indicates which faces should be used in creating this shape.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

**virtual bool EpcfMakePlanar(int msl, DWORD flag=MN\_SEL, TimeValue t=0);**

**Remarks:**

This method moves the flagged area into its "average plane". (Plane computed using average vertex positions and normals.)

**Parameters:**

**int msl**

MNMesh-based selection level, one of **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag=MN\_SEL**

The flag that indicates which components to align. Ignored if **msl** is set to

**MNM\_SL\_OBJECT.**

**TimeValue t=0**

The current time. (This action can be used to set a key for animation.)

**Return Value:**

Indicates whether anything was moved.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnMoveToPlane(Point3 planeNormal, float  
planeOffset, int msl, DWORD flag=MN_SEL, TimeValue t=0);
```

**Remarks:**

This method moved flagged region to the plane given (by orthogonal projection into the plane).

**Parameters:**

**Point3 planeNormal**

**float planeOffset**

The definition of the plane. The plane is that region of points X for which  $\text{DotProd}(\text{planeNormal}, X) = \text{planeOffset}$ .

**int msl**

MNMesh-based selection level, one of **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag=MN\_SEL**

The flag that indicates which components to align. Ignored if **msl** is set to **MNM\_SL\_OBJECT**.

**TimeValue t=0**

The current time. (This action can be used to set a key for animation.)

**Return Value:**

Indicates whether anything was moved.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnAlignToGrid(int msl, DWORD flag=MN_SEL);
```

**Remarks:**

This method aligns flagged parts to be on the current construction plane (in the current viewport).

**Parameters:**

**int msl**

MNMesh-based selection level, one of **MNM\_SL\_OBJECT**, **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag=MN\_SEL**

The flag that indicates which components to align. Ignored if **msl** is set to **MNM\_SL\_OBJECT**.

**Return Value:**

Indicates whether anything was aligned.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnAlignToView(int msl, DWORD flag=MN_SEL);
```

**Remarks:**

This method aligns flagged parts to be at right angles to the current view (through the active viewport).

**Parameters:**

**int msl**

MNMesh-based selection level, one of **MNM\_SL\_OBJECT**, **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag=MN\_SEL**

The flag that indicates which components to align. Ignored if **msl** is set to **MNM\_SL\_OBJECT**.

**Return Value:**

Indicates whether anything was aligned.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnDeleteIsoVerts();
```

**Remarks:**

This method deletes isolated vertices, those that aren't used by any faces.

**Return Value:**

Indicates whether any vertices were deleted.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpfnMeshSmooth(int msl, DWORD flag=MN_SEL);
```

**Remarks:**

This method applies the NURMS style MeshSmooth algorithm to the flagged area of the mesh. Uses parameter block parameters:

**ep\_ms\_smoothness:** Smoothness value for MeshSmooth type subdivision (float spinner, range 0-1)

**ep\_ms\_smooth:** Currently unused.

**ep\_ms\_sep\_smooth:** Separate by smoothing groups for MeshSmooth type subdivision (checkbox)

**ep\_ms\_sep\_mat:** Separate by material IDs for MeshSmooth type subdivision (checkbox)

**Parameters:**

**int msl**

MNMesh-based selection level, one of **MNM\_SL\_OBJECT**, **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag=MN\_SEL**

The flag that indicates which components to meshsmooth. Ignored if **msl** is set to **MNM\_SL\_OBJECT**.

**Return Value:**

TRUE if applied, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EpcfTessellate (int msl, DWORD flag=MN_SEL);
```

**Remarks:**

This method tessellates flagged area of the mesh. Uses **ep\_tess\_type** and **ep\_tess\_tension** from the param block.

**Parameters:**

**int msl**

MNMesh-based selection level, one of **MNM\_SL\_OBJECT**, **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.

**DWORD flag=MN\_SEL**

The flag that indicates which components to tessellate. Ignored if **msl** is set to **MNM\_SL\_OBJECT**.

**Return Value:**

TRUE is tessellated, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void EpcfForceSubdivision();
```

**Remarks:**

This method will cue an update of the subdivision surface based on this Editable Poly. (See the Editable Poly documentation on the Object-level Surface Properties dialog for information on this surface.) Equivalent to the user pressing the "Update" button.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EpcfSelectVertByColor(BOOL add, BOOL sub, int
```

**mp=0, TimeValue t=0);**

**Remarks:**

This method selects (or deselects) vertices based on their color. Note that the color to compare to is part of the parameter block - see **ep\_vert\_sel\_color** and related parameters for details.

**Parameters:**

**BOOL add**

If TRUE, this selection should be in addition to the current selection. If FALSE, only the vertices within the color range should be selected.

**BOOL sub**

If TRUE, then instead of the options under "add" above, the vertices within the specified color range are deselected (while other selections are unmodified).

**int mp=0**

The map channel used for vertex colors. Use 0 for the traditional vertex color channel, 1-99 for a texture map channel, **MAP\_SHADING** for the vertex illumination channel, or **MAP\_ALPHA** for the vertex alpha channel.

**TimeValue t=0**

The current time.

**Default Implementation:**

{ }

**Prototype:**

**virtual void EpfSetDiagonal(int face, int corner1, int corner2);**

**Remarks:**

This method modifies a polygon's triangulation so that the specified diagonal is part of it.

**Parameters:**

**int face**

The face whose triangulation we want to modify

**int corner1, corner2**

The endpoints of the new diagonal. They should be indices into the vertex list. So for instance corner values of (0,3) would refer to vertices **f[face].vtx[0]**

and **f[face].vtx[3]**. Note that if **(corner1+1)%deg == corner2** (or vice versa), or if **corner1==corner2**, nothing will happen.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual bool EfnRetriangulate(DWORD flag=MN_SEL);
```

**Remarks:**

This method will automatically re-triangulate flagged faces, using the standard polygon triangulation algorithm. Note that this algorithm is designed to work well on a single face, but not necessarily to produce the best results on groups of faces.

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which faces should be affected.

**Return Value:**

Indicates whether anything happened. (If nothing was in fact flagged, it returns false.)

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool EfnFlipNormals(DWORD flag=MN_SEL);
```

**Remarks:**

This method flips the normals in selected elements. (Note that because of topological rules for the PolyMesh, there is no way to flip a single face - whole elements must be flipped at once.)

**Parameters:**

**DWORD flag=MN\_SEL**

Indicates which elements should be flipped. An element is considered flagged if any of its faces have that flag set.

**Return Value:**

Indicates which elements should be flipped. An element is considered flagged if any of its faces have that flag set.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void EpfnSelectByMat(int index, bool clear, TimeValue t);
```

**Remarks:**

This method selects or deselects faces by material ID.

**Parameters:**

**int index**

The material ID that indicates a face should be affected.

**bool clear**

If TRUE, these faces should have their selection cleared. (If FALSE, their selection is set.)

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EpfnSelectBySmoothGroup(DWORD bits, BOOL clear, TimeValue t);
```

**Remarks:**

This method selects or deselects faces by smoothing group.

**Parameters:**

**DWORD bits**

The smoothing group bits that indicate a face should be affected.

**BOOL clear**

If TRUE, these faces should have their selection cleared. (If FALSE, the selection is set.)

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void EfnAutoSmooth(TimeValue t);
```

**Remarks:**

This method autosmooths the current face selection, using the autosmooth threshold set in the parameter block.

**Parameters:**

**TimeValue t**

The current time.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void CollapseDeadStructs();
```

**Remarks:**

This method deletes all the components in all the levels that have the MN\_DEAD flag set. Note that this causes a renumbering of affected component levels. If you're deleting faces in the middle of a complex operation, it's usually best to just set their MN\_DEAD flags, then call **CollapseDeadStructs** at the end, so that you don't have to worry about other faces' indices changing before you're done.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int EfnPropagateComponentFlags(int slTo, DWORD flTo,  
int slFrom, DWORD flFrom, bool ampersand=FALSE, bool  
set=TRUE, bool undoable=FALSE);
```

**Remarks:**

This method is used for setting flags in the MNMesh components based on flags of other components that they touch. This is very versatile. For instance, to hide all faces that use selected vertices, you would call

**EpfNPropagateComponentFlags (MNM\_SL\_FACE, MN\_HIDE, MNM\_SL\_VERTEX, MN\_SEL, false, true, true).** (Hiding faces should be undoable.) To set the MN\_WHATEVER flag on all vertices that are used only by selected edges, you'd call **EpfNPropagateComponentFlags (MNM\_SL\_VERTEX, MN\_WHATEVER, MNM\_SL\_EDGE, MN\_SEL, true);**

**Parameters:**

**int slTo**

The selection level we wish to modify. (One of **MNM\_SL\_OBJECT**, **MNM\_SL\_VERTEX**, **MNM\_SL\_EDGE**, or **MNM\_SL\_FACE**.)

**DWORD flTo**

The flags we wish to change in that selection level.

**int slFrom**

The selection level we wish to base the changes on.

**DWORD flFrom**

The flags that indicate a change should happen.

**bool ampersand=FALSE**

When **slFrom** and **slTo** are different, this indicates whether the flags of the nearby components should be "or'd" or "and'd". If it's false, then any flagged components in the "from" level will cause the component in the "to" level to be affected. If true, then all the components in the "from" level that touch a component in the "to" level must be flagged in order for the "to" level component to be affected. (i.e., if from is faces and to is vertices, a vertex would only be modified if all faces that use it have the **flFrom** flag set.)

**bool set=TRUE**

If TRUE, this parameter indicates that the **flTo** flags should be set on targeted components. If false, it indicates that the flags should be cleared. For instance, to clear MN\_HIDE flags on vertices that are used by selected edges, you'd call **EpfNPropagateComponentFlags (MNM\_SL\_VERTEX, MN\_HIDE, MNM\_SL\_EDGE, MN\_SEL, false, false, true);**

**bool undoable=FALSE**

Indicates if this action should create an entry for the undo system. Changes to **MN\_SEL**, **MN\_HIDE**, and **MN\_DEAD** flags should generally be undoable, but changes to more minor flags like **MN\_WHATEVER** that are used to set up for other operations generally don't have to be.

**Return Value:**

The number of components in the **sITo** level that were affected by the call. (If 0, nothing happened.)

**Default Implementation:**

```
{ return 0; }
```

## Class ILookAtConstRotation

See Also: [Class Control](#), [Class FPMixinInterface](#).

class ILookAtConstRotation : public Control , public FPMixinInterface

### Description:

This class is available in release 4.0 and later only.

This class is an interface to the LookAt Constraint rotation controller. You can obtain a pointer to the list control interface using;

**GetILookAtConstInterface(cd)**. This macro will return **(ILookAtConstRotation\*)(CD)-**

**>GetFPInterface(LOOKAT\_CONSTRAINT\_INTERFACE).**

**LOOKAT\_ROT\_PBLOCK\_REF** may be used to access the look-at controller's references:

All methods of this class are Implemented by the System.

### Methods:

public:

### Prototype:

**virtual INode\* GetTarget(int targetNumber)=0;**

### Remarks:

This method will return a pointer to a node of one of the Look-At nodes that the Look-At constraint controller targets, specified by **targetNumber**.

### Parameters:

**int targetNumber**

The node number in the Look-At target list to be obtained.

### Prototype:

**virtual BOOL SetTarget(INode \*target, int targetNumber)=0;**

### Remarks:

Sets one of the Look-At nodes that the Look-At constraint controller targets, specified by **targetNumber**. If **targetNumber** is greater than the number of targets in the current list, it returns a FALSE. In this case use the function **AppendTarget**.

**Parameters:**

**INode \*node**

Points to the node to follow.

**int targetNumber**

The node number in the Look-At target list to be set.

**Return Value:**

TRUE if success; FALSE otherwise.

**Prototype:**

**virtual float GetTargetWeight(int targetNumber)=0;**

**Remarks:**

Gets the weight of one of the Look-At nodes that the Look-At constraint controller targets, specified by **targetNumber**.

**Parameters:**

**int targetNumber**

The node number in the Look-At target list whose weight is to be obtained.

**Prototype:**

**virtual BOOL SetTargetWeight(float weight, int targetNumber)=0;**

**Remarks:**

Sets the weight of one of the Look-At nodes that the Look-At constraint controller follows, specified by **targetNumber**.

**Parameters:**

**float weight**

The weight to set.

**targetNumber**

The node number in the Look-At target list whose weight is to be set.

**Return Value:**

TRUE if there is more than one Look-At targets in the list and you are trying to set weight, FALSE otherwise.

**Prototype:**

**virtual BOOL AppendTarget(INode \*target, float weight=50.0)=0;**

**Remarks:**

Appends the current Look-At target list by one and appends the current Look-At target weightlist by one.

**Parameters:**

**INode \*target**

The node that is to be appended to the current Look-At target list.

**float weight=50.0**

This is the weight that is to be assigned to the newly appended Look-At target. The default weight is 50.0.

**Return Value:**

TRUE if the target was appended, otherwise FALSE.

**Prototype:**

**virtual int GetNumTargets()=0;**

**Remarks:**

Returns the number of nodes in the list of nodes to look at.

**Prototype:**

**virtual BOOL GetRelative()=0;**

**Remarks:**

Gets the relative/absolute mode corresponding to the "Keep Initial Offset" checkbox in the UI.

**Prototype:**

**virtual BOOL GetVLisAbs()=0;**

**Remarks:**

Gets the ViewLine relative/absolute mode corresponding to the "Keep ViewLine Length Absolute" checkbox in the UI. When Viewline Length is absolute, the "ViewLine Length" spinner sets the length of the ViewLine. A negative length implies that starting from the source object the line travels opposite to the direction of the target object. The source/target distance has no

effect on the ViewLine length in this mode. If the "Keep ViewLine Length Absolute" checkbox is unchecked, the ViewLine length is determined from the spinner value, which is interpreted as a percentage of the source/target distance.

**Return Value:**

TRUE if the ViewLine length is absolute, FALSE otherwise.

**Prototype:**

**virtual void SetVLisAbs(BOOL rel)=0;**

**Remarks:**

Sets the relative/absolute mode corresponding to the "Keep ViewLine Length Absolute" checkbox in the UI.

**Parameters:**

**BOOL rel**

TRUE if "Keep ViewLine Length Absolute" is active (checked), FALSE otherwise.

**Prototype:**

**virtual BOOL GetUpnodeWorld()=0;**

**Remarks:**

Returns TRUE if the "World" checkbox is on; FALSE if off.

**Prototype:**

**virtual BOOL GetStoUPAxisFlip()=0;**

**Remarks:**

Returns TRUE if the "selected" axis flip checkbox is on; FALSE if off.

**Prototype:**

**virtual BOOL GetTargetAxisFlip()=0;**

**Remarks:**

Returns TRUE if the "source" axis flip checkbox is on; FALSE if off.

**Prototype:****virtual BOOL Get\_SetOrientation()=0;****Remarks:**

Returns TRUE if the orientation flag is set, FALSE if off.

**Prototype:****virtual int GetTargetAxis()=0;****Remarks:**

Gets the selection corresponding to the "Select LookAt Axis" button in the UI. Obtains which of the source axes is required to coincide with the target axis.

**Return Value:**

(0) if the target axis coincides with the x axis of the source object. (1) if the target axis coincides with the y axis of the source object. (2) if the target axis coincides with the z axis of the source object.

**Prototype:****virtual int GetUpNodeAxis()=0;****Remarks:**

Gets the selection corresponding to the "Source/Upnode Alignment: Aligned to UpNode Axis:" radiobutton in the UI. Obtains which of the upnode axes is required to align with a specified source axis.

**Return Value:**

(0) if the upnode x axis coincides with a specified source object. (1) if the upnode y axis coincides with a specified source object. (2) if the upnode z axis coincides with a specified source object.

**Prototype:****virtual int Get\_StoUPAxis()=0;****Remarks:**

Gets the selection corresponding to the "Source/Upnode Alignment: Aligned to UpNode Axis:" radiobutton in the UI. Obtains which of the source axes is required to align with a specified upnode axis.

**Return Value:**

(0) if the source x axis coincides with a specified upnode axis. (1) if the source y axis coincides with a specified upnode axis. (2) if the source z axis coincides with a specified upnode axis.

**Prototype:**

**virtual void SetRelative(BOOL rel)=0;**

**Remarks:**

This method allows you to set the "relative" flag.

**Parameters:**

**BOOL rel**

TRUE to set the relative flag, otherwise FALSE.

**Prototype:**

**virtual void SetUpnodeWorld(BOOL uw)=0;**

**Remarks:**

This method allows you to set the "World" flag.

**Parameters:**

**BOOL uw**

TRUE to set the world flag, otherwise false.

**Prototype:**

**virtual void SetTargetAxisFlip(BOOL rel)=0;**

**Remarks:**

This method allows you to set the "source" flip axis flag.

**Parameters:**

**BOOL rel**

TRUE to set the source flip axis flag, otherwise FALSE.

**Prototype:**

**virtual void SetStoUPAxisFlip(BOOL rel)=0;**

**Remarks:**

This method allows you to set the "selected" axis flip flag.

**Parameters:**

**BOOL rel**

TRUE to set the selected axis flip flag, otherwise FALSE.

**Prototype:**

**virtual void Set\_SetOrientation(BOOL rel)=0;**

**Remarks:**

This method allows you to set the orientation flag.

**Parameters:**

**BOOL rel**

TRUE to set the orientation flag, otherwise FALSE.

**Prototype:**

**virtual void Set\_Reset\_Orientation()=0;**

**Remarks:**

Resets to zero the amount of orientation offset, effected through the "Set Orientation" feature.

**Prototype:**

**virtual void SetTargetAxis(int axis)=0;**

**Remarks:**

Sets the selection corresponding to the "Set Orientation" button in the UI. Specifies which of the source axes is required to coincide with the target axis.

**Parameters:**

**int axis**

(0) if TargetAxis coincides with the X axis of the source object. (1) if TargetAxis coincides with the Y axis of the source object. (2) if TargetAxis coincides with the Z axis of the source object

**Prototype:**

**virtual void SetUpNodeAxis(int axis)=0;**

**Remarks:**

Sets the selection corresponding to the "Source/Upnode Alignment: Aligned to UpNode Axis:" radiobutton in the UI. Specifies which of the upnode axes is required to align with a specified source axis.

**Parameters:**

**int axis**

(0) if the upnode X axis coincides with a specified source axis. (1) if the upnode Y axis coincides with a specified source axis. (2) if the upnode Z axis coincides with a specified source axis.

**Prototype:**

**virtual void Set\_StoUPAxis(int axis)=0;**

**Remarks:**

Sets the selection corresponding to the "Source/Upnode Alignment: Aligned to UpNode Axis:" radiobutton in the UI. Specifies which of the source axes is required to align with a specified upnode axis.

**Parameters:**

**int axis**

(0) if the source X axis coincides with a specified upnode axis. (1) if the source Y axis coincides with a specified upnode axis. (2) if the source Z axis coincides with a specified upnode axis.

**Prototype:**

**virtual void DeleteTarget(int targetNumber)=0;**

**Remarks:**

This method allows you to delete a specified target.

**Parameters:**

**int targetNumber**

The zero based node number in the list of nodes the controller looks at.

## Class IOrientConstRotation

See Also: [Class Control](#), [Class FPMixinInterface](#).

```
class IOrientConstRotation : public Control, public FPMixinInterface
```

### Description:

This class is available in release 4.0 and later only.

The Orientation Constraint matches the orientation of an object to its target without affecting its position. Multiple weighted targets are supported. This class is an interface to the parameters of this controllers. You can obtain a pointer to the list control interface using; **GetIOrientConstInterface(cd)**. This macro will return **(IOrientConstRotation\*)(CD)->GetFPInterface(ORIENT\_CONSTRAINT\_INTERFACE)**.

**ORIENT\_ROT\_PBLOCK\_REF** may be used to access the orientation constraint controller's references:

All methods of this class are Implemented by the System.

### Methods:

public:

### Prototype:

```
virtual INode* GetNode(int targetNumber)=0;
```

### Remarks:

Gets one of the orientation nodes that the orientation constraint controller targets, specified by **targetNumber**.

### Parameters:

```
int targetNumber
```

The node number in the orientation target list to be obtained.

### Prototype:

```
virtual float GetTargetWeight(int targetNumber)=0;
```

### Remarks:

Gets the weight of one of the orientation nodes that the orientation constraint controller targets, specified by **targetNumber**.

**Parameters:**

**int targetNumber**

The node number in the orientation target list to set.

**Return Value:**

Returns the orientation target weight if the **targetNumber** is relevant, 0.0 otherwise.

**Prototype:**

**virtual BOOL SetTargetWeight(int targetNumber, float weight)=0;**

**Remarks:**

Sets the weight of one of the orientation nodes that the orientation constraint controller follows, specified by **targetNumber**.

**Parameters:**

**int targetNumber**

The node number in the orientation target list whose weight is to be set.

**float weight**

The weight to set.

**Return Value:**

TRUE if there is more than one orientation target in the list and you are trying to set weight, FALSE otherwise.

**Prototype:**

**virtual BOOL AppendTarget(INode \*target, float weight=50.0)=0;**

**Remarks:**

Appends the current orientation target list by one and appends the current orient\_targ\_wtlist (orientation target weightlist) by one.

**Parameters:**

**INode \*target**

The node that is to be appended to the current orientation target list.

**float weight=50.0**

This is the weight that is to be assigned to the newly appended orientation target. The default weight is 50.0.

**Return Value:**

TRUE if the target was appended, otherwise FALSE.

**Prototype:**

**virtual int GetNumTargets()=0;**

**Remarks:**

Returns the number of target nodes in the orientation target list.

**Prototype:**

**virtual BOOL DeleteTarget(int selection)=0;**

**Remarks:**

This method allows you to delete a specified target.

**Parameters:**

**int selection**

The node number in the orientation target list to delete.

**Return Value:**

TRUE if successful, otherwise FALSE.

## Class IPosConstPosition

See Also: [Class Control](#), [Class FPMixinInterface](#).

```
class IPosConstPosition : public Control, public FPMixinInterface
```

### Description:

This class is available in release 4.0 and later only.

This class represents the interface to the Position Constraint. You can obtain a pointer to the position constraint interface using;

**GetIPosConstInterface(cd)**. This macro will return

**(IPosConstPosition\*)(CD)-**

**>GetFPInterface(POS\_CONSTRAINT\_INTERFACE).**

**POSPOS\_PBLOCK\_REF** may be used to access the position constraint controller's references:

All methods of this class are Implemented by the System.

### Methods:

public:

### Prototype:

```
virtual INode* GetNode(int targetNumber)=0;
```

### Remarks:

Gets one of the position nodes that the position constraint controller targets, specified by **targetNumber**.

### Parameters:

**int targetNumber**

The node number in the position target list to be obtained.

### Prototype:

```
virtual float GetTargetWeight(int targetNumber)=0;
```

### Remarks:

Gets the weight of one of the position nodes that the position constraint controller targets, specified by **targetNumber**.

### Parameters:

### **int targetNumber**

The node number in the position target list to set.

### **Return Value:**

Returns the position target weight if the **targetNumber** is relevant, 0.0f otherwise.

### **Prototype:**

**virtual BOOL SetTargetWeight(int targetNumber, float weight)=0;**

### **Remarks:**

Sets the weight of one of the position nodes that the position constraint controller follows, specified by **targetNumber**.

### **Parameters:**

#### **int targetNumber**

The node number in the position target list whose weight is to be set.

#### **float weight**

The weight to set.

### **Return Value:**

TRUE if there is more than one position target in the list and you are trying to set weight, FALSE otherwise.

### **Prototype:**

**virtual BOOL AppendTarget(INode \*target, float weight=50.0)=0;**

### **Remarks:**

Appends the current position target list by one and appends.

### **Parameters:**

#### **INode \*target**

The node that is to be appended to the current position target list.

#### **float weight=50.0**

This is the weight that is to be assigned to the newly appended position target. The default weight is 50.0.

### **Return Value:**

TRUE if the target was appended, otherwise FALSE.

**Prototype:**

**virtual int GetNumTargets()=0;**

**Remarks:**

Returns the number of target nodes in the position target list.

**Prototype:**

**virtual BOOL DeleteTarget(int selection)=0;**

**Remarks:**

This method allows you to delete a specified target.

**Parameters:**

**int selection**

The node number in the position target list to delete.

**Return Value:**

TRUE if successful, otherwise FALSE.

# Class IPathPosition

See Also: [Class Control](#), [Class INode](#).

class IPathPosition : public Control, public FPMixinInterface

## Description:

This class is available in release 2.0 and later only.

This class represents the interface to the Path Position Controller. You can obtain a pointer to the path position controller interface using;

**GetIPathConstInterface(cd)**. This macro will return

**(IPathPosition\*)(CD)-**

**>GetFPInterface(PATH\_CONSTRAINT\_INTERFACE).**

**PATHPOS\_PATH\_REF** may be used to access the position constraint controller's references and **PATHPOS\_PBLOCK\_REF** to reference the parameter block.

All methods of this class are Implemented by the System.

## Methods:

public:

## Prototype:

**virtual int GetNumTargets()=0;**

## Remarks:

This method is available in release 4.0 and later only.

Returns the number of nodes in the path list.

## Prototype:

**virtual INode\* GetNode(int targetNumber)=0;**

## Remarks:

This method is available in release 4.0 and later only.

Gets one of the path nodes that the path controller follows, specified by **targetNumber**.

## Parameters:

### **int targetNumber**

The node number in the path list to be obtained.

#### **Prototype:**

**virtual float GetTargetWeight(int targetNumber)=0;**

#### **Remarks:**

This method is available in release 4.0 and later only.

Gets the weight of one of the path nodes that the path controller follows, specified by **targetNumber**, and time t. If the **targetNumber** is not relevant then 0.0f is returned.

#### **Parameters:**

### **int targetNumber**

The node number in the path list whose weight is to be obtained.

#### **Prototype:**

**virtual BOOL SetTargetWeight(int targetNumber, float weight)=0;**

#### **Remarks:**

This method is available in release 4.0 and later only.

Sets the weight of one of the path nodes that the path controller follows, specified by **targetNumber**.

#### **Parameters:**

### **int targetNumber**

The node number in the path list whose weight is to be set.

### **float weight**

The weight to assign.

#### **Return Value:**

TRUE if there is more than one path in the list and you are trying to set weight, FALSE otherwise.

#### **Prototype:**

**virtual BOOL AppendTarget(INode \*target, float weight=50.0)=0;**

#### **Remarks:**

This method is available in release 4.0 and later only.

Appends the current path list by one and appends the current weight list by one.

**Parameters:**

**INode \*target**

The node that is to be appended to the current path list.

**float weight=50.0**

The weight to be assigned to the newly appended path.

**Prototype:**

**virtual BOOL DeleteTarget(int selection)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to delete a specified target.

**Parameters:**

**int selection**

The node number in the orientation target list to delete.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual void SetFollow(BOOL f)=0;**

**Remarks:**

This method allows you to set the follow flag.

**Parameters:**

**BOOL f**

TRUE for on, FALSE for off.

**Prototype:**

**virtual BOOL GetFollow()=0;**

**Remarks:**

This method returns the state of the follow flag. TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetBankAmount(float a)=0;**

**Remarks:**

Sets the bank amount parameter.

Bank and tracking are scaled in the UI.

The bank values are scaled in the user interface. The following macros may be used to convert to and from the UI values.

**#define BANKSCALE 100.0f**

**#define FromBankUI(a) ((a)\*BANKSCALE)**

**#define ToBankUI(a) ((a)/BANKSCALE)**

**Parameters:**

**float a**

The bank amount.

**Prototype:**

**virtual float GetBankAmount()=0;**

**Remarks:**

Returns the bank amount setting. See the remarks in **SetBankAmount()** above.

**Prototype:**

**virtual void SetBank(BOOL b)=0;**

**Remarks:**

Sets the bank parameter to on or off.

**Parameters:**

**BOOL b**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetBank()=0;**

**Remarks:**

Returns the on/off state of the bank parameter. TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetTracking(float t)=0;**

**Remarks:**

Sets the smoothness parameter.

The smoothing (tracking) values are scaled in the user interface. The following macros may be used to convert to and from the UI values.

```
#define TRACKSCALE 0.04f
```

```
#define FromTrackUI(a) ((a)*TRACKSCALE)
```

```
#define ToTrackUI(a) ((a)/TRACKSCALE)
```

**Parameters:**

**float t**

The smoothness setting.

**Prototype:**

**virtual float GetTracking()=0;**

**Remarks:**

Returns the smoothness setting. See remarks in **SetTracking()** above.

**Prototype:**

**virtual void SetAllowFlip(BOOL f)=0;**

**Remarks:**

Sets the state of the 'Allow Upside Down' parameter.

**Parameters:**

**BOOL f**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetAllowFlip()=0;**

**Remarks:**

Returns the state of the 'Allow Upside Down' parameter.

**Return Value:**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetConstVel(BOOL cv)=0;**

**Remarks:**

Sets the state of the 'Constant Velocity' parameter.

**Parameters:**

**BOOL cv**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetConstVel()=0;**

**Remarks:**

Returns the state of the 'Constant Velocity' parameter.

**Return Value:**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetFlip(BOOL onOff)=0;**

**Remarks:**

Sets the state of the 'Flip' parameter.

**Parameters:**

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetFlip()=0;**

**Remarks:**

Returns the state of the 'Flip' parameter.

**Return Value:**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetAxis(int axis)=0;**

**Remarks:**

Set the state of the axis parameter.

**Parameters:**

**int axis**

The axis setting. One of the following values:

**0:** X axis.

**1:** Y axis.

**2:** Z axis.

**Prototype:**

**virtual int GetAxis()=0;**

**Remarks:**

Returns the axis setting.

**Return Value:**

One of the following values:

**0:** X axis.

**1:** Y axis.

**2:** Z axis.

**Prototype:**

**virtual void SetLoop(BOOL l)=0;**

**Remarks:**

This method allows you to set the state of the loop flag.

**Parameters:**

## **BOOL l**

TRUE for on; FALSE for off.

### **Prototype:**

**virtual BOOL GetLoop()=0;**

### **Remarks:**

Returns the state of the loop flag.

### **Return Value:**

TRUE for on; FALSE for off.

### **Prototype:**

**virtual void SetRelative(BOOL rel)=0;**

### **Remarks:**

This method allows you to set the state of the relative/absolute flag.

### **Parameters:**

**BOOL rel**

TRUE to set to relative; FALSE to set to absolute.

### **Prototype:**

**virtual BOOL GetRelative()=0;**

### **Remarks:**

Returns the state of the relative/absolute flag.

### **Return Value:**

TRUE if relative is on; FALSE is off (i.e. absolute).

## List of Reference Messages and their PartID parameters

See Also: [Class ReferenceTarget](#), [Class INode](#), [Class Control](#).

This section describes some of the common messages used by references and the meaning of the **PartID** parameter associated with these messages.

Some messages are sent by the system while others are sent by the plug-in. Each method may need to pass along additional information so the reference maker may process the message. This information is passed in the **PartID** parameter. The meaning of the information stored in the **PartID** is specific to the message sent along with it.

Note that not all messages use the **PartID** parameter (in fact most don't). In these cases the **PartID** will be set to 0. If the plug-in is sending the message, it should set the **PartID** to 0 if not used. In the cases where the **PartID** is used, it is documented below.

Developers who define their own reference messages should do so using a value greater than:

```
#define REFMSG_USER 0x00010000
```

The system uses numbers less than this value.

### **REFMSG\_CHANGE**

This is the most common message sent. Any time a reference target changes in a way that may affect items which reference it, this message should be sent.

Note the following for the **partIDs** that are sent during this message:

**PART\_HIDESTATE** -- This is a special **partID** sent by visibility controllers when they change the hidden in viewport state.

**PART\_TM** -- This is passed in **partID** when the reference is to a node in the scene and its transformation matrix has changed.

**PART\_OBJECT\_TYPE** -- This is sent if the object type changes.

Objects and Modifier set the **PartID** to the channel which changed. (See the section on the [Geometric Pipeline](#) for more information on channels). There are several specific **PartID** referring to channels. These are:

- **PART\_TOPO = TOPO\_CHANNEL**
- **PART\_GEOM = GEOM\_CHANNEL**

- **PART\_TEXMAP = TEXMAP\_CHANNEL**
- **PART\_MTL = MTL\_CHANNEL**
- **PART\_SELECT = SELECT\_CHANNEL**
- **PART\_SUBSEL\_TYPE = SUBSEL\_TYPE\_CHANNEL**
- **PART\_DISPLAY = DISP\_ATTRIB\_CHANNEL**
- **PART\_VERTCOLOR = VERTCOLOR\_CHANNEL**

### **REFMSG\_TARGET\_DELETED**

This message is sent when a reference target is deleted. This allows the reference maker to handle this condition if it depends on the deleted item. For example this is sent when the item you are referencing is actually deleted and you need to set your pointer to the item to NULL.

### **REFMSG\_IS\_OK\_TO\_CHANGE\_TOPOLOGY**

This message is sent to ask a reference maker if it is okay to change the topology of a node. If any dependents have made topology-dependent modifiers, they should return **REF\_FAIL**. A return of **REF\_SUCCEED** means that the answer is YES, it is okay to change the topology. A return of **REF\_FAIL** means that the answer is NO, it is not okay to change the topology.

### **REFMSG\_NODE\_LINK**

This message is sent by a node when it has a child linked to it or unlinked from it.

### **REFMSG\_NODE\_NAMECHANGE**

This message is sent by a node when its name has been changed. For example, the path controller displays the name of the node in the scene which it follows. It responds to this message by changing the name displayed in the UI.

### **REFMSG\_OBREF\_CHANGE**

This message is sent by a node (or derived object) when the object it references changes. Note: This message is no longer used. It is basically synonymous with **REFMSG\_SUBANIM\_STRUCTURE\_CHANGED**.

### **REFMSG\_MODIFIER\_ADDED**

This message is sent by a derived object when a modifier is added or deleted.

### **REFMSG\_CONTROLREF\_CHANGE**

This message is sent when an animatable switches controllers for one of its

parameters.

### **REFMSG\_GET\_PARAM\_NAME**

A parameter block sends this message to its client when it needs the name of the i-th parameter. The **PartID** is set to a pointer to a [GetParamName](#) structure. See the Advanced Topics section on [Parameter Blocks](#) for more details.

### **REFMSG\_GET\_PARAM\_DIM**

A parameter block sends this message to its client when it needs to know the dimension type of the 'i-th' parameter. The **PartID** is set to a pointer to a [GetParamDim](#) structure. See the Advanced Topics section on [Parameter Blocks](#) for more details.

### **REFMSG\_GET\_CONTROL\_DIM**

A controller can send this to its client to get its parameters dimension. It should set **PartID** to a [ParamDimension](#).

### **REFMSG\_TM\_CHANGE**

This message is sent by a node when its transformation matrix has changed because it was evaluated at a different time. Normally this isn't necessary - anyone depending on the node's TM would have a validity interval that reflected the validity of the TM. The axis system doesn't store a validity interval so this message is needed for it.

### **REFMSG\_RANGE\_CHANGE**

A node sends this message when its animation range changes.

### **REFMSG\_LINEHEIGHT\_CHANGE**

This message is sent to the track view when an animatable's line height changes.

### **REFMSG\_BECOMING\_ANIMATED**

A controller should send this message to the track view when it becomes animated. If the user has the animated only filter on then the track view will display this item.

### **REFMSG\_SUBANIM\_STRUCTURE\_CHANGED**

If a plug-in has a variable number of parameters this message may be used. This is intended mainly for the track view to tell it to re-generate the view below the message sender's level.

### **REFMSG\_REF\_DELETED**

This message is sent when a target has had a reference deleted.

## **REFMSG\_REF\_ADDED**

This message is sent when a target has had a reference added.

## **REFMSG\_BRANCHED\_HISTORY\_CHANGED**

This message is sent by an object that provides branching in the history to notify it that the structure of the branches has changed.

## **REFMSG\_NODEINSELSET\_CHANGED**

The selection set sends this notification when it receives a

**REFMSG\_CHANGE** from an item in the selection set. The selection set doesn't propagate the **REFMSG\_CHANGE** message.

## **REFMSG\_TEST\_DEPENDENCY**

This method is used to see if this reference target depends on something. In 3ds max 2.0 and later, if the **partID** is nonzero, the dependency test will include child nodes. Otherwise, child nodes will not be considered dependents. See **ReferenceTarget::BeginDependencyTest()**.

## **REFMSG\_WANT\_SHOWPARAMLEVEL**

A Parameter block sends this to its client to ask if it should display a distinct "**Parameters**" level in the track view hierarchy. A pointer to a boolean is passed in for **PartID**: set this to the desired answer. The default is NO -- in this case the message doesn't need to be responded to.

## **REFMSG\_BEFORE\_PASTE**

## **REFMSG\_NOTIFY\_PASTE**

These messages are sent before and after a paste has been done. Sent as **partID** is a pointer to a data structure containing three RefTargetHandle's: the reference maker, the old target, and the new target. The message is sent to the reference maker initially.

## **REFMSG\_UV\_SYM\_CHANGE**

This message is sent when a UV Generator changes symmetry, so interactive texture display updates.

## **REFMSG\_GET\_NODE\_NAME**

The first node that gets this message will fill in the **TSTR** which **partID** points to with its name and stop the message from propagating.

## **REFMSG\_SEL\_NODES\_DELETED**

This message is sent by the selection set whenever it has just deleted nodes.

## **REFMSG\_PRENOTIFY\_PASTE**

This message is sent **before** a reference target is pasted. It is sent by the target about to be replaced.

## **REFMSG\_SHAPE\_START\_CHANGE**

Sent when a shape enters a state where it'll be changing a lot and it would be a good idea for anybody using it for mesh generation to suppress updates.

## **REFMSG\_SHAPE\_END\_CHANGE**

Sent to terminate the above state.

## **REFMSG\_TEXMAP\_REMOVED**

A texture map has been removed. This tells the Materials Editor to remove it from the viewport if it is active.

## **REFMSG\_FLAG\_NODES\_WITH\_SEL\_DEPENDENTS**

Sent by an unselected node to see if any selected nodes depend on it. The **partID** parameter points to a boolean. If a selected node receives this message it should set the boolean to true and return **REF\_STOP**.

## **REFMSG\_CONTAINED\_SHAPE\_POS\_CHANGE**

This messages is sent by objects which contain shapes when the position changes.

## **REFMSG\_CONTAINED\_SHAPE\_SEL\_CHANGE**

This messages is sent by objects which contain shapes when the selection changes.

## **REFMSG\_CONTAINED\_SHAPE\_GENERAL\_CHANGE**

This messages is sent by objects which contain shapes when the selection, or the position changes.

## **REFMSG\_LOOPTEST**

This tests for a cyclic reference. It will return **REF\_FAIL** if there is a loop.

## **REFMSG\_BEGIN\_EDIT**

This is used by modifiers to indicate when they are beginning an edit. For example in **SimpleMod::BeginEditParams()** this message is sent.

## **REFMSG\_END\_EDIT**

This is used by modifiers to indicate when they are ending an edit. For example in **SimpleMod::EndEditParams()** this message is sent. Typically what a modifier will do while it is being edited it will have its **LocalValidity()** return **NEVER** so that a cache is built before it. This will

ensure it is more interactive while it is being edited. When this message is sent to indicate the edit is finished the system can discard the cache.

### **REFMSG\_MOD\_DISPLAY\_ON**

This is used by modifiers to indicate that their apparatus (gizmo) is displayed. For example in **SimpleMod::BeginEditParams()** this message is sent.

### **REFMSG\_MOD\_DISPLAY\_OFF**

This is used by modifiers to indicate that their apparatus (gizmo) is no longer displayed.

### **REFMSG\_MOD\_EVAL**

This is sent by a modifier to cause its ModApp to call **Eval()** on the modifier. If a modifier wants its **ModifyObject()** method to be called it can send this message.

The **PartID** should contain the bits that specify which channels are to be evaluated, for example **PART\_GEOM|PART\_TOPO** or **ALL\_CHANNELS**. The interval passed should be set to **Interval(t, t)**, where **t** is the time to evaluate. Note that before **NotifyDependents()** returns, **ModifyObject()** will be called.

### **REFMSG\_SELECT\_BRANCH**

This message is available in release 2.0 and later only.

When an object receives this message it should do whatever it needs to do (usually select the appropriate sub-object) to make the dependent object be the object returned from **Object::GetPipeBranch()**. The **partID** will point to an **Inode** pointer that will be filled in by the first node to receive this message. Thus, when an object that supports branching in the history receives this message it selects the target that sent the message.

### **REFMSG\_MOUSE\_CYCLE\_STARTED**

This message is available in release 2.0 and later only.

This message is sent to dependents of the transform controllers of selected objects when the user begins a mouse transformation in the viewports (move/rotate/scale).

### **REFMSG\_MOUSE\_CYCLE\_COMPLETED**

This message is available in release 2.0 and later only.

This message is sent to dependents of the transform controllers of selected objects when the user ends a mouse transformation in the viewports

(move/rotate/scale).

### **REFMSG\_CHECK\_FOR\_INVALID\_BIND**

This message is available in release 2.0 and later only.

Sent by a node to other nodes (which depend on that node) when the user attempts to link another node to a node. The **partID** parameter contains a pointer to the new parent node.

### **REFMSG\_OBJECT\_CACHE\_DUMPED**

This message is available in release 2.0 and later only.

Sent when a cache is dumped in the pipeline. A **REFMSG\_CHANGE** message used to be sent, however that was misleading since the object itself didn't change even though any old object pointer has become invalid. For example, if a path controller depends on a spline object and that object dumps some caches in the pipeline, the path controller hasn't actually changed.

### **REFMSG\_SFX\_CHANGE**

This message is available in release 3.0 and later only.

Sent by Atmospheric Effects or Render Effects when they make or delete a reference to a node. When Atmospherics or Effects add or delete a gizmo they should send this message via **NotifyDependents()**.

### **REFMSG\_OBJECT\_REPLACED**

This message is available in release 3.0 and later only.

Sent when objects are replaced from another scene (File->Replace). Other objects referencing the object that is replaced may want to perform some validity checking; this message is more specific than

### **REFMSG\_SUMANIM\_STRUCTURE\_CHANGED**

### **REFMSG\_NODE\_WIRECOLOR\_CHANGED**

This message is available in release 4.0 and later only.

Sent when nodes wireframe color is changed.

### **REFMSG\_NODE\_WSCACHE\_UPDATED**

This message is sent from the node (without propagation) whenever the world state cache gets updated (e.g. when the pipeline gets reevaluated).

### **REFMSG\_NODE\_HANDLE\_CHANGED**

After merging nodes into the scene, all merged objects will receive this reference notification. The PartID will be a pointer to a merge manager

interface that you can use to see if a specific handle was converted and convert between the old and the new handle; **IMergeManager\* pMergeManager = (IMergeManager\*)partID;**

### **REFMSG\_NUM\_SUBOBJECTTYPES\_CHANGED**

This message is available in release 4.0 and later only.

This message is sent to indicate that the sub-object types have changed and that the StackView should be updated.

### **REFMSG\_NODE\_MATERIAL\_CHANGED**

This message is sent from the node whenever the node material is replaced by a different material.

### **REFMSG\_SUBANIM\_NUMBER\_CHANGED**

This message is available in release 4.0 and later only.

This notification is sent to dependents when a subanim changes the ordering. It is used by things like scripted plugins and custom attributes to inform expression and wire controllers when the user redefines the ordering of parameters so these controllers can keep pointing at the correct parameter. The **PartID** is a **Tab<DWORD>\*** in which each DWORD contains an old-to-new mapping with the **LOWORD()** = old subanim number and the **HIWORD()** = new subanim number A new subanim ID of -1 implies the subanim was removed. See

**MAXSDK\SAMPLES\CONTROLLERS\EXPRCTRL.CPP** for example use.

### **REFMSG\_TAB\_ELEMENT\_NULLED**

This message is available in release 4.0 and later only.

Sent by a ParamBlock2 to its owner whenever a reftarg element in a **Tab<>** parameter is forcibly deleted and the reference set to NULL (typically for INODE\_TABs when a scene node is deleted in the viewport).

### **REFMSG\_GET\_NODE\_HANDLE**

This message is available in release 4.0 and later only.

The first node that gets this message will fill in the ULONG which partID points to with its handle and stop the message from propogating.

### **REFMSG\_END\_MODIFY\_PARAMS**

This message is available in release 4.0 and later only.

This will cause **Animatable::EndEditParams()** to be called on the object

displayed in the modify panel.

### **REFMSG\_BEGIN\_MODIFY\_PARAMS**

This message is available in release 4.0 and later only.

This will cause **Animatable::BeginEditParams()** to be called on the object displayed in the modify panel.

### **REFMSG\_MODAPP\_DELETING**

This message is used internally.

### **REFMSG\_EVAL**

This message is used internally.

### **REFMSG\_RESET\_ORIGIN**

This message is used internally.

### **REFMSG\_FLAGDEPENDENTS**

This message is used internally.

### **REFMSG\_TARGET\_SELECTIONCHANGE**

This message is used internally.

### **REFMSG\_DISABLE**

This message is used internally.

### **REFMSG\_ENABLE**

This message is used internally.

### **REFMSG\_TURNON**

This message is used internally.

### **REFMSG\_TURNOFF**

This message is used internally.

### **REFMSG\_LOOKAT\_TARGET\_DELETED**

This message is used internally.

### **REFMSG\_INVALIDATE\_IF\_BG**

This message is used internally.

### **REFMSG\_OBJXREF\_UPDITEMAT**

This message is for internal use only.

### **REFMSG\_OBJXREF\_GETNODES**

This message is for internal use only.

### **REFMSG\_NODE\_GI\_PROP\_CHANGED**

It's broadcasted when a property in the radiosity prop page changes

The following applies only to release 4.0 and later.

This section describes the messages sent by reference maker to its reference target.

Developers who define their own reference target messages should do so using a value greater than:

**#define TARGETMSG\_USER 0x00010000**

The system uses numbers less than this value.

### **TARGETMSG\_ATTACHING\_NODE**

This message is sent to a node's ObjectRef when the node is attaching the object to itself.

### **TARGETMSG\_DELETING\_NODE**

This message is sent to a node's ObjectRef when the node is about to be explicitly deleted.

### **TARGETMSG\_DETACHING\_NODE**

The message is sent to a Node's ObjectRef when the node is detaching the object from itself

## List of Miscellaneous Utility Functions

See Also: [Class Interface](#).

The following functions are not part of any class but are available as part of the API.

### Function:

```
int CoreExecute(int cmd, ULONG arg1=0, ULONG arg2=0,  
ULONG arg3=0);
```

### Remarks:

This global function is available in release 3.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

### Parameters:

**int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

### Return Value:

An integer return value (defined uniquely for each **cmd**).

### Function:

```
int MaxMsgBox(HWND hWnd, LPCTSTR lpText, LPCTSTR  
lpCaption, UINT type, UINT exType=0, DWORD *exRet=NULL);
```

### Remarks:

This global function is available in release 3.0 and later only.

This function provides an extended message box functionality. This is used to

easily support message dialogs with a 'Hold' button and/or a 'Don't Show Again' checkbox. Note that MB\_ABORTRETRYIGNORE, MB\_YESNOCANCEL, and MB\_RETRYCANCEL are not supported by this function and will be treated identically to MB\_OKCANCEL.

**Parameters:**

**HWND hWnd**

The parent window handle.

**LPCTSTR lpText**

Address of the string for the dialog body text.

**LPCTSTR lpCaption**

Address of the string for the dialog title caption.

**UINT type**

The type is the similar to the Win32 MessageBox() API.

**UINT exType=0**

The 'extended' type supports the following values (which are ORed together to create this argument):

**MAX\_MB\_HOLD**

Indicates to add a "Hold" button to the dialog.

**MAX\_MB\_DONTSHOWAGAIN**

Indicates to add a "Don't show this dialog again" checkbox to the dialog.

**DWORD \*exRet=NULL**

If non-NULL and **MAX\_MB\_DONTSHOWAGAIN** is specified above then this DWORD will have the **MAX\_MB\_DONTSHOWAGAIN** bit set if the checkbox was checked by the user.

**Return Value:**

The return value is zero if there is not enough memory to create the message box. If the function succeeds, the return value is similar to the Win32 MessageBox() API.

**Function:**

**inline float Dbl2Flt(double val, BOOL \*valid = NULL);**

**Remarks:**

This global function is available in release 3.0 and later only.

This function is defined in `\MAXSDK\INCLUDE\WINUTIL.H`. It provides safe type casting from double to float with an indication of overflow returned in the **valid** parameter.

**Parameters:**

**double val**

The value to cast.

**BOOL \*valid = NULL**

TRUE if the no overflow occurred; otherwise FALSE.

**Return Value:**

A float version of the double **val**.

**Sample Code:**

Depending on machine/compiler settings, the following code may throw an under/overflow exception or quietly return a junk value:

```
float val = (float)_tcstod(pNptr, ppEndptr);
```

A safer method would be:

```
BOOL valid;
```

```
float val = Dbl2Flt(_tcstod(pNptr, ppEndptr), &valid);
```

**Function:**

```
inline int Dbl2Int(double val, BOOL *valid = NULL);
```

**Remarks:**

This global function is available in release 3.0 and later only.

This function is defined in `\MAXSDK\INCLUDE\WINUTIL.H`. It provides safe type casting from double to int with an indication of overflow returned in the **valid** parameter.

**Parameters:**

**double val**

The value to cast.

**BOOL \*valid = NULL**

TRUE if the no overflow occurred; otherwise FALSE.

**Return Value:**

An int version of the double **val**.

**Function:**

**inline void SinCos(float angle, float \*sine, float \*cosine);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns both the sine and cosine of the angle specified, as floats. This routine uses assembly language on Intel CPUs. This is defined in **\MAXSDK\INCLUDE\GFLOAT.H**.

**Parameters:**

**float angle**

The angle in radians.

**float \*sine**

Points to storage for the output sine of the angle.

**float \*cosine**

Points to storage for the output cosine of the angle.

**Function:**

**inline float Sin(float angle);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns the sine of the angle specified as a float. This routine uses assembly language on Intel CPUs. This is defined in **\MAXSDK\INCLUDE\GFLOAT.H**.

**Parameters:**

**float angle**

The angle in radians.

**Function:**

**inline float Cos(float angle);**

**Remarks:**

This global function is available in release 3.0 and later only.

Returns the cosine of the angle specified as a float. This routine uses assembly language on Intel CPUs. This is defined in

## **\\MAXSDK\\INCLUDE\\GFLOAT.H.**

### **Parameters:**

**float angle**

The angle in radians.

### **Function:**

**inline float Sqrt(float arg);**

### **Remarks:**

This global function is available in release 3.0 and later only.

Returns the square root of the argument specified as a float. This routine uses assembly language on Intel CPUs. This is defined in

**\\MAXSDK\\INCLUDE\\GFLOAT.H.**

### **Parameters:**

**float arg**

The number whose square root is computed.

### **Function:**

**bool IsDebugging();**

### **Remarks:**

This global function is available in release 4.0 and later only.

Returns true if 3ds max is running under the debugger; otherwise false.

### **Function:**

**int NumberOfProcessors();**

### **Remarks:**

This global function is available in release 4.0 and later only.

Returns the number of processors in the machine 3ds max is running on.

### **Function:**

**bool IsWindows9x();**

### **Remarks:**

This global function is available in release 4.0 and later only.

Returns true if 3ds max is running on Windows 9x (95 or 98); otherwise false.

**Function:**

**bool IsWindows98or2000();**

**Remarks:**

This global function is available in release 4.0 and later only.

Returns true if 3ds max is running on Windows 98 or Windows 2000; otherwise false.

**Function:**

**int GetScreenWidth();**

**Remarks:**

This global function is available in release 4.0 and later only.

Returns the width of the screen (taking into consideration if multiple monitors are in use).

**Function:**

**int GetScreenHeight();**

**Remarks:**

This global function is available in release 4.0 and later only.

Returns the height of the screen (taking into consideration if multiple monitors are in use).

## Class CustAttrib

See Also: [Class ICustAttribContainer](#) , [Class ReferenceTarget](#) , [Class ParamDlg](#), [Class Animatable](#)

```
class CustAttrib: public ReferenceTarget
```

### Description:

This class is available in release 4.0 and later only.

This class represents the Custom Attributes.

A sample on how to use this class is located in

**\MAXSDK\SAMPLES\HOWTO\CUSTATTRIBUTIL**

### Methods:

public:

### Prototype:

```
virtual TCHAR* GetName();
```

### Remarks:

Implemented by the plugin.

A CustAttrib plugin can implement this method in order to provide the name that gets displayed in the TrackView.

### Default Implementation:

```
{ return "Custom Attribute";}
```

### Prototype:

```
virtual ParamDlg *CreateParamDlg(HWND hwMtlEdit,  
IMtlParams *imp);
```

### Remarks:

Implemented by the plugin.

This method gets called when the material or texture is to be displayed in the material editor parameters area. The plug-in should allocate a new instance of a class derived from ParamDlg to manage the user interface.

### Parameters:

**HWND hwMtlEdit**

The window handle of the materials editor.

### **IMtlParams \*imp**

The interface pointer for calling methods in 3ds max.

#### **Return Value:**

A pointer to the created instance of a class derived from **ParamDlg**.

#### **Default Implementation:**

```
{return NULL;}
```

#### **Prototype:**

```
virtual bool CheckCopyAttribTo(ICustAttribContainer *to);
```

#### **Remarks:**

Implemented by the plugin.

This method will check if it possible to copy the current custom attributes to the specified custom attributes container.

#### **Parameters:**

**ICustAttribContainer \*to**

A pointer to the custom attributes container you wish to check for possible reception of the custom attributes..

#### **Return Value:**

TRUE if it is possible to copy, otherwise FALSE.

#### **Default Implementation:**

```
{ return true; }
```

## Class ICollision

See Also: [Class CollisionOps](#), [Class CollisionPlane](#), [Class CollisionSphere](#), [Class CollisionVNormal](#), [Class CollisionMesh](#), [Class ReferenceTarget](#), [Class Box3](#), [Class Point3](#)

```
class ICollision : public ReferenceTarget
```

### Description:

This class is available in release 4.0 and later only.

This class represents the virtual class to support collision objects. All collision detection classes should derive from this class. The purpose of this class is to improve the particle collision system for older particle systems and to let other systems such as Flex and MAXScript benefit from a more open architecture and to provide an interface to determine if a particle hit a surface. Additionally three basic collision detection classes are provided, a planar, a spherical, and a mesh deflection class.

### Methods:

public:

### Prototype:

```
virtual int SupportedCollisions() = 0;
```

### Remarks:

This method returns the collision type supported by the engine.

### Return Value:

One of the following;

**POINT\_COLLISION** for point collision, currently supported.

**SPHERE\_COLLISION** for spherical collision, currently not supported.

**BOX\_COLLISION** for box collision, currently not supported.

**EDGE\_COLLISION** for edge collision, currently not supported.

### Prototype:

```
virtual void PreFrame(TimeValue t, TimeValue dt) = 0;
```

### Remarks:

This method will be called once before the checkcollision is called for each frame which allows you to do any required initialization.

**Parameters:**

**TimeValue t**

The time at which to initialize.

**TimeValue dt**

The delta of time the particle wil travel.

**Prototype:**

**virtual void PostFrame(TimeValue t, TimeValue dt) = 0;**

**Remarks:**

This method will be called at the end of each frame solve to allow you to destroy and deallocate any data you no longer need.

**Parameters:**

**TimeValue t**

The time at which to initialize.

**TimeValue dt**

The delta of time the particle wil travel.

**Prototype:**

**virtual BOOL CheckCollision (TimeValue t, Point3 pos, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;**

**Remarks:**

This method will be called to execute a point to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**TimeValue t**

The end time of the particle.

**Point3 pos**

The position of the particle in world space.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

**virtual BOOL CheckCollision (TimeValue t,Point3 pos, float radius, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;;**

**Remarks:**

This method will be called to execute a sphere to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**TimeValue t**

The end time of the particle.

**Point3 pos**

The position of the particle in world space.

**float radius**

The radius of the sphere.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

**virtual BOOL CheckCollision (TimeValue t, Box3 box, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;**

**Remarks:**

This method will be called to execute a box to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**TimeValue t**

The end time of the particle.

**Box3 box**

The box itself.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

**virtual BOOL CheckCollision (TimeValue t,Point3 edgeA,Point3 edgeB ,Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;**

**Remarks:**

This method will be called to execute an edge to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**TimeValue t**

The end time of the particle.

**Point3 edgeA**

The first edge.

**Point3 edgeB**

The second edge.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

## Class RandGenerator

class RandGenerator

### Description:

This class is available in release 4.0 and later only.

This class has interfaces for **srand()** and **rand()** methods of VC++ and other functions for random number generation. The **srand()** and **rand()** methods from **stdlib.h** have two main problems:

a) It's not satisfactorily random. The **rand()** function returns a pseudorandom integer in the range 0 to 0x7fff=32767. If we need a lot of random numbers using **rand()** (i.e. for generating 100,000 particles), we run out of continuity of random numbers. Generated random numbers becomes too discrete.

b) The **rand()** method is global function, not class object. Hence it is shared between all modules of your plug-in. Changes in one module may change randomness pattern in other independent module. To solve this contradiction, rand methods have to be implemented as a class object.

The **RandGenerator** does exactly that. It has much more random numbers:  $RAND\_MAX = 0xFFFFFFFF = 4,294,967,295$ . Also, using instances of the class, it's much easier to create separate threads of random numbers for a specific module.

### Data Members:

public:

**static const DWORD32 RAND\_MAX**

This definition is used to override the VC++ rand methods.

### Methods:

public:

### Prototype:

**RandGenerator();**

### Remarks:

Constructor.

**Prototype:**

**void srand(DWORD32 seed);**

**Remarks:**

This method sets the starting point for generating a series of pseudorandom integers. To reinitialize the generator, use 1 as the seed argument. Any other value for seed sets the generator to a random starting point. **rand()** retrieves the pseudorandom numbers that are generated. Calling **rand()** before any call to this method generates the same sequence as calling it with seed passed as 1.

**Parameters:**

**DWORD32 seed**

The starting seed.

**Prototype:**

**DWORD32 rand(void);**

**Remarks:**

This method returns a pseudorandom integer in the range 0 to RAND\_MAX

**Prototype:**

**int RandSign(void);**

**Remarks:**

This method returns the random number sign, either -1 or 1.

**Prototype:**

**float Rand01(void);**

**Remarks:**

This method return a random number between 0.0f and 1.0f.

**Prototype:**

**float Rand11(void);**

**Remarks:**

This method return a random number between -1.0f and 1.0f.

**Prototype:**

**float Rand55(void);**

**Remarks:**

This method return a random number between -0.5f and 0.5f.

**Prototype:**

**int Rand0X(void);**

**Remarks:**

This method return a random number between 0 and maxnum.

**Prototype:**

**bool Valid(void) const;**

**Remarks:**

This method returns TRUE if the random number generator has been explicitly initialized by the **srand()** method.

## Class ID3DGraphicsWindow

See Also: [Class IDX8VertexShader](#), [Class IDX8PixelShader](#), [Class BaseInterface](#)

```
class ID3DGraphicsWindow : public BaseInterface
```

### Description:

This class is available in release 4.0 and later only.

The abstract interface to the D3D graphics window class. The methods here provide low-level access to 3ds max's Direct-3D graphics system. These methods are available for plug-ins to do any graphics work not possible using the standard high-level graphics methods of 3ds max. These methods are for use in the existing 3ds max viewports. Note that these APIs are not for casual use, as they are not intended to be a high level graphics library. A number of structures, classes, and data types used here are part of the Microsoft DirectX SDK.

### Methods:

public:

### Prototype:

```
virtual Interface_ID GetID();
```

### Remarks:

This method returns the interface ID of the class.

### Default Implementation:

```
{ return D3D_GRAPHICS_WINDOW_INTERFACE_ID; }
```

### Prototype:

```
virtual LifetimeType LifetimeControl();
```

### Remarks:

This method returns the type of lifetime control for this interface.

### Default Implementation:

```
{ return noRelease; }
```

### Prototype:

**virtual LPDIRECT3DDEVICE8 GetDevice() = 0;**

**Remarks:**

This method returns the Direct3D Device from GFX.

**Prototype:**

**virtual LPDIRECT3DVERTEXBUFFER8 GetVertexBuffer(UINT length, DWORD FVF) = 0;**

**Remarks:**

This method returns the VertexBuffer from GFX.

**Parameters:**

**UINT length**

The size of vertex buffer.

**DWORD FVF**

The Flexible Vertex Format. Unless older Flexible Vertex Formats are in use, FVF should be zero.

**Prototype:**

**virtual LPDIRECT3DINDEXBUFFER8 GetIndexBuffer(UINT length, D3DFORMAT format) = 0;**

**Remarks:**

This method returns the Direct-3D index buffer from GFX.

**Parameters:**

**UINT length**

The size of vertex buffer.

**D3DFORMAT format**

The Direct-3D format to use.

**Prototype:**

**virtual D3DXMATRIX GetWorldXform() = 0;**

**Remarks:**

This method returns the World transformation from GFX as a Direct-3D Matrix.

**Prototype:**

**virtual D3DXMATRIX GetViewXform() = 0;**

**Remarks:**

This method returns the View transformation from GFX as a Direct-3D Matrix.

**Prototype:**

**virtual D3DXMATRIX GetProjXform() = 0;**

**Remarks:**

This method returns the Projection transformation from GFX as a Direct-3D Matrix.

**Prototype:**

**virtual D3DCOLOR GetColor(ColorType t) = 0;**

**Remarks:**

This method returns the Constant Color of specified type from GFX.

**Parameters:**

**ColorType t**

The type of color you wish to retrieve.

**Prototype:**

**virtual Tab<D3DLIGHT8 \*> \*GetLights() = 0;**

**Remarks:**

This method returns a pointer to a table of pointers to enabled Direct3D Lights from GFX

**Prototype:**

**virtual D3DMATERIAL8 GetMaterial() = 0;**

**Remarks:**

This method returns the Direct-3D Material from GFX.

**Prototype:**

**virtual DWORD GetTextureTiling(int texStage, int coord) = 0;**

**Remarks:**

This method returns the Texture Tiling for specified texStage and coord from GFX.

**Parameters:**

**int texStage**

The specified texture stage to get the tiling for.

**int coord**

The specified texture coordinate to get the tiling for.

**Prototype:**

**virtual D3DXMATRIX GetTexXform(int texStage) = 0;**

**Remarks:**

This method returns the Texture Transform for specified texStage from GFX.

**Parameters:**

**int texStage**

The specified texture stage to get the texture transformation for.

## Class IDX8PixelShader

See Also: [Class ID3DGraphicsWindow](#), [Class IDX8VertexShader](#), [Class BaseInterface](#), [Class Material](#), [Class INode](#)

class IDX8PixelShader: public BaseInterface

### Description:

This class is available in release 4.0 and later only.

The abstract interface to the Direct-3D Pixel Shader architecture.

### Methods:

public:

### Prototype:

```
virtual Interface_ID GetID();
```

### Remarks:

This method returns the interface ID of the class.

### Default Implementation:

```
{ return DX8_PIXEL_SHADER_INTERFACE_ID; }
```

### Prototype:

```
virtual HRESULT ConfirmDevice(ID3DGraphicsWindow *gw) = 0;
```

### Remarks:

This method will confirm that the Direct3D Device can handle this PixelShader.

### Parameters:

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

### Prototype:

```
virtual HRESULT ConfirmVertexShader(IDX8VertexShader *pvs) = 0;
```

**Remarks:**

This method will confirm that an associated VertexShader will work with this PixelShader.

**Parameters:**

**IDX8VertexShader \*pvs**

A pointer to the vertex shader to check for.

**Prototype:**

**virtual HRESULT Initialize(Material \*mtl, INode \*node) = 0;**

**Remarks:**

This method will load the PixelShader instructions and textures. PixelShader instructions should be loaded once and shared among all the nodes using this PixelShader. In addition, any textures necessary for the PixelShader effect should be loaded once and shared among all the nodes using this PixelShader.

**Parameters:**

**Material \*mtl**

A pointer to the pixel shader material.

**INode \*node**

A pointer to the node.

**Prototype:**

**virtual int GetNumMultiPass() = 0;**

**Remarks:**

This method returns the number of passes for the effect this PixelShader creates. Note that this value will depend on the hardware currently in use.

**Prototype:**

**virtual DWORD GetPixelShaderHandle(int numPass) = 0;**

**Remarks:**

This method returns the PixelShader handle for the specified pass for use in GFX.

**Parameters:**

**int numPass**

The pass for which to return the pixelshader handle.

**Prototype:**

**virtual HRESULT SetPixelShader(ID3DGraphicsWindow \*gw, int numPass) = 0;**

**Remarks:**

This method allows you to set the PixelShader for the specified pass. This call will be made at least once per object to set the per object data for the PixelShader such as the PixelShader constants.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**int numPass**

The pass for which to set the pixel shader.

## Class IDX8VertexShader

See Also: [Class IVertexShader](#), [Class IDX8PixelShader](#), [Class ID3DGraphicsWindow](#) , [Class BaseInterface](#)

```
class IDX8VertexShader : virtual public IVertexShader, public BaseInterface
```

### Description:

This class is available in release 4.0 and later only.

The abstract interface to the Direct-3D Vertex Shader architecture.

The drawing functions are necessary as something other than a simple default body if:

- · The VertexShader needs to add additional per vertex data unknown to the Mesh to the VertexBuffer.
- · The VertexShader needs to have per vertex data ordered differently than the standard position, normal, {color, tex coords ordering}.
- · The VertexShader is being used to create cached VertexBuffers or using higher order surfaces.

In the cases of **DrawMeshStrips()** and **DrawWireMesh()**, the VertexShader has the option of not only locking and filling the VertexBuffer with data, but also of making the actual DrawPrimitive call. In the case of **StartLines()**, the VertexShader must make the DrawPrimitive call. The VertexShader indicates that it has done the drawing by returning 'true' in the Draw functions provided.

In the case where the VertexShader does not want to do the drawing but does want to fill in a VertexBuffer with data, the VertexShader can request the GFX to create a VertexBuffer (and possibly an IndexBuffer) of appropriate size. The GetVertexBuffer and GetIndexBuffer calls on the ID3DGraphicsWindow object will do this and return the allocated buffers in subsequent calls or reallocate them if necessary.

Please note that if a PixelShader or PixelShaders are in use, these Draw functions may need to set them for the appropriate passes of a multipass rendering if the drawing is done in these Draw() functions. If the GFX is doing the drawing, then these Draw() functions are only being used to fill in the VertexBuffer with data; the GFX will be doing the drawing and will be setting

the PixelShaders as appropriate.

**Methods:**

public:

**Prototype:**

```
virtual Interface_ID GetID();
```

**Remarks:**

This method returns the interface ID of the class.

**Default Implementation:**

```
{ return DX8_VERTEX_SHADER_INTERFACE_ID; }
```

**Prototype:**

```
virtual HRESULT ConfirmDevice(ID3DGraphicsWindow *gw) =  
0;
```

**Remarks:**

This method will confirm that the Direct3D Device can handle this VertexShader.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**Prototype:**

```
virtual HRESULT ConfirmPixelShader(IDX8PixelShader *pps) =  
0;
```

**Remarks:**

This method will confirm that an associated PixelShader will work with this VertexShader.

**Parameters:**

**IDX8PixelShader \*pps**

A pointer to the pixel shader to confirm for.

**Prototype:****virtual bool CanTryStrips() = 0;****Remarks:**

This method will indicate if it can try tristrips for drawing or must geometry using this VertexShader be drawn as triangles? This should return TRUE unless additional per vertex data is generated by this VertexShader and this data does not map to the Mesh vertices in the same way as existing data the Mesh knows about such as texture coordinates.

**Prototype:****virtual int GetNumMultiPass() = 0;****Remarks:**

This method returns the number of passes for the effect this VertexShader creates. Note that this value will depend on the hardware currently in use.

**Prototype:****virtual DWORD GetVertexShaderHandle(int numPass) = 0;****Remarks:**

This method returns the VertexShader handle for the specified pass for use in GFX.

**Prototype:****virtual HRESULT SetVertexShader(ID3DGraphicsWindow \*gw, int numPass) = 0;****Remarks:**

This method allows you to set the VertexShader for the specified pass. This call will be made at least once per object to set the per object data for the VertexShader such as the VertexShader constants.

**Parameters:****ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**int numPass**

The pass for which to set the vertex shader.

**Prototype:**

```
virtual bool DrawMeshStrips(ID3DGraphicsWindow *gw,  
MeshData *data) = 0;
```

**Remarks:**

This method will draw the 3D Mesh as TriStrips. Fill in the VertexBuffer with data in the order desired by the VertexShader.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**MeshData \*data**

A pointer to the mesh data.

**Return Value:**

TRUE if the Mesh has actually been drawn in this call, FALSE if the GFX is required to make the DrawPrimitive call.

**Prototype:**

```
virtual bool DrawWireMesh(ID3DGraphicsWindow *gw,  
WireMeshData *data) = 0;
```

**Remarks:**

This method will draw the 3D Mesh as wireframe. Fill in the VertexBuffer with data in the order desired by the VertexShader.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**WireMeshData \*data**

A pointer to the wire mesh data.

**Return Value:**

TRUE if the Mesh has actually been drawn in this call, FALSE if the GFX is required to make the DrawPrimitive call.

**Prototype:**

```
virtual void StartLines(ID3DGraphicsWindow *gw,
```

**WireMeshData \*data) = 0;**

**Remarks:**

This method will draw 3D lines. A Mesh is being drawn by having line segments handed down one at a time. Pass in the Mesh data in preparation for drawing 3D lines.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**WireMeshData \*data**

A pointer to the wire mesh data.

**Prototype:**

**virtual void AddLine(ID3DGraphicsWindow \*gw, DWORD \*vert, int vis) = 0;**

**Remarks:**

This method will draw 3D lines. A Mesh is being drawn by having line segments handed down one at a time. Add the connectivity information for one two point line segment.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**DWORD \*vert**

The array of vertices.

**int vis**

The visibility flag.

**Prototype:**

**virtual bool DrawLines(ID3DGraphicsWindow \*gw) = 0;**

**Remarks:**

This method will draw the line segments accumulated. This should restart the filling of a VertexBuffer with the next AddLine call if additional data needs to

be drawn before EndLines is called.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**Return Value:**

TRUE if the Mesh line segments have actually been drawn in this call, FALSE if the GFX is required to make the DrawPrimitive call.

**Prototype:**

```
virtual void EndLines(ID3DGraphicsWindow *gw,  
GFX_ESCAPE_FN fn) = 0;
```

**Remarks:**

This method will let the Mesh know that all drawing and data access is finished.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**GFX\_ESCAPE\_FN fn**

The graphics escape function.

**Prototype:**

```
virtual void StartTriangles(ID3DGraphicsWindow *gw,  
MeshFaceData *data) = 0;
```

**Remarks:**

This method will Draw 3D triangles. A Mesh is being drawn by having triangles handed down one at a time. Pass in the Mesh data in preparation for drawing 3D triangles.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**MeshFaceData \*data**

A pointer to the mesh face data.

**Prototype:**

```
virtual void AddTriangle(ID3DGraphicsWindow *gw, DWORD  
index, int *edgeVis) = 0;
```

**Remarks:**

This method will Draw 3D triangles. A Mesh is being drawn by having triangles handed down one at a time. Add the connectivity information for one triangle.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**DWORD index**

The triangle index.

**int \*edgeVis**

The array of edge visibility information/

**Prototype:**

```
virtual bool DrawTriangles(ID3DGraphicsWindow *gw) = 0;
```

**Remarks:**

This method will draw the triangles accumulated. This should restart the filling of a VertexBuffer with the next AddTriangle call if additional data needs to be drawn before EndTriangles is called. Return 'true' if the Mesh triangles have actually been drawn in this call, 'false' if the GFX is required to make the DrawPrimitive call.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**Prototype:**

```
virtual void EndTriangles(ID3DGraphicsWindow *gw,  
GFX_ESCAPE_FN fn) = 0;
```

**Remarks:**

This method will let the Mesh know that all drawing and data access is

finished.

**Parameters:**

**ID3DGraphicsWindow \*gw**

A pointer to the Direct-3D Graphics Window.

**GFX\_ESCAPE\_FN fn**

The graphics escape function.

# Class PatchTVert

See Also: [Class PatchMesh](#), [Template Class Tab](#), [Working with Patches](#).

## Description:

This class is available in release 4.0 and later only.

This class stores the texture vertex information associated with a patch and provides methods to access this. All methods of this class are implemented by the system.

## Data Members:

public:

**UVVert p;**

The texture vertex location.

**int aux1;**

Used to track topology changes during editing (Edit Patch).

## Methods:

### Prototype:

**PatchVert();**

### Remarks:

Constructor. The UVVert is set to 0,0,0. The aux member is set to -1.

### Prototype:

**PatchTVert(float u, float v, float w);**

### Remarks:

Constructor. The UVVert is set to the provided uvw parameters passed to the method. The aux member is set to -1.

### Parameters:

**float u, float v, float w;**

The u, v, and w values for the texture vertex.

## Operators:

### Prototype:

**PatchTVert& operator=(cont UVVert& from);**

**Remarks:**

Assignment operator.

**Parameters:**

**UVVert& from**

The texture vertex to copy from.

**Prototype:**

**operator UVVert&()**

**Remarks:**

Conversion operator. Returns a reference to UVVert p.

## Class ActionTable

See Also: [Class BaseInterfaceServer](#), [Class ClassDesc](#), [Structure ActionDescription](#), [Class ActionItem](#), [Class ActionCallback](#), [Class ActionContext](#), [Class IActionManager](#), [Class DynamicMenu](#), [Class DynamicMenuCallback](#), [Class Interface](#).

```
class ActionTable : public BaseInterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This is the class used to create Action Tables. An ActionTable holds a set of ActionItems, which are operations that can be tied to various UI elements, such as keyboard shortcuts, CUI buttons, the main menu and the Quad menu. 3ds max's core code exports several ActionTables for built-in operations in 3ds max. Plug-ins can also export their own action tables via methods available in ClassDesc.

All methods of this class are implemented by the system. Note however that many methods are virtual and may be customized by the plug-in developer as this class may be sub-classed if required. See the Advanced Topics section [UI Customization](#) for details on sub-classing this class and ActionItem. For details on implementing an ActionTable please refer to `\MAXSDK\SAMPLESMODIFIERS\FFD`

### Methods:

```
public:
```

### Prototype:

```
ActionTable(ActionTableId id, ActionContextId contextId,  
TSTR& name, HACCEL hDefaults, int numIds,  
ActionDescription* pOps, HINSTANCE hInst);
```

### Remarks:

Constructor. This constructor builds the action table using an array of descriptors. It takes the ID of the table, the context id, a name for the table, a windows accelerator table that gives default keyboard assignments for the operations, the number of items, the table of operation descriptions, and the instance of the module where the string resources in the table are stored.

At the same time the action table is built developers need to register the action

context ID with the system. This is done using the **IActionManager::RegisterActionContext()** method.

**Parameters:**

**ActionTableId id**

The unique ID for the ActionTable. Every ActionTable has a unique 32-bit integer ID. For new tables exported by plug-ins, the developer should choose a random 32-bit integer. You can use the Class\_ID program to generate this identifier: See [Class Class\\_ID](#) for more details. Simply use one of the two DWORDs that comprise the Class\_ID for the ActionTableId.

**ActionContextId contextId**

The ActionContextID associated with this table. Several tables may share the same ActionContextID.

**TSTR& name**

The name for the ActionTable.

**HACCEL hDefaults**

The handle of the a windows accelerator table that gives default keyboard assignments for the operations.

**int numIds**

The number of items in the description array below.

**ActionDescription\* pOps**

Points to the array of the operator descriptors.

**HINSTANCE hInst**

The handle to the instance of the module where the string resources in the array of operator descriptors are stored.

**Prototype:**

**ActionTable(ActionTableId id, ActionContextId contextId,  
TSTR& name);**

**Remarks:**

Constructor. This constructor build a new empty action table with the given ID, context ID and name. You then need to add ActionItems to the table separately using the **AppendOperation()** method described below.

**Parameters:**

**ActionTableId id**

The unique ID for the ActionTable.

**ActionContextId contextId**

The ActionContextID associated with this table. Several tables may share the same ActionContextID.

**TSTR& name**

The name for the ActionTable.

**Prototype:**

**~ActionTable();**

**Remarks:**

Destructor. Deletes all the operations maintained by the table and deletes the keyboard accelerator table if in use.

**Prototype:**

**HACCEL GetHAccel();**

**Remarks:**

Returns the handle of the current keyboard accelerator for the table.

**Prototype:**

**void SetHAccel(HACCEL hAccel);**

**Remarks:**

Sets the current keyboard accelerator for the table.

**Prototype:**

**HACCEL GetDefaultHAccel();**

**Remarks:**

Get the default keyboard accelerator table. This is used when the user has not assigned any accelerators.

**Prototype:**

**TSTR& GetName();**

**Remarks:**

Returns the name of the ActionTable. This is the name in the Customize UI dialog drop-down.

**Prototype:**

**ActionTableId GetId();**

**Remarks:**

Returns the ActionTableId for this ActionTable.

**Prototype:**

**ActionContextId GetContextId();**

**Remarks:**

Returns the ActionContextId for this ActionTable.

**Prototype:**

**ActionCallback\* GetCallback();**

**Remarks:**

Get the current callback associated with this table. Returns NULL if the table is not active.

**Prototype:**

**void SetCallback(ActionCallback\* pCallback);**

**Remarks:**

Sets the callback object used by this ActionTable.

**Parameters:**

**ActionCallback\* pCallback**

Points to the callback to set.

**Prototype:**

**int Count();**

**Remarks:**

Returns the number of ActionItems in the table.

**Prototype:**

**ActionItem\* operator[](int i);**

**Remarks:**

This operator returns a pointer to the 'i-th' ActionItem.

**Parameters:**

**int i**

The zero based index in the list of ActionItems.

**Prototype:**

**ActionItem\* GetAction(int cmdId);**

**Remarks:**

Returns a pointer to the ActionItem associated with the command ID passed.

**Parameters:**

**int cmdId**

The command ID.

**Prototype:**

**void AppendOperation(ActionItem\* pAction);**

**Remarks:**

This method adds an operation to the table.

**Parameters:**

**ActionItem\* pAction**

Points to the ActionItem to append.

**Prototype:**

**BOOL DeleteOperation(ActionItem\* pAction);**

**Remarks:**

Remove an operation from the table

**Parameters:**

**ActionItem\* pAction**

Points to the ActionItem to delete.

**Return Value:**

TRUE if the operation was deleted; FALSE if it could not be found and wasn't.

**Prototype:**

```
void DeleteThis();
```

**Remarks:**

Deletes this ActionItem.

**Default Implementation:**

```
{ delete this; }
```

**Prototype:**

```
virtual BOOL GetButtonText(int cmdId, TSTR& buttonText);
```

**Remarks:**

This method retrieves the text that will be used when the ActionItem is on a text button.

**Parameters:**

**int cmdId**

The unique ID of the command whose button text is retrieved.

**TSTR& buttonText**

Storage for the text.

**Return Value:**

TRUE if the command is in the table; otherwise FALSE.

**Prototype:**

```
virtual BOOL GetMenuText(int cmdId, TSTR& menuText);
```

**Remarks:**

This method retrieves the text to use when the item is on a menu (either Quad menu or main menu bar). This can be different from the button text.

**Parameters:**

**int cmdId**

The unique ID of the command whose menu text is retrieved.

**TSTR& menuText**

Storage for the text.

**Return Value:**

TRUE if the command is in the table; otherwise FALSE.

**Default Implementation:**

```
{ return GetButtonText(cmdId, menuText); }
```

**Prototype:**

```
virtual BOOL GetDescriptionText(int cmdId, TSTR& descText);
```

**Remarks:**

This method gets the text that will be used for tool tips and menu help. This is also the string that is displayed for the operation in all the lists in the customization dialogs.

**Parameters:**

**int cmdId**

The unique ID of the command whose description text is retrieved.

**TSTR& descText**

Storage for the text.

**Return Value:**

TRUE if the command is in the table; otherwise FALSE.

**Prototype:**

```
virtual BOOL IsChecked(int cmdId);
```

**Remarks:**

Returns TRUE if the menu item should be checked or a CUI button should be in the pressed state.

**Parameters:**

**int cmdId**

The unique ID of the command.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual BOOL IsItemVisible(int cmdId);**

**Remarks:**

This method determines if an item is to be visible on a menu. Returns TRUE if visible; FALSE if not.

**Parameters:**

**int cmdId**

The unique ID of the command.

**Default Implementation:**

**{ return TRUE; }**

**Prototype:**

**virtual BOOL IsEnabled(int cmdId);**

**Remarks:**

This method determines if the operation is currently enabled and available. Returns TRUE if enabled; FALSE if disabled.

**Parameters:**

**int cmdId**

The unique ID of the command.

**Default Implementation:**

**{ return TRUE; }**

**Prototype:**

**virtual void WritePersistentActionId(int cmdId, TSTR& idString);**

**Remarks:**

This method will write an action identifier to a \*.CUI file or \*.KBD file. It's default implementation is to write the integer ID but will be over-riden when command IDs are not persistent.

**Parameters:**

**int cmdId**

The unique ID of the command.

**TSTR& idString**

The action ID placed in the string.

**Prototype:**

**virtual int ReadPersistentActionId(TSTR& idString);**

**Remarks:**

This method will read an action identifier from a \*.CUI file or \*.KBD file. It's default implementation is to read the integer ID but will be over-riden when command IDs are not persistent.

**Parameters:**

**TSTR& idString**

The action ID string.

**Return Value:**

This method returns -1 if the command was not found in the table.

**Prototype:**

**virtual MaxIcon\* GetIcon(int cmdId);**

**Remarks:**

Returns an optional icon for the command, or NULL if there is none.

**Parameters:**

**int cmdID**

The unique ID of the command.

**Prototype:**

**void BuildActionTable(HACCEL hDefaults, int numIds,  
ActionDescription\* pOps, HINSTANCE hInst);**

**Remarks:**

This method will fill the action table with the given action descriptions.

**Parameters:**

**HACCEL hDefaults**

The handle of the a windows accelerator table that provides keyboard assignments for the operations.

**int numIds**

The number of ID's to add to the action table.

**ActionDescription\* pOps**

The array of action descriptions to build the table from.

**HINSTANCE hInst**

The handle to the instance of the module.

**Prototype:**

**ActionItem\* GetCurrentAssignment(ACCEL accel);**

**Remarks:**

Get the action assigned to the given accelerator, if any.

**Parameters:**

**ACCEL accel**

The accelerator key you wish to check the assignment for.

**Prototype:**

**void AssignKey(int cmdId, ACCEL accel);**

**Remarks:**

Assign the command to the given accelerator. Also removes any previous assignment to that accelerator.

**Parameters:**

**int cmdId**

The command ID.

**ACCEL accel**

The accelerator key you wish to assign.

**Prototype:**

**void RemoveShortcutFromTable(ACCEL accel);**

**Remarks:**

removes the given assignment from the shortcut table

**Parameters:**

**ACCEL accel**

The accelerator key you wish to remove from the shortcut table.

## Class GBufReader

See Also: [Class GBuffer](#), [Class GBufWriter](#), [Structure GBufData](#), [List of GBuffer Channels Indexes](#).

```
class GBufReader : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This is the object returned by **GBuffer::CreateReader()**. Methods of this class allow the G-Buffer data to be read.

Here is an example of reading multiple layer data from the G-Buffer using methods of this class.

```
void ReadExample(GBuffer *gb) {  
    float zr, UBYTE midr;  
    GBufReader *rdr = gb->CreateReader();  
    for (int y=0; y<10; y++) {  
        rdr->StartLine(y);  
        for (int x=5; x<100; x+=4) {  
            int res = rdr->StartPixel(x);  
            rdr->ReadChannelData(GB_Z,(void *)&zr);  
            rdr->ReadChannelData(GB_MTL_ID,(void *)&midr);  
            while (rdr->StartNextLayer()) {  
                rdr->ReadChannelData(GB_Z,(void *)&zr);  
                rdr->ReadChannelData(GB_MTL_ID,(void *)&midr);  
            }  
        }  
    }  
    gb->DestroyReader(rdr);  
}
```

All methods of this class are implemented by the System.

### Methods:

public:

### Prototype:

**virtual int StartLine(int y)=0;**

**Remarks:**

Call this method to start a new scan line. Call this method before the first scan line.

**Parameters:**

**int y**

The zero based index of the scan line to start.

**Return Value:**

Returns -1 if there was no data for line, or the x value of first non-empty pixel.

**Prototype:**

**virtual BOOL StartPixel(int x)=0;**

**Remarks:**

Call this method to start a new pixel. This method automatically starts the first layer.

**Parameters:**

**int x**

The zero based index of the pixel to start.

**Return Value:**

Returns TRUE.

**Prototype:**

**virtual BOOL StartPixel(int x, int y)=0;**

**Remarks:**

This method is called to start a new line and pixel. This method is equivalent to:

**StartLine(y);**

**return StartPixel(x);**

**Prototype:**

**virtual BOOL StartNextLayer()=0;**

**Remarks:**

This method is called to begin reading data from a new layer. Do **not** call this method before reading the first layer.

**Return Value:**

TRUE if more data to read; otherwise FALSE.

**Prototype:**

**virtual int NextPixel()=0;**

**Remarks:**

Call this method to prepare for reading the next pixel.

**Return Value:**

Returns TRUE.

**Prototype:**

**virtual BOOL ReadChannelData(int chan, void \*data)=0;**

**Remarks:**

Reads a data element from the specified channel of the G-Buffer from the current scan line and pixel.

**Parameters:**

The channel to read. One of the items from: [List of GBuffer Channels Indexes](#).

**void \*data**

Points to storage for the data.

**Return Value:**

TRUE if data was available; otherwise FALSE.

**Prototype:**

**virtual BOOL ReadAllData(GBufData \*data)=0;**

**Remarks:**

Reads all the data from the G-Buffer into the **GBufData** structure passed from the current scan line and pixel.

**Parameters:**

**GBufData \*data**

Points to storage for the data. See [Structure GBufData](#).

**Return Value:**

TRUE if data was available; otherwise FALSE.

**Prototype:**

**virtual BOOL ModifyChannelData(int chan, void \*data)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows values in the specified layer to be written. Note that it may seem strange, writing data from the reader, but developers asked for the capability of writing to the already created gbuffer, and it is much simpler to add this capability to the GBufReader than to GBufWriter, which is designed to construct gbuffers from scratch, not modify existing ones.

**Parameters:**

**int chan**

Specifies the channel to write to.

**void \*data**

Points to the data to write.

**Return Value:**

TRUE indicates success; FALSE indicates failure.

**Prototype:**

**virtual BOOL ModifyAllData(GBufData \*data)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows values in the current layer to be written. Note that it may seem strange, writing data from the reader, but developers asked for the capability of writing to the already created gbuffer, and it is much simpler to add this capability to the GBufReader than to GBufWriter, which is designed to construct gbuffers from scratch, not modify existing ones.

**Parameters:**

**GBufData \*data**

Points to the data to write.

**Return Value:**

TRUE indicates success; FALSE indicates failure.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Deletes this reader object. Call this method when finished.

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);**

**Remarks:**

This method is available in release 3.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

**Parameters:**

**int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value (defined uniquely for each **cmd**).

**Default Implementation:**

**{ return 0; }**

## List of Material Requirement Flags

See Also: [Class MtlBase](#), [Class RenderInstance](#).

Material requirements flags. One or more of the following values:

### **MTLREQ\_2SIDE**

The material is 2-sided.

### **MTLREQ\_WIRE**

The material is a wire frame material.

### **MTLREQ\_WIRE\_ABS**

The material is a wire frame material, absolute size.

### **MTLREQ\_TRANSP**

The material uses transparency

### **MTLREQ\_UV**

The material requires UVW coordinates

### **MTLREQ\_FACEMAP**

The material uses "face map" UV coordinates.

### **MTLREQ\_XYZ**

The material requires object XYZ coordinates.

### **MTLREQ\_OXYZ**

This is not used.

### **MTLREQ\_BUMPUV**

The material requires UV bump vectors.

### **MTLREQ\_BGCOL**

The material requires the background color.

### **MTLREQ\_PHONG**

The material requires an interpolated normal.

### **MTLREQ\_AUTOREFLECT**

The material needs to build an auto-reflect map.

### **MTLREQ\_AUTOMIRROR**

The material needs to build an auto-mirror map

### **MTLREQ\_NOATMOS**

This suppress the atmospheric shader. This is used by the Matte material for example. The matte material samples the background itself and handles the

fogging characteristics. Thus it does not need to have the atmospheric shader do this again.

### **MTLREQ\_ADDITIVE\_TRANSP**

Specifies that transparent composites are done additively. Normally, if this is not specified, the background color is attenuated. For instance, say you have a material that is 30% transparent (70% opaque). This means that you take 30% of the background color and 70% of the material color.

Setting this flag causes the computation to be done by multiplying 0.7 times the material color and then adding this to the background color (which is left un-attenuated).

### **MTLREQ\_VIEW\_DEP**

This flag is available in release 2.0 and later only.

Maps or materials which depend on the view should set this bit in their Requirements() method.

### **MTLREQ\_UV2**

This flag is available in release 2.0 and later only.

The material requires the second uv channel values (vertex colors).

### **MTLREQ\_BUMPUV2**

This flag is available in release 2.0 and later only.

The material requires the second uv channel bump vectors.

### **MTLREQ\_PREPRO**

This flag is available in release 2.0 and later only.

Setting this flag will cause **MtlBase::BuildMaps()** to be called on every frame.

### **MTLREQ\_DONTMERGE\_FRAGMENTS**

This is no longer used.

### **MTLREQ\_DISPLACEMAP**

This flag is available in release 2.0 and later only.

Material has a Displacement map channel.

### **MTLREQ\_SUPERSAMPLE**

This flag is available in release 2.0 and later only.

Material requires super sampling. This tells the scanline renderer that you want super sampling (the Standard material uses this). This takes multiple samples spread around in the pixel which provides an additional level of anti-

aliasing.

### **MTLREQ\_WORLDCOORDS**

This flag is available in release 3.0 and later only.

Material has world coordinates that are used in material / map evaluation. This flag is set by UVGen and XYZGen when world coordinates are involved. The renderer looks at this and if it is set, takes the validity interval of the Node's transform matrix into account in computing the validity of a displacement mapped mesh.

### **MTLREQ\_TRANSP\_IN\_VP**

This flag is available in release 4.0 and later only.

This flag should be returned true for any material that wants to be transparent in the viewport.

### **MTLREQ\_FACETED**

This flag is available in release 4.0 and later only.

This material should be rendered faceted in the viewports.

# Class FaceCreate

See Also: [Class MeshDelta](#) , [Class Face](#)

class FaceCreate

## Description:

This class is available in release 4.0 and later only.

This class represents the notion of a face create within a MeshDelta. The public data members provide the index of any original face the new face may be based on as well as the new face itself. The virtue of providing the original face information is that we may use it to track per-face data that isn't contained within the face itself.

Note that this class relates closely to the new custom per-face data channels. For details on the implementations see [Class IFaceDataChannel](#), [Class IFaceDataMgr](#).

## Data Members:

public:

### **DWORD original;**

The face in the input mesh this new face should copy properties from. If UNDEFINED, this face is assumed to be totally original and should not copy properties from any faces in the input mesh.

### **Face face;**

The new face.

## Methods:

public:

## Prototype:

**FaceCreate (DWORD f, const Face & fc) : original(f), face(fc);**

## Remarks:

Constructor.

Constructor. Initializes data members to the values passed.

## Default Implementation:

```
{ }
```

**Prototype:**

**FaceCreate (const Face & fc) : original(UNDEFINED), face(fc);**

**Remarks:**

Constructor.

Constructor. Initializes face to the value passed, and initializes original to UNDEFINED.

**Default Implementation:**

{ }

**Prototype:**

**FaceCreate (DWORD f) : original(f);**

**Remarks:**

Constructor.

Constructor. Initializes original to the value passed. Does not initialize the face data member.

**Default Implementation:**

{ }

**Prototype:**

**FaceCreate () : original(UNDEFINED);**

**Remarks:**

Constructor. Initializes original to UNDEFINED. Does not initialize the face data member.

**Default Implementation:**

{ }

**Prototype:**

**FaceCreate (const FaceCreate & fc) : original(fc.original),  
face(fc.face);**

**Remarks:**

Constructor

Constructor. Initializes data members to match those in fc.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
FaceCreate & operator= (const FaceCreate & fc);
```

**Remarks:**

Assignment operator. Sets all data members to be equal to those in fc.

**Default Implementation:**

```
{ original = fc.original; face=fc.face; return *this; }
```

## Class MNMapFace

See Also: [Class MNMesh](#), [Class MNMap](#).

```
class MNMapFace
```

### Description:

This class is available in release 3.0 and later only.

This class is a face used to store map vertex information for a given face and map channel. It replaces the information previously contained in the MNFace::tvrt and MNFace::cvrt data members (prior to release 3).

By way of analogy: MNMapFace is to MNFace as TVFace is to Face. MNMapFace is to MNMap as MNFace is to MNMesh as TVFace is to MeshMap as Face is to Mesh.

### Friend Classes:

```
friend class MNMesh;
```

### Data Members:

```
public:
```

```
    int deg
```

Degree of this face. Must match degree of related MNFace in MNMesh.

```
    int *tv
```

Mapping vertices used by this mapping face. (This array is of size **deg**).

### Methods:

```
public:
```

### Prototype:

```
    MNMapFace();
```

### Remarks:

Constructor. Calls **Init()**.

### Prototype:

```
    MNMapFace(int d);
```

### Remarks:

Constructor. Sets the degree and hidden vertex count.

**Parameters:**

**int d**

Desired degree.

**Prototype:**

**~MNMapFace();**

**Remarks:**

Destructor. Calls **Clear()**.

**Prototype:**

**void Init();**

**Remarks:**

Initializes MNMapFace. **deg** is set to 0, pointers set to NULL.

**Prototype:**

**void Clear();**

**Remarks:**

Clears and frees MNMapFace.

**Prototype:**

**void SetAlloc(int d);**

**Remarks:**

Allocates enough memory in the arrays for the face to have degree **d**, but does not actually set the degree. If the arrays are already large enough (or larger), it does not reallocate them.

**Parameters:**

**int d**

The degree for this map face.

**Prototype:**

**void SetSize(int d);**

**Remarks:**

Allocates enough memory in the arrays for the face to have degree **d**, but does not actually set the degree. If the arrays are already large enough (or larger), it does not reallocate them. You generally don't need to use this method separately; MakePoly, Insert, and other methods which may require additional memory will call this if needed.

**Prototype:**

```
void MakePoly(int fdeg, int *tt);
```

**Remarks:**

Makes this face into a polygon with the specified vertices and other information.

**Parameters:**

**int fdeg**

The degree to set this face to.

**int \*tt**

The list of vertices for this face.

**Prototype:**

```
void Insert(int pos, int num=1);
```

**Remarks:**

Inserts space for more vertices at the specified position.

**Parameters:**

**int pos**

The location within the map face where the new vertices should be added.

**int num=1**

The number of new vertices to add.

**Prototype:**

```
void Delete(int pos, int num=1);
```

**Remarks:**

Deletes vertices from this map face.

**Parameters:**

The location within the map face where the vertices should be deleted.

**int num=1**

The number of vertices to delete.

**Prototype:**

**void RotateStart(int newstart);**

**Remarks:**

Re-indexes the vertices and edges so that the vertex in position **newstart** becomes the new first vertex. Triangulation is also corrected.

**Parameters:**

**int newstart**

The new first vertex.

**Prototype:**

**void Flip();**

**Remarks:**

Reverses order of verts, effectively inverting the face.

**Prototype:**

**int VertIndex(int vv);**

**Remarks:**

Returns the position of vertex **vv** in this face's list of vertices.

**Parameters:**

**int vv**

The vertex whose index is returned.

**Prototype:**

**void ReplaceVert(int ov, int nv);**

**Remarks:**

Replaces vertex **ov** with vertex **nv** in the list of vertices.

**Parameters:**

**int ov**

The vertex to replace.

**int nv**

The vertex to replace it with.

**Prototype:**

**MNMapFace & operator=(const MNMapFace & from);**

**Remarks:**

Assignment operator.

**Prototype:**

**void MNDebugPrint(bool hinfo=TRUE);**

**Remarks:**

Uses DebugPrint to print out information to the Debug Results window in DevStudio. This includes the **tv** array and optionally the **htv** array.

**Parameters:**

**bool hinfo=TRUE**

If TRUE the **htv** array is output.

## Class MNFace

See Also: [Class FlagUser](#), [Class MNFace](#), [Class BitArray](#).

```
class MNFace : public FlagUser
```

### Description:

This class is available in release 2.0 and later only.

**MNFace** is the face structure used with the **MNMesh** mesh. **MNFaces** need not be triangles. They also may contain "hidden" vertices which are used in converting the face back to triangles. This "triangulation" is always maintained in the face.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class MNMesh;
```

### Data Members:

private:

#### **int dalloc**

This is the amount of space allocated for verts, edges, mapping coords, and other information that's based on degree (deg).

#### **int talloc**

This is the amount of space allocated for the triangulation.

public:

#### **int deg**

Degree: this is the number of vertices and edges that this face has.

#### **int \*vtx**

This is the list of vertices that make up the corners of this face. Each value is an index into the parent MNMesh's list of MNVerts.

#### **int \*edg**

This is the list of edges that border this face, in order. Each edg[i] goes between vtx[i] and vtx[(i+1)%deg]. Each value is an index into the parent MNMesh's list of MNEdges. These values may not be valid if the MNMesh's MN\_MESH\_FILLED\_IN flag is not set.

#### **int \*diag**

This is where the triangulation is stored. The number of triangles in a face is

given by  $\text{deg} - 2 + \text{hdeg} * 2$ . This array contains three times this number, for all the corners of all the sub-triangles. The triangle values are indices into the vtx and hvtx arrays of this face. Hidden vertices are indicated by values less than zero: hvtx[i] is represented by -1-i. Thus a triangle (1, 2, -2) would represent a triangle using vtx[1], vtx[2], and hvtx[-1]. The diag array's allocated size is always  $(\text{dalloc}-3)*2$ . If  $\text{dalloc}==3$  (triangle), this pointer is NULL.

### **DWORD smGroup**

This contains the smoothing groups assigned to this face.

### **MtlID material**

This is the material ID assigned to this face.

### **int track**

This member is obsolete and should not be used.

### **BitArray visedg**

Contains a visibility bit for each edge on this face. See the MNMesh note on edge selection & visibility for more information.

### **BitArray edgsel**

Contains a selection bit for each edge on this face. See the MNMesh note on edge selection & visibility for more information.

## **Flags:**

### **MN\_SEL**

Indicates that the face is selected.

### **MN\_TARG**

Indicates that the face is targeted. (See the MNMesh methods starting with the words TargetBy.)

### **MN\_DEAD**

Indicates that the face is not used and should be ignored. Faces with the MN\_DEAD flag are deleted in the next MNMesh call to CollapseDeadFaces ().

### **MN\_FACE\_OPEN\_REGION**

This flag is available in release 2.5 and later only.

This face is part of a region of the mesh that is not closed, i.e. there are 1-sided edges. This means that the mesh is not a solid object, it has gaps or holes. The flag is set by the

### **MN\_FACE\_CHECKED**

This flag is available in release 2.5 and later only.

Reserved for internal use (in recursive face-traversing algorithms).

### **MN\_FACE\_CHANGED**

This flag can be cleared on all faces before an operation that moves some of the vertices of an MNMesh, then set for each face touching one of the moved vertices. This tells the parent MNMesh that these faces may need to have information such as triangulation recomputed. This flag is set by the MNMesh method SabinDoo in particular.

## **MN\_FACE\_CULLED**

Indicates that the face is culled (used during hit-testing). **Release 4.2 and above**

## **MN\_FACE\_WHATEVER**

Developers should not use this flag and should restrict themselves to **MN\_USER** and higher bits.

## **MN\_USER(1<<16)**

Flag bits at or above **MN\_USER** are reserved in all MNMesh components for the plug-in developer, if needed. Since FlagUser-derived classes have 32 flag bits, this allows for up to 16 user-defined flags.

### **Methods:**

#### **Prototype:**

**MNFace();**

#### **Remarks:**

Constructor. Initializes the face's arrays to NULL.

#### **Prototype:**

**MNFace(int d);**

#### **Remarks:**

Constructor. Initializes the face's degree to d and allocates space for all the arrays.

#### **Prototype:**

**MNFace(const MNFace \*from);**

#### **Remarks:**

Constructor. Copies flags, smoothing groups, and material from "from", but initializes the face's arrays to NULL and degree to 0.

#### **Prototype:**

**~MNFace();**

#### **Remarks:**

Frees all arrays.

**Prototype:**

**void Init();**

**Remarks:**

This method is available in release 4.0 and later only.  
Initialize the face.

**Prototype:**

**void SetDeg (int d);**

**Remarks:**

This method is available in release 4.0 and later only.  
Set the number of edges and vertices this face has.

**Parameters:**

**int d**

Number of vertices and edges.

**Prototype:**

**void Clear();**

**Remarks:**

Frees all arrays, setting them to NULL, and resets degree.

**Prototype:**

**int TriNum();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the number of triangles in this face.

**Prototype:**

**int FindTriPoint (int edge);**

**Remarks:**

This method is available in release 4.0 and later only.

Given the index of a particular edge, this routine returns the point (distinct from edge and  $(edge+1)\%deg$ ) that forms a triangle with the edge, given the current scheme of diagonals.

**Parameters:**

**int edge**

An index into the vertex array (in the range 0 to  $deg-1$ ) that indicates the starting vertex of the edge. (In other words, the edge falls between vertex  $vtx[edge]$  and  $vtx[(edge+1)\%deg]$ .)

**Return Value:**

The index of the desired vertex, again in the  $(0,deg-1)$  range, or edge if there's an error.

**Prototype:**

**int FindTriPoint (int a, int b);**

**Remarks:**

This method is available in release 4.0 and later only.

Given two verts that form a diagonal in the polygon, this method finds the vertex between them that connects by a diagonal or an outer edge to both of them. (Here, "between them" means after a and before b in sequence around the outside of the polygon. If we have an octagon where  $a=6$  and  $b=2$ , the result would be 7, 0, or 1. To get the other result, in the 3,4,5 range, call the method with  $a=2$  and  $b=6$ .)

**Parameters:**

**int a, b**

Two vertices, "internally indexed" in the 0 to  $deg-1$  range. This method is only guaranteed to work if the vertices share a diagonal. (Otherwise, there may be no solution.)

**Return Value:**

The index of the desired vertex, again in the  $(0,deg-1)$  range, or a if there's an error.

**Prototype:**

**void GetTriangles (Tab<int> &tri);**

**Remarks:**

This method is available in release 4.0 and later only.

This method fills in the table with the full triangulation for the face, based on the internal diagonal list. The table is set to size  $(deg-2)*3$ .

**Parameters:**

**Tab<int> &tri**

The table of triangles.

**Prototype:**

**void DiagSort (int dnum, int \*diag);**

**Remarks:**

This method is available in release 4.0 and later only.

**Note that this function is not part of the class but is available for us.**

This sorts the diagonals in the following fashion: each diagonal is reordered so that its smaller index comes first, then its larger. Then the list of diagonals is sorted so that it increases by second index, then decreases by first index. Such an ordered list for a 9-gon might be (1,3),(0,3), (0,4),(5,7),(4,7),(4,8). (This order is especially convenient for converting into triangles - it makes for a linear-time conversion.) **DiagSort()** uses qsort for speed.

**Parameters:**

**int dnum**

The size of the diag array - essentially double the number of diagonals.

**int \*diag**

The diagonals.

**Prototype:**

**void SetAlloc(int d, int h=0);**

**Remarks:**

Allocates enough memory in the arrays for the face to have degree d, but does not actually set the degree. If the arrays are already large enough (or larger), it does not reallocate them. You generally don't need to use this method separately; MakePoly, Insert, and other methods which may require additional

memory will call this if needed.

**Prototype:**

```
void MakePoly(int fdeg, int *vv, bool *vis=NULL, bool *sel=NULL);
```

**Remarks:**

This method is available in release 3.0 and later only.

Makes this face into a polygon with the specified vertices and other information. This routine also supplies a default triangulation for the face; however, since this MNFace-level routine cannot access the vertex positions contained in the parent MNMesh, this triangulation may not work for non-convex faces. If the face may not be convex, a call to MNMesh::RetriangulateFace for this face will correct the triangulation.

**Parameters:**

**int fdeg**

The degree to set this face to.

**int \*vv**

The list of vertices for this face. There must be at least fdeg of these. These values should be indices into the parent MNMesh's array of MNVerts.

**bool \*vis=NULL**

The edge visibility flags for the edges of this face. If this is NULL, it is ignored; otherwise, there must be at least fdeg of these. vis[i] represents the visibility of the edge going from vv[i] to vv[(i+1)%fdeg]. See the MNMesh note on edge selection & visibility for more information.

**bool \*sel=NULL**

The edge selection flags for the edges of this face. If this is NULL, it is ignored; otherwise, there must be at least fdeg of these. sel[i] represents the selection bit of the edge going from vv[i] to vv[(i+1)%fdeg]. See the MNMesh note on edge selection & visibility for more information.

**Prototype:**

```
void Insert(int pos, int num=1);
```

**Remarks:**

Inserts space for more vertices and edges on this face. This is used, for

example, when two faces are joined, to add room for the vertices & edges of one face to the other. This routine also renumbers the existing vertices and corrects the existing face triangulation, although it cannot provide the triangulation for the new vertices. It reserves space for the new triangles at the end of the triangle array. If you do not want to compute the triangulation for the new vertices yourself, you may use the MNMesh RetriangulateFace method after filling in the new vertices.

**Parameters:**

**int pos**

The location within the face where the new vertices and edges should be added.

**int num**

The number of new vertices and edges.

**Prototype:**

**void Delete(int pos, int num=1, int edir=1, bool fixtri=TRUE);**

**Remarks:**

Deletes vertices & edges from this face. This routine also corrects the face triangulation, removing those triangles that include the deleted edges and re-indexing the rest. However, delete may cause the triangulation to become invalid, by causing one or more of the corrected triangles to have a flipped normal.

**Parameters:**

**int pos**

The position of the first vertex to be deleted.

**int num=1**

The number of vertices & edges to delete.

**int edir=1**

There are two choices for the edges to be deleted: we can delete the edges going from pos to pos+1, pos+1 to pos+2, ... pos+num-1 to pos+num, or we can delete pos-1 to pos, pos to pos+1, ... pos+num-2 to pos+num-1.

(pos+num-1 is the last vertex deleted.) That is to say, we can delete the edges "before" the vertices we're deleting, or we can delete the edges "after" them. If edir is positive, we delete the edges after the vertices. If it's negative, we

delete the edges before. Keep in mind that this also affects edge visibility and selection information on this face.

**bool fixtri=TRUE**

This parameter is available in release 3.0 and later only.

This argument indicates how far Delete should go in fixing the triangulation. Delete will always correct the values of the tri array to correspond to the reduced-degree face. If fixtri is true, it will also delete those triangles that have collapsed because they had two vertices in the deleted region. If not, it will leave these triangles with overlapping vertices, as in (0,0,2).

**Return Value:**

Delete returns TRUE if fixtri is FALSE. If fixtri is TRUE, Delete will return TRUE if it successfully corrected the triangulation, or FALSE if there was a problem. If FALSE is returned, the triangulation will need to be revised with a call to RetriangulateFace.

**Prototype:**

**RotateStart(int newstart);**

**Remarks:**

This method is available in release 2.5 and later only.

Re-indexes the vertices and edges so that the vertex in position **newstart** becomes the new first vertex. Triangulation is also corrected. Mapping coordinates and vertex colors are corrected automatically.

**Prototype:**

**void Flip();**

**Remarks:**

This method is available in release 2.5 and later only.

Reverses order of verts, effectively inverting the face. vtx[0] remains unchanged, but vertex deg-1 becomes vertex 1, etc. Note that this operation wreaks havoc on nearby edges and should be used with caution.

**Prototype:**

**int VertIndex(int vv, int ee=-1);**

**Remarks:**

Returns the position of vertex **vv** in this face's list of vertices. For a given face

fc, if `fc.vtx[i] = vv`, `fc.VertIndex (vv) = i`.

Sometimes a single vertex from the MNMesh's MNVert list can be referenced more than once by a single face. The picture below illustrates this problem. The small triangle is actually outside of the face, and the vertex at the top of it is referenced twice by the face. Thus an additional edge parameter can be accepted. If `ee` is -1, it is ignored, and the first instance of `vv` is used. If `ee > -1`, this method looks for the instance of `vv` that starts out edge `ee`. Thus if `fc.vtx[i] = vv` and `fc.vtx[j] = vv`, but `fc.edg[i] != ee` and `fc.edg[j] = ee`, `j` is returned.

IMPORTANT: If no vertex is found matching the given parameters, this method generates an assertion failure. Please be sure that vertex `vv` is actually on the face (and that edge `ee` follows it if `ee` is not -1) before using this method.

**Prototype:**

**`int EdgeIndex(int ee, int vv=-1);`**

**Remarks:**

Returns the position of edge `ee` in this face's list of edges. For a given face `fc`, if `fc.edg[i] = ee`, `fc.EdgeIndex (ee) = i`.

Sometimes a single edge from the MNMesh's MNEdge list can be referenced more than once by a single face. The small rectangle is actually outside of the face, and the edge above it is referenced twice by the face, once in each direction. Thus an additional vertex parameter can be accepted. If `vv` is -1, it is ignored, and the first instance of `ee` is used. If `vv > -1`, this method looks for the instance of `ee` that starts out with vertex `vv`. Thus if `fc.edg[i] = ee` and `fc.edg[j] = ee`, but `fc.vtx[i] != vv` and `fc.vtx[j] = vv`, `j` is returned.

IMPORTANT: If no edge is found matching the given parameters, this method generates an assertion failure. Please be sure that edge `ee` is actually on the face (and that vertex `vv` follows it if `vv` is not -1) before using this method.

**Prototype:**

**`ReplaceVert(int ov, int nv, int ee=-1);`**

**Remarks:**

This method is available in release 2.5 and later only.

Replaces vertex **ov** with vertex **nv** in the list of vertices. It is possible for a

face to reference the same vertex more than once, however the combination of a vertex followed by a specified edge is unique. Therefore if **ee<0**, all instances of **ov** are replaced by **nv**, but if not, only the instance of **ov** followed by **ee** is replaced.

**Prototype:**

**ReplaceEdge(int oe, int ne, int vv=-1);**

**Remarks:**

Replaces edge **oe** with edge **ne** in the list of edges. It is possible for a face to reference the same edge twice, however the combination of an edge preceded by a specified vertex is unique. Therefore if **vv<0**, all instances of **oe** are replaced by **ne**, but if not, only the instance of **oe** preceded by **vv** is replaced.

**Prototype:**

**void MNDebugPrint(bool triprint=FALSE, bool hinfo=TRUE);**

**Remarks:**

This method is available in release 2.5 and later only.

Uses DebugPrint to print out face information to the Debug Results window in DevStudio. The information consists of the vertices and edges used by this face. It is generally a good idea to put in a DebugPrint immediately before this with the index of the edge, so you know which one is being printed out:

**DebugPrint("Face %d: ", fid);**

**F(fid)->MNDebugPrint();**

**Parameters:**

**bool triprint=FALSE**

Print out triangulation information.

**bool hinfo=TRUE**

Print out hidden vertex information.

**Prototype:**

**MNFace & operator=(const MNFace & from);**

**Remarks:**

Assignment operator. Copies all information from "from", including triangulation, hidden vertices, flags, smoothing & material info, and "track".

**Prototype:**

```
int &operator[](int i);
```

**Remarks:**

This method is available in release 4.0 and later only.

Access operator.

**Default Implementation:**

```
{ return vtx[i]; }
```

**Prototype:**

```
const int &operator[](int i) const;
```

**Remarks:**

This method is available in release 4.0 and later only.

Access operator.

**Default Implementation:**

```
{ return vtx[i]; }
```

## Class MNMap

See Also: [Class FlagUser](#), [Class MNMapFace](#), [Class Point3](#), [Class Matrix3](#).

```
class MNMap : public FlagUser
```

### Description:

This class is available in release 3.0 and later only. It replaces the old `tv` and `cv` arrays in `MNMesh`, and the old `tvrt` and `cvrt` arrays in `MNFace`.

Each `MNMap` holds the mapping information for a particular map channel of the `MNMesh`. As with regular Mesh maps, the `MNMap` holds an array of map vertices and an array of map faces that define how those mapping vertices are applied to the `MNMesh`. The number (`numf`) of mapping faces should always match the number of faces in the parent `MNMesh`, and each `MNMapFace` gives the map vertices for the related `MNFace`. The number of map vertices may be different from the number of `MNVerts` in the parent `MNMesh`.

### Friend Classes:

```
friend class MNMesh;
```

### Data Members:

```
public:
```

```
    MNMapFace *f;
```

The array of map faces.

```
    UVVert *v;
```

The array of UVVerts. Note: **typedef Point3 UVVert;**

```
    int numv
```

The number of vertices in the `v` array.

```
    int numf;
```

The number of map faces in the `f` array.

### Methods:

```
public:
```

### Prototype:

```
    MNMap();
```

### Remarks:

Constructor. This calls **Init()**.

**Prototype:**

**~MNMap();**

**Remarks:**

Destructor. This calls **ClearAndFree()**.

**Prototype:**

**void Init();**

**Remarks:**

Initializes the map – sets numv and numf to 0, and sets the pointers to NULL. Also initializes private allocation related data members.

**Prototype:**

**void VAlloc(int num, bool keep=TRUE);**

**Remarks:**

Allocates the specified number of UVVerts in the **v** array. (If you're creating a number of map verts, but you're not sure exactly how many, it's good to pre-allocate a large number using this method.) This method doesn't affect MNMap::numv, only the allocation amount.

**Parameters:**

**int num**

The number of UVVerts to allocate.

**bool keep=TRUE**

If TRUE any previous verts are kept; otherwise they are discarded.

**Prototype:**

**void FAlloc(int num, bool keep=TRUE);**

**Remarks:**

Allocates the specified number of map faces in the **f** array. This method doesn't affect MNMap::numf, just the number allocated.

**Parameters:**

**int num**

The number of map verts to allocate.

**bool keep=TRUE**

If TRUE any previous map faces are kept; otherwise they are discarded.

**Prototype:**

**int VNum() const;**

**Remarks:**

Returns the current number of UVVerts.

**Prototype:**

**UVVert V(int i) const;**

**Remarks:**

Returns the 'i-th' UVVert.

**Parameters:**

**int i**

The zero based index into the **v** array of the UVVert to return.

**Prototype:**

**int FNum() const;**

**Remarks:**

Returns the number of map faces.

**Prototype:**

**MNMapFace \*F(int i) const;**

**Remarks:**

Returns a pointer to the 'i-th' map face.

**Parameters:**

**int i**

The zero based index of the map face to return.

**Prototype:**

**int NewTri(int a, int b, int c);**

**Remarks:**

Like MNMesh::NewTri, this creates a new mapping face, of degree 3, with the specified map vertices.

**Parameters:**

**int a**

**int b**

**int c**

The mapping vertices for the new triangle.

**Prototype:**

**int NewTri(int \*vv);**

**Remarks:**

Like MNMesh::NewTri, this creates a new mapping face, of degree 3, with the specified map vertices.

**Parameters:**

**int \*vv**

The mapping vertices for the new triangle.

**Prototype:**

**int NewQuad(int a, int b, int c, int d);**

**Remarks:**

Like MNMesh::NewQuad, this creates a quad, with the specified map vertices.

**Parameters:**

**int a**

**int b**

**int c**

**int d**

The mapping vertices for this quad. (The two faces are formed with mapping verts (a,b,c) and (c,d,a).

**Prototype:**

**int NewQuad(int \*vv);**

**Remarks:**

Like MNMesh::NewQuad, this creates a quad, with the specified map vertices.

**Parameters:**

**int \*vv**

The mapping vertices for this quad. (The two faces are formed with mapping

verts (vv[0],vv[1],vv[2]) and (vv[2],vv[3],vv[0]).

**Prototype:**

**int NewFace(int degg=0, int \*vv=NULL);**

**Remarks:**

Like MNMesh::NewFace, this creates a new mapping face with the specified degree and map vertices.

**Parameters:**

**int degg=0**

The degree of the new face. (Note that this should match the degree of the associated MNFace.)

**int \*vv=NULL**

The new mapping vertices, in order around the perimeter.

**Prototype:**

**void setNumFaces(int nfnun);**

**Remarks:**

Sets the number of map faces keeping any previously allocated faces.

**Parameters:**

**int nfnun**

The number of faces to set.

**Prototype:**

**int NewVert(UVVert p, int uoff=0, int voff=0);**

**Remarks:**

Allocates storage for a new specified UVVert. An optional offset to the UVVert may be supplied. (The offset is useful when you want to create a new UVVert which is on the other side of a seam in U or V from some existing UVVert p.)

**Parameters:**

**UVVert p**

The UVVert to store.

**int uoff=0**

If non-zero this is an offset applied in U when p is stored.

**int voff=0**

If non-zero this is an offset applied in V when p is stored.

**Prototype:**

**void setNumVerts(int nvnum);**

**Remarks:**

Sets the number of UVVerts keeping any previously allocated UVVerts.

**Parameters:**

**int nvnum**

The number of UVVerts to set.

**Prototype:**

**void CollapseDeadVerts(MNFace \*faces);**

**Remarks:**

Eliminates the mapping vertices not used by any active mapping faces in this map. (Analogous to the Mesh::DeleteIsoMapVerts method.)

**Parameters:**

**MNFace \*faces**

This should point to the MNFace array of the parent MNMesh. This allows the method to find out if any faces should be considered MN\_DEAD, so that it won't mark such faces' map verts as used. (The MN\_DEAD flag is stored in the MNFace, but not in the related MNMapFaces.)

**Prototype:**

**void CollapseDeadFaces(MNFace \*faces);**

**Remarks:**

Eliminates the MN\_DEAD map faces from the array. This should be called immediately before calling CollapseDeadFaces on the parent MNMesh, to keep the face arrays in sync.

**Parameters:**

**MNFace \*faces**

The parent MNMesh's array of MNFaces. This array is used to find out which faces are MN\_DEAD.

**Prototype:**

**void Clear();**

**Remarks:**

Sets the number of UVVerts and map faces to 0.

**Prototype:**

**void ClearAndFree();**

**Remarks:**

Sets the number of UVVerts and map faces to 0 and deallocates the memory.

**Prototype:**

**void Transform(Matrix3 & xfm);**

**Remarks:**

Transforms each UVVert with the specified matrix.

**Parameters:**

**Matrix3 &xfm**

The matrix to transform the UVVerts by.

**Prototype:**

**MNMap & operator=(const MNMap &from);**

**Remarks:**

Assignment operator.

**Prototype:**

**MNMap & operator+=(const MNMap &from);**

**Remarks:**

Appends the specified MNMap object onto this MNMap.

**Parameters:**

**const MNMap &from**

The source MNMap.

**Prototype:**

**MNMap & operator+=(const MNMesh &from);**

**Remarks:**

Appends a default planar map corresponding to the mesh passed onto this map. This is typically used when joining two MNMeshes together when one

MNMesh has a map channel active but the other doesn't. Rather than eliminate the map channel, we just use the object coordinates of the other mesh to generate a planar map for its faces.

**Parameters:**

**const MNMesh &from**

The source MNMesh.

**Prototype:**

**void ShallowCopy(const MNMap & from);**

**Remarks:**

Copies pointers. To avoid memory errors, this method should only be used by the pipeline.

**Parameters:**

**const MNMap & from**

The map to copy pointers from.

**Prototype:**

**void NewAndCopy();**

**Remarks:**

Creates new pointers and copies over all the data. To avoid memory errors, this method should only be used by the pipeline.

**Prototype:**

**void MNDebugPrint(MNFace \*faces);**

**Remarks:**

Uses DebugPrint to print out information about this MNMap to the Debug Results window in DevStudio. This includes all map verts and faces, one per line.

**Parameters:**

**MNFace \*faces**

This should be the MNFace array of the parent MNMesh. It's used to prevent the DebugPrinting of MN\_DEAD faces.

**Prototype:**

**bool CheckAllData(int mp, int nf, MNFace \*faces);**

**Remarks:**

Checks the MNMap for internal errors, such as a MNMapFace referring to an out of range map vertex. If an error is found, an error message is DebugPrinted, and the method returns FALSE. This is strictly a debugging tool of no use in releases – a good way to use it is to say DbgAssert (CheckAllData()), so it won't be called when not in a Debug build, and so it'll throw up an assertion failure if something's wrong.

Error messages generated:

Map %d: Wrong number of faces. Should be %d, is %d. (Sent if nf != MNMap::numf)

Map %d, Face %d has wrong degree. Should be %d, is %d. (Sent if f[i].deg != faces[i].deg)

Map %d, Face %d has wrong hdegree. Should be %d, is %d. (Sent if f[i].hdeg != faces[i].hdeg)

Map %d, Face %d has an out-of-range map vertex: %d.

Map %d, Face %d has an out-of-range hidden map vertex: %d

**Parameters:****int mp**

The map channel this MNMap represents – used to give more helpful debug information.

**int nf**

The number of faces of the parent MNMesh.

**MNFace \*faces**

The parent MNMesh's face array. This is used to prevent checking of MN\_DEAD faces, which may safely be invalid.

**Prototype:**

**IOResult Save(ISave \*isave, MNFace \*faces=NULL);**

**Remarks:**

Used internally to save this MNMap to the 3ds max file.

**Prototype:**

**IOResult Load(ILoad \*iload, MNFace \*faces=NULL);**

**Remarks:**

Used internally to load this MNMap from the 3ds max file.

## Class RadiosityEffect

See Also: : [Class ObjLightDesc](#), [Class RadiosityInterface](#), [Class IRadiosityEffectExtension](#), [Class NodeDisplayCallback](#), [Class IRadiosityPreferences](#)

```
class RadiosityEffect : public SpecialFX
```

### Description:

This class is only available in release 5 or later.

The class provides the interfaces for creating Advanced Lighting plug-ins in 3ds max. RadiosityEffect plug-ins are controlled through the Advanced Lighting dialog from the render menu.

A RadiosityEffect behaves as a light in the scene. An implementation of RadiosityEffect must also provide an implementation of ObjLightDesc, to be returned from RadiosityEffect::CreateLightDesc(). This ObjLightDesc is then used by the renderer to apply the radiosity light during a render.

The architecture is designed to support scene based lighting solvers (which use a precalculated solution), as well as image based solvers (which calculate their solution at render time). The method RunProcess() and its relatives, StopProcess(), AbortProcess(), and WaitForCompletion() are called by the system to request processing of a scene based solution. Image based solvers only need stub implementations of these methods.

See also [Class NodeDisplayCallback](#). A RadiosityEffect can use a NodeDisplayCallback to control the viewport display of scene geometry. In particular, it is possible for a lighting solution to be displayed interactively in the viewport if the NodeDisplayCallback provides geometry with appropriate vertex illumination.

A RadiosityEffect may want to reference objects in the scene to detect events which invalidate its lighting solution. A problem arises that the RadiosityEffect can become inundated with reference messages, so to address this problem the RadiosityEffect should ignore any message where the PART\_EXCLUDE\_RADIOSITY flag is set in the PartID of the message. Any message where ((partID|PART\_EXCLUDE\_RADIOSITY)!=0) is an event that the system deems should not invalidate the radiosity, for example, when texture display in viewport is turned on or off.

See also `IsInterestedInChannels()` in [Class IRadiosityEffectExtension](#).

All methods of this class are implemented by the plug-in.

**Methods:**

**Prototype:**

**virtual void SetActive(bool active, TimeValue t)**

**Remarks:**

Enables or disables the radiosity effect. Corresponds to the Active checkbox in the Advanced Lighting dialog. Although a TimeValue parameter is passed to this method, the active state of the plug-in is not animatable. Plug-ins should call the default implementation of `SetActive()`, optionally adding their own code.

**Parameters:**

**bool active**

Specifies whether the plug-in is active or inactive

**TimeValue t**

Specifies the current scene time when the method is called.

**Prototype:**

**virtual IOResult Merge(ILoad\* iload, )**

**Remarks:**

Reserved for future versions. Not currently used

**Default Implementation:**

```
{ return IO_OK; }
```

**Prototype:**

**virtual RadiosityParamDlg \*CreateParamDlg(IRendParams \*ip)**

**Remarks:**

This method creates and returns a new instance of RadiosityParamDlg to manager the user interface. This displays one or more rollouts in the Advanced Lighting dialog. A typical implementation would call ClassDesc2::CreateParamDialogs() to instantiate the RadiosityParamDlg.

**Parameters:**

**IRendParams \*ip**

This is the interface given to the plug-in so it may display its parameters.

**Return Value:**

The instance of RadiosityParamDlg to manager the plug-in's UI

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

**virtual BOOL SetDlgThing(RadiosityParamDlg \*ip)**

**Remarks:**

Implement this if you are using the ParamMap2 AUTO\_UI system and the RadiosityEffect has secondary dialogs that don't have the effect as their 'thing'. Called once for each secondary dialog for you to install the correct thing.

Note: Developers needing more information on this method can see the remarks for [MtlBase::CreateParamDlg\(\)](#) which describes a similar example of this method in use (in that case it's for use in a texture map plug-in).

**Parameters:**

## **RadiosityParamDlg \*ip**

Points to the ParamDlg.

### **Return Value:**

Return TRUE if you process the dialog; otherwise FALSE

### **Default Implementation:**

```
{ return FALSE; }
```

### **Prototype:**

```
virtual bool UseLight(INode* node)
```

### **Remarks:**

Returns whether the given light should render it's illumination in the production render. If the return value is true, the light is disabled while rendering.

This is used to allow the radiosity plug-in to override lights in the scene with light from it's own solution.

### **Parameters:**

**INode\* node**

The INode of the light.

### **Return Value:**

Return true if the light should be enabled while rendering; or false if it should be disabled

### **Default Implementation:**

```
{ return true; }
```

**Prototype:****virtual int NumLightDesc() const = 0****Return Value:**

The number of ObjLightDesc objects the RadiosityEffect will return from CreateLightDesc()

**Prototype:****virtual void CreateLightDesc(ObjLightDesc\*\* buffer) = 0****Remarks:**

Creates light objects that the renderer can use to get the RadiosityEffect's contribution. CreateLightDesc() creates a number of ObjLightDesc objects indicated by NumLightDesc(), and stores their addresses in the buffer. Caller is responsible for ensuring that the buffer is large enough.

Note: the caller will delete the ObjLightDesc objects when the render is completed, so this method should dynamically allocate the ObjLightDesc instances, rather than providing static instances.

**Parameters:****ObjLightDesc\*\* buffer**

The buffer into which the ObjLightDesc pointers should be stored.

**Prototype:****virtual void RunProcess(TimeValue t, RenderGlobalContext\* rgc, bool interactiveRender) = 0****Remarks:**

Called by the system to start the radiosity processing. This should start the process from the beginning, or where it stopped previously, if applicable. The method should launch a separate thread and return immediately; the system will call WaitForCompletion() to wait for the thread to complete.

This is specific to solvers which use a pre-calculated solution. Other solvers need only a stub implementation.

**Parameters:**

**TimeValue t**

Specifies the scene time when the method is called.

**RenderGlobalContext\* rgc**

This can be used to retrieve information about the global rendering environment.

**bool interactiveRender**

Specifies whether the lighting solution is being calculated for interactive rendering. Note that the default renderer does not support interactive rendering with Advanced Lighting, but other plug-in renderers might potentially do so, in which case the RadiosityEffect should attempt to generate a fast, lower quality solution for interactive display.

**Prototype:**

**virtual void StopProcess(bool allowAbort = true) = 0**

**Remarks:**

Stop the lighting calculation. If possible, the RadiosityEffect should attempt to reach an intermediate solution, so that calculation can be continued later. However, this may take awhile, and If allowAbort is true, the RadiosityEffect is expected to prompt the user with a dialog, asking if they wish to abort.

**Parameters:**

**bool allowAbort node**

If true, the RadiosityEffect is expected to prompt the user with a dialog so they may abort the process of stopping the calculation and saving an intermediate solution

**Prototype:****virtual void AbortProcess() = 0****Remarks:**

Abort the lighting calculation. The RadiosityEffect should to abort immediately, without saving an intermediate solution.

**Prototype:****virtual void WaitForCompletion(RenderContext\* rc = NULL, DWORD timeout = INFINITE) = 0****Remarks:**

This is called by the system when waiting for the lighting calculation to finish. This method should not return to the caller until the calculation is complete, or when the timeout (in milliseconds) expires. The RenderContext can be used to display the progress of the calculation to the user, and to check if the cancel button is pressed.

**Parameters:****RenderContext\* rc**

Use this to display a progress bar for the user, or to detect when the user hits the cancel button.

**DWORD timeout**

An amount of time (measured in milliseconds) that the system is willing to wait. The method should measure its own running time and return to the caller when the timeout expires.

**Prototype:****virtual bool NeedsCamVerts(TimeValue t, RenderGlobalContext\* rgc, bool interactiveRender, bool saveMem)**

**Remarks:**

Indicates whether the RadiosityEffect plug-in wants the renderer to build camera-space vertices.

**Parameters:****TimeValue\* t**

Specifies the current scene time when the method is called.

**RenderGlobalContext\* rgc**

This can be used to retrieve information about the global rendering environment.

**bool interactiveRender**

Specifies whether the lighting solution is being calculated for interactive rendering. Note that the default renderer does not support interactive rendering with Advanced Lighting, but other plug-in renderers might potentially do so, in which case the RadiosityEffect should attempt to generate a fast, lower quality solution for interactive display.

**bool saveMem**

True if the user has selected “Conserve Memory” in the Render Dialog, under the “MAX Default Scanline A-Buffer” rollout, or if the render is occurring in the material editor.

**Return Value:**

True if the RadiosityEffect wants the renderer to calculate camera-space vertices, false otherwise.

**Default Implementation:**

```
{ return false; }
```

## Class ISpecularCompositeShader

```
class ISpecularCompositeShader : public BaseInterface
```

### Description:

**This class is only available in release 5 or later.**

This class is only used to communicate some information between the MAX Standard Material and the Shaders for the MAX Standard Material. The information that the Shaders require that wasn't supplied previous is the RenderGlobalContext. This would be used by 3rd parties who want to write shaders for the MAX standard material. This is only required if the shader performs operations that are not valid for arbitrary light values. For example: Using the default values, the Physical Sun has light multiplier values that are around 60.

Several standard MAX shaders, like the Aniso and ONB shaders, composite specular highlights over the diffuse light, and the calculation looks something like:

**color = diffuse \* (1 - specular) + specular;**

This works fine as long as specular is between 0 and 1. If specular goes above 1, you get all kinds of unusual artifacts. This calculation is done this way to prevent color clipping because of bright specular highlights. With lighting values at 60, the specular is frequently greater than 1. With an exposure control, the exposure control manages the color clipping and we can simply add the diffuse and specular components.

This interface gives the shader a chance to investigate the render parameters and decide how the lighting is to be combined.

This is usually done by using multiple inheritances and deriving the Shader

Implementation from Shader and ISpecularCompositeShader.

The Shader needs to override GetInterface(Interface\_ID id) and return the address of the ISpecularCompositShader interface and GetRequirements also needs to be overridden to return MTLREQ\_PREPRO as one of the shader requirements.

Then the shader needs to implement ChooseSpecularMethod. The typical implementation is:

```
void CombineComponentsCompShader::ChooseSpecularMethod(TimeValue t,
RenderGlobalContext* rgc)
{
useComposite = true;
if (rgc == NULL) {
ToneOperatorInterface* tint = static_cast<ToneOperatorInterface*>
(
GetCOREInterface(TONE_OPERATOR_INTERFACE));
if (tint != NULL) {
ToneOperator* top = tint->GetToneOperator();
if (top != NULL && top->Active(t))
useComposite = false;
}
} else {
ToneOperator* top = rgc->pToneOp;
if (top != NULL && top->Active(t))
useComposite = false;
}
}
}
```

**Methods:**

**Prototype**

**virtual void ChooseSpecularMethod(TimeValue t,  
RenderGlobalContext\* rgc) = 0**

**Remarks:**

To be implemented Material Shaders. Provides a selection mechanism for the specular methods provided by various tone operators either provided by the System or by the implementing Shader.

**Parameters:**

## **TimeValue t**

Present scene time value.

**RenderGlobalContext \*rgc**

Pointer to a RenderGlobalContext that encapsulates the shared data between a Standard Material and the implementing Shader.

**Return Value:**

None

## Class IEmissionColor

```
class IEmissionColor : public BaseInterface
```

### Description:

**This class is only available in release 5 or later.**

Function-published class: Material emission color, for self-illuminates meshes in viewport

### Methods:

#### Prototype

```
virtual Interface_ID GetID() { return  
EMISSION_COLOR_INTERFACE_ID; }
```

#### Remarks:

Returns the Function Published interface ID for this interface.

It is **#define EMISSION\_COLOR\_INTERFACE\_ID  
Interface\_ID(0x4f803aa8, 0x71611798)**

#### Prototype

```
virtual LifetimeType LifetimeControl() { return noRelease; }
```

#### Remarks:

#### Prototype

```
virtual void SetColor(Point3 color)=0;
```

#### Remarks:

Sets the emission color as Point3.

### **K Prototype**

**virtual Point3 GetColor()=0;**

#### **Remarks:**

Returns the emission color as Point3.

### **K Prototype**

**virtual void SetAlpha(float alpha)=0;**

#### **Remarks:**

Sets the alpha color value for the emission in the range of 0.f to 1.f.

### **K Prototype**

**virtual float GetAlpha()=0;**

#### **Remarks:**

Gets the alpha color value for the emission in the range of 0.f to 1.f.

## Class INodeGIProperties

```
class INodeGIProperties : public FPMixinInterface
```

### Description:

**This class is only available in release 5 or later.**

This class defines an interface for accessing a node's global illumination properties.

An instance of this interface can be retrieved using the following line of code (assuming 'node' is of type INode\*):

```
static_cast<INodeGIProperties*>(node->GetInterface(NODEGIPROPERTIES_INTERFACE))
```

Geometric and object objects have different properties. Accessing/setting geometric properties of non-geometric objects is safe, but will have no effect in the global illumination solution.

Here is a description of the global illumination properties:

GENERAL PROPERTIES (all types of objects)

- **Excluded:** Excluded objects should be ignored by the radiosity engine. They should act as if these objects do not exist.
- **ByLayer:** Specifies whether the GI properties of this node's layer should be used instead of the local settings.

GEOMETRIC OBJECTS (affects only geometric objects):

- Occluder: Occluding objects will block rays of light that hit their surface. Non-occluding objects will not block those rays, but will still receive illumination if the Receiver property is set.
- Receiver: Receiver objects will receive and store illumination from rays of light that hit their surface. Non-receiver objects will not store illumination.
- Diffuse: Diffuse surfaces will reflect and transmit light based on their diffuse color and transparency value.
- Specular: Specular surfaces will generate specular reflections and transparency. ex.: glass is specular transparent and a mirror is specular reflective.
- UseGlobalMeshSettings: When subdividing the geometry for a more accurate GI solution, this flag specifies whether some 'global' settings, or the node's local settings should be used.
- MeshingEnabled: When using local settings, this specifies whether geometry subdivision should occur on this node.
- MeshSize: The maximum size, in MAX universe units, that a face should have after being subdivided.
- NbRefineSteps: This is the saved number of refining steps to be performed on this node by the global illumination engine.
- ExcludedFromRegather: Set to 'true' to excluded an object from the 'regathering' process.
- RayMult: Specifies a multiplier that will increase or decrease the number of rays cast for this object when regathering.

#### LIGHT OBJECTS (affects only light sources):

- StoreIllumToMesh: Specifies whether the light emitted from this object should be stored in the GI solution's mesh, and not be re-cast at render-time.

### **K Prototype**

**virtual void CopyGIPropertiesFrom(const INodeGIProperties& source) = 0;**

#### **Remarks:**

Clones the properties from a source

### **K Prototype**

**virtual BOOL GIGetIsExcluded() const = 0;**

#### **Remarks:**

Saves local properties - to be called on every node

### **K Prototype**

**virtual void GISetIsExcluded(BOOL isExcluded) = 0;**

#### **Remarks:**

Saves local properties - to be called on every node. See also GIGetIsExcluded()

### **K Prototype**

**virtual float GIGetMeshSize() const = 0;**

#### **Remarks:**

Geometry object property – retrieves the current mesh size for the node.

### **K Prototype**

**virtual void GISetIsOccluder(BOOL isOccluder) = 0;**

#### **Remarks:**

Geometry object property – query for whether the node is an occluder object in the global illumination computations.

### **K Prototype**

**virtual void GISetIsReceiver(BOOL isReceiver) = 0;**

**Remarks:**

Geometry object property – query for whether the node is a receiver for global illumination.

**K Prototype**

**virtual void GISetIsDiffuse(BOOL isDiffuseReflective) = 0;**

**Remarks:**

Geometry object property – query for whether the node has diffuse properties for usage in global illumination.

**K Prototype**

**virtual void GISetIsSpecular(BOOL isSpecular) = 0;**

**Remarks:**

Geometry object property – query for whether the node has specular properties for usage in global illumination.

**K Prototype**

**virtual void GISetUseGlobalMeshSettings(BOOL globalMeshing) = 0;**

**Remarks:**

Geometry object property – set usage of global rendering context mesh refinement settings..

**K Prototype**

**virtual void GISetMeshingEnabled(BOOL meshingEnabled) = 0;**

**Remarks:**

Geometry object property – set whether mesh refinement is enabled for this pass.

## **K Prototype**

**virtual void GISetNbRefineSteps(unsigned short nbRefineSteps) = 0;**

### **Remarks:**

Geometry object property – set the number of mesh refinement steps for the regathering pass.

## **K Prototype**

**virtual void GISetNbRefineStepsDone(unsigned short nbRefineStepsDone) = 0;**

### **Remarks:**

Geometry object property – query the number of steps performed thus far during the mesh refinement stage.

## **K Prototype**

**virtual void GISetMeshSize(float size) = 0;**

### **Remarks:**

Geometry object property – query for the refinement mesh size for the regathering pass.

## **K Prototype**

**virtual BOOL GIGetIsExcludedFromRegather() const = 0;**

### **Remarks:**

Geometry object property – query whether this node is to be excluded from the regathering pass.

## **K Prototype**

**virtual void GISetIsExcludedFromRegather(BOOL isExcluded) = 0;**

**Remarks:**

Geometry object property – set whether this node is to be excluded from the regathering pass.

**K Prototype**

**virtual BOOL GIGetStoreIllumToMesh() const = 0;**

**Remarks:**

Get light object property.

**K Prototype**

**virtual void GISetStoreIllumToMesh(BOOL storeIllum) = 0;**

**Remarks:**

Set light object property.

**K Prototype**

**virtual BOOL GIGetByLayer() const = 0;**

**Remarks:**

Get 'by layer' flag for radiosity properties.

**K Prototype**

**virtual void GISetByLayer(BOOL byLayer) = 0;**

**Remarks:**

Set the 'by layer' flag for radiosity properties.

**K Prototype**

**virtual float GIGetRayMult() const = 0;**

**Remarks:**

Getting regathering ray multiplier node property

**K Prototype**

**virtual void GISetRayMult(float rayMult) = 0;**

**Remarks:**

Setting regathering ray multiplier node property.

## Class ISplineIKControl

### **Description:**

**This class is only available in release 5 or later.**

This class refers to the new modifier which, when assigned to a spline (or a NURBS) curve, generates a certain number of helper objects attached to the knots of the curve.

Spline IK Control Modifier (works for NURBS Point curve and NURBS CV Curve too): This is a modifier and can be used independently of the SplineIK. When applied to a spline, this modifier creates one point helper for each knot of the spline. The user can then animate the spline by simply animating (position and rotation) the point helpers. Thus to animate the spline, the user wouldn't need to get into the sub-object level.

There are three options Link Types, presented as a set of 3 radio buttons:

### **Link All in Hierarchy (default):**

Makes a helper a child to its immediately previous helper. So, Helper#2 is child to Helper#1, Helper#3 is child to Helper#2, and so on. Helper#1 is still child to the world. Translation and rotation of a helper then "solidly" moves/rotates part of the spline *\_subsequent\_* to the selected helper. The part of the spline previous to the helper is unaffected.

### **Link All to Root**

Makes all helpers children to Helper#1, i.e., knot#1. Helper#1 can be position constrained or linked to another object, like it is possible above. Additionally individual helpers can be moved and rotated without any other helper being affect.

### **No Linking**

All helpers are independent -- not linked to any other helper -- so that they can be moved and rotated without any other helper being affect.

### **"Create Helpers" button:**

Helpers are not automatically added to the spline on the assignment of the modifier. To do that the user need to press the "Create Helpers" button.

Helper Display:

These are the all the options:

- center marker (default OFF)
- axis tripod (default OFF)
- cross (default OFF)
- box (default ON)
- size (default 20.0)
- constant screen size (default OFF)
- draw on top (default ON) for a standard point object.

If the user adds ("insert") a knot to the spline, a new helper object is automatically created at that knotpoint.

Furthermore, please refer to the implementation project which is in **/maxsdk/samples/modifiers/splineikcontrol**.

The following helper methods have been added to **istdplug.h** for general access:

### **Methods:**

#### **Prototype:**

**virtual int GetHelperCount()=0;**

#### **Remarks:**

Obtain the number of helpers created

#### **Prototype:**

**virtual int GetKnotCount()=0;**

**Remarks:**

Gets the total number of knots for spline.

**Prototype:**

**virtual BOOL LinkToRoot() = 0;**

**Remarks:**

Makes all helpers children to Helper#1, i.e., knot#1. Helper#1 can be position constrained or linked to another object, like it is possible above. Additionally individual helpers can be moved and rotated without any other helper being affected.

**Prototype:**

**virtual BOOL LinkInHierarchy() = 0;**

**Remarks:**

Makes a helper a child to its immediately previous helper. So, Helper#2 is child to Helper#1, Helper#3 is child to Helper#2, and so on. Helper#1 is still child to the world. Translation and rotation of a helper then "solidly" moves/rotates part of the spline *\_subsequent\_* to the selected helper. The part of the spline previous to the helper is unaffected.

**Prototype:**

**virtual BOOL UnLink() = 0;**

**Remarks:**

All helpers are independent - not linked to any other helper so that they can be moved and rotated without any other helper being affected.

**Prototype:**

**virtual BOOL CreateHelpers(int knotCt) = 0;**

**Remarks:**

Helpers are not automatically added to the spline on the assignment of the modifier. To do that the user need to press the "**Create Helpers**" button.

## Class MaxBakeElement

See Also: [Class IRenderElement](#) , [Class MaxRenderElement](#)

```
class MaxBakeElement : public MaxRenderElement
```

### Description:

**This class is only available in release 5 or later.**

The class provides the interfaces for creating a Texture Baking plugins in 3ds max. The texture baking plugins are controlled through the Maxscript “Render To Texture”. There are methods available in this class that provide the ability to produce a dynamic UI in Maxscript.

This class provides the basic infrastructure for the UI and Maxscript interaction, to fully understand the workings of these plugins please look at the RenderElement samples

For an example of this class in use by **Texture Baking** plug-ins see  
`\MAXSDK\SAMPLES\RENDER\RENDERELEMENTS\INCLUDE\`

All methods of this class are implemented by the plug-in.

### Methods:

#### Prototype:

```
virtual void SetOutputSz( int xSz, int ySz ) = 0
```

### Remarks:

Set the size of the bitmap to be created

### Parameters:

**int xSz**

Specifies the width of the output.

**int ySz**

Specifies the height of the output.

**Prototype:**

**virtual void GetOutputSz( int& xSz, int &ySz ) = 0**

**Remarks:**

Get the size of the bitmap to be created

**Parameters:**

**int& xSz**

The new width

**int& ySz**

the new height

**Prototype:**

**virtual void SetFileName( TCHAR\* newName) = 0**

**Remarks:**

This is the name used in the UI, so it does not consist of the path – just the actual file name

**Parameters:**

**TCHAR\* newName**

The name to set for the UI

**Prototype:**

**virtual TCHAR \* GetFileName()const = 0**

**Remarks:**

Returns the actual filename. See remark above

**Prototype:**

**virtual void SetFileType( TCHAR\* newType) = 0**

**Remarks:**

This is full name of the file including path, that is used by the renderer to actually create the bitmap to store the baked element. This should not really be set as it is created dynamically for sole use by the renderer.

**Parameters:**

**TCHAR\* newType**

The name for the full file name

**Prototype:**

**virtual const TCHAR\* GetFileType() const = 0**

**Remarks:**

Returns the filename used for this baked element – See comment above

**Prototype:**

**virtual void SetFileNameUnique(BOOL on) = 0**

**Remarks:****Prototype:**

**virtual BOOL IsFileNameUnique() const = 0**

**Remarks:****Prototype:**

**virtual void SetRenderBitmap( Bitmap\* pBitmap ) = 0**

**Remarks:**

The render will create a bitmap from the data available and will pass the Bitmap to the plugin. This will be the Bitmap the max actually uses to render to.

**Parameters:**

**Bitmap\* pBitmap**

The bitmap used for the output

**Prototype:**

**virtual Bitmap\* GetRenderBitmap() const = 0**

**Remarks:**

Returns the bitmap containing the final output

**Prototype:**

**virtual void SetLightApplied(BOOL on) = 0**

**Remarks:**

This defines whether the baker uses lighting or not. For example a diffuse texture baker can specify whether the result uses lighting or not.

**Parameters:**

**BOOL on**

The state for the lighting parameter

**Prototype:**

**virtual BOOL IsLightApplied() const = 0**

**Remarks:**

Returns whether lighting is used in the calculation

**Prototype:**

**virtual void SetShadowApplied(BOOL on) = 0**

**Remarks:**

This defines whether the baker uses shadows or not. For example a diffuse texture bake can specify whether the result uses shadows or not.

**Parameters:**

**BOOL on**

The state for the shadow parameter

**Prototype:**

**virtual BOOL IsShadowApplied() const = 0**

**Remarks:**

Returns whether shadows are used in the calculation

**Prototype:**

**virtual void SetAtmosphereApplied(BOOL on) = 0**

**Remarks:**

Defines whether the texture baking uses Atmosphere in the final result.

**Parameters:**

**BOOL on**

Specifies whether Atmospheres are used or not.

**Prototype:**

**virtual BOOL IsAtmosphereApplied() const = 0**

**Remarks:**

Returns whether Atmospheres are used. Most texture baking plugins would return false.

**The following methods are used for the creation of the Maxscript UI in the Render To Texture tool**

**Prototype:**

**virtual int GetNParams() const = 0**

**Remarks:**

Return the number of parameters to be exposed to the UI

**Prototype:**

**virtual const TCHAR\* GetParamName( int nParam ) = 0**

**Remarks:**

Returns the name used by maxscript for the parameter value passed in. This is a 1 based access, so if GetNParams returned 8 the valid range for GetParamName will be 1 to 8.

**Parameters:**

**int nParam**

The value for the parameter whose name is to be returned.

**Prototype:**

**virtual const int FindParamByName( TCHAR\* name ) = 0**

**Remarks:**

Returns the index of the parameter based on the name

```
const int FindParamByName1( TCHAR* name ) {  
    for( int i = 0; i < 8; ++i ){  
        if( strcmp( name, GetString( paramNames[i] )) == 0 )
```

```
    return i+1;
}
return 0;
}
```

**Prototype:**

**virtual int GetParamType( int nParam ) = 0**

**Remarks:**

Currently only 1 = Boolean, 0 = Undefined are supported. In release 5 this means that any parameter you have is defined and used as a Boolean in maxscript

**Parameters:**

**int nParam**

The value for the parameter whose type is to be returned.

**Prototype:**

**virtual int GetParamValue( int nParam ) = 0**

**Remarks:**

The value to set when the Render to Texture tool is loaded

**Parameters:**

**int nParam**

The value for the parameter whose value is to be returned.

**Prototype:**

**virtual void SetParamValue( int nParam, int newVal ) = 0**

**Remarks:**

The Render to Texture Tool will save out the data

**Parameters:**

**int nParam**

The value for the parameter whose value is to be set

**int newVal**

The value to set

## Class IViewportShaderManager

See Also: : [Class IDXShaderManagerInterface](#)

```
class IViewportShaderManager : public FPStaticInterface
```

### Description:

**This class is only available in release 5 or later.**

The class provides access to the Viewport Shaders the Viewport manager hosts.

### Methods:

#### Prototype:

```
virtual int GetNumEffects()=0
```

#### Remarks:

This will return the number of viewport shaders the ViewportManager has in its database.

#### Prototype:

```
virtual ReferenceTarget* GetActiveEffect()=0
```

#### Remarks:

This will return the effect that is currently active in the viewport. This ReferenceTarget can be queried for any Interfaces it may be hosting including

.

The following code can be used to get an IDXDataBridge interface where effect is a ReferenceTarget

```
IDXDataBridge * vp = (IDXDataBridge*)effect->GetInterface(VIEWPORT_SHADER_CLIENT_INTERFACE);
```

#### Prototype:

**virtual TCHAR \* GetEffectName(int i)=0**

**Remarks:**

This will return the name of the effect based on the index passed in. This is a 1 based index, so that it can be used from Maxscript directly. The reason for this is that internally position "0" is reserved for the "None" element in the drop down list.

**Parameters:**

**int i**

The index of the effect whose name will be returned

**Prototype:**

**virtual ReferenceTarget \* SetViewportEffect(int i)=0**

**Remarks:**

This will set the active effect and return a pointer to it for further setup etc

**Parameters:**

**int i**

The index of the effect the set.

**Prototype:**

**virtual void ActivateEffect(MtlBase \* mtl, BOOL State)=0**

**Remarks:**

For a shader to be initialized and allowed to draw in the viewport the material needs to be told that it contains a Hardware Shader (or effect). This method controls this flag.

**Parameters:**

**MtlBase \* mtl**

The material whose flag is to be set

## **BOOL State**

True if the flag is to be set or FALSE to clear it

## Class IDXDataBridge

See Also: : [Class IDXShaderManagerInterface](#), [Class IViewportShaderManager](#), [Class IHardwareMaterial](#)

```
class IDXDataBridge: public BaseInterface
```

### Description:

**This class is only available in release 5 or later.**

This is the interface used by the Viewport Shader plugins. For the manager to load up a Viewport Shader it must implement this class.

As well as implementing the dialog loading it also provides a link to the GFX. This means the developer is free use IDX8VertexShader and IDX8PixelShader interfaces, for more advanced shaders. Examples of these shaders are

`\MAXSDK\SAMPLES\HardwareShaders\Cubemap\Cubemap.cpp`

and

`\MAXSDK\SAMPLES\HardwareShaders\Membrane\Membrane.cpp`

**An important point to make with the usage of these plugins is in the way the ViewportManager loads them up. To determine whether the plugin is a shader it checks the Category in the ClassDesc. This must return “DXViewportEffect” otherwise it will not be listed in the database. The reason for this is that the Manager supports deferred loading, and this is one way of checking the plugin without requiring a full DLL load.**

For an example of this class in use by **ViewportShader** plug-ins see

`\MAXSDK\SAMPLES\HardwareShaders\LightMap\Lightmap.cpp`

If the Viewport Shader plugin hosts any animated parameters that will be viewed in the Trackview then it is important the plugin implements the following Reference Message in the

```

RefResult LightMap::NotifyRefChanged(Interval changeInt,
RefTargetHandle hTarget, PartID& partID, RefMessage message )
{
  switch (message)
  {
    //this allows the manager to control the TV better.
    case REFMSG_WANT_SHOWPARAMLEVEL:
    {
      BOOL * pb = (BOOL*)(partID);
      *pb = TRUE;
      return REF_STOP;
    }
  }
  return(REF_SUCCEED);
}

```

This will make sure that the Parameters show up under the Viewport Manager in the correct format.

All methods of this class are implemented by the plug-in.

### **Methods:**

### **Prototype:**

**virtual Interface\_ID GetID()**

### **Remarks:**

The returns the unique ID for this interface. By default it will return VIEWPORT\_SHADER\_CLIENT\_INTERFACE

### **Prototype:**

**virtual ParamDlg \* CreateEffectDlg(HWND hWnd, IMtlParams \* imp)= 0**

**Remarks:**

This allows the Viewport shader to create a UI. This will be added as a child of the Viewport Manager.

**Parameters:**

**HWND hWnd**

The window handle to the parent window

**IMtlParams \* imp**

This can be used in the call to CreateParamsDlg.

**Prototype:**

**virtual void DisableUI()=0**

**Remarks:**

Currently this method is not used

**Prototype:**

**virtual TCHAR \* GetName()=0**

**Remarks:**

This allows the plugin to return a name to be used in max. This may be extended for future use, but currently it is only used in the trackview.

**Prototype:**

**virtual void SetDXData(IHardwareMaterial \* pHWMTl, Mtl \* pMtl)=0**

**Remarks:**

This allows the plugin to use methods of [Class IHardwareMaterial](#) to perform custom drawing of an object. This could be special texture setups and control over the Render States and Texture Stages. If the developer needs finer control

over the object, then the r4 interfaces `IDX8VertShader` can be used instead, and this function will not do anything. This method will be called whenever the object needs drawing. For a detailed example of this usage, please look at the Lightmap sample in the sdk

**Parameters:**

**`IHardwareMaterial * pHWMTl`**

Provides access to various drawing techniques

**`Mtl * pMtl`**

A pointer to the parent material.

# Class IHardwareMaterial

See Also: [Class IDXDataBridge](#)

class IHardwareMaterial: public BaseInterface

## Description:

**This class is only available in release 5 or later.**

A pointer to this class is provided by **IDXDataBridge::SetDXData()**. The GFX layer will implement all these methods. Most of the methods are direct replicas of the DirectX API for SetRenderState and SetTextureStageState.

The reason for this is that only one thread can access the graphics hardware at any one time. Using this interface means that the GFX driver can have its database initialized with the DirectX states required for this object, when it comes to access the graphics device.

To use this class good knowledge of DirectX is needed, a copy of the DirectX documentation is also recommended.

For an example usage of this class see

**MAXSDK\SAMPLES\HardwareShaders\LightMap\Lightmap.cpp**

## Methods:

### Prototype:

**virtual Interface\_ID GetID()**

### Remarks:

The returns the unique ID for this interface. By default it will return **IHardware\_Material\_Interface\_ID**

### Prototype:

**virtual void SetFillMode(DWORD mode) = 0**

### Remarks:

This is equivalent to the DirectX method SetRenderState(FILLMODE,mode)

**Parameters:**

**DWORD mode**

A member of D3DFILLMODE

**Prototype:**

**virtual void SetShadeMode(DWORD mode) = 0**

**Remarks:**

This is equivalent to DirectX method SetRenderState(SHADEMODE,mode)

**Parameters:**

**DWORD mode**

A member of D3DSHADEMODE

**Prototype:**

**virtual void SetMaterial(LPD3DXMATERIAL pMtl) = 0**

**Remarks:**

Please see the DirectX documentation for more information on this method

**Prototype:**

**virtual void SetDiffuseColor(LPD3DXCOLOR pClr) = 0**

**Remarks:**

Specifies Diffuse color to be set by using the DirectX data structure **LPD3DXCOLOR**.

**Prototype:**

**virtual void SetDiffuseColor(Color c, float alpha = 1.0f) = 0**

**Remarks:**

Allows the Diffuse color to be set

**Parameters:**

**Color c**

The diffuse color

**float alpha**

The colors alpha

**Prototype:**

**virtual void SetDiffuseColor(Point3 c, float alpha = 1.0f) = 0**

**Remarks:**

Allows the Diffuse color to be set

**Parameters:**

**Point3 c**

The diffuse color

**float alpha**

The colors alpha

**Prototype:**

**virtual void SetAmbientColor(LPD3DXCOLOR pClr) = 0**

**Remarks:**

Specifies the Ambient color to set by using a DirectX data structure.

**Prototype:**

**virtual void SetAmbientColor(Color c, float alpha = 1.0f) = 0**

**Remarks:**

Allows the Ambient color to be set

**Parameters:**

**Color c**

The ambient color

**float alpha**

The colors alpha

**Prototype:**

**virtual void SetAmbientColor(Point3 c, float alpha = 1.0f) = 0**

**Remarks:**

Allows the Ambient color to set

**Parameters:****Point3 c**

The ambient color

**float alpha**

The colors alpha

**Prototype:**

**virtual void SetSpecularColor(LPD3DXCOLOR pClr) = 0**

**Remarks:**

Specifies the Specular color to set by using a DirectX data structure.

**Prototype:**

**virtual void SetSpecularColor(Color c, float alpha = 1.0f) = 0**

**Remarks:**

Allows the Specular color to be set

**Parameters:****Color c**

The specular color

**float alpha**

The colors alpha

**Prototype:**

**virtual void SetSpecularColor(Point3 c, float alpha = 1.0f) = 0**

**Remarks:**

Allows the Specular color to set

**Parameters:**

**Point3 c**

The specular color

**float alpha**

The colors alpha

**Prototype:**

**virtual void SetEmissiveColor(LPD3DXCOLOR pClr) = 0**

**Remarks:**

Allows the Emissive color to set by using a DirectX data structure.

**Prototype:**

**virtual void SetEmissiveColor(Color c, float alpha = 1.0f) = 0**

**Remarks:**

Allows the Emissive color to be set

**Parameters:**

**Color c**

The emissive color

## **float alpha**

The colors alpha

### **Prototype:**

**virtual void SetEmissiveColor(Point3 c, float alpha = 1.0f) = 0**

### **Remarks:**

Allows the Emissive color to be set

### **Parameters:**

#### **Point3 c**

The emissive color

#### **float alpha**

The colors alpha

### **Prototype:**

**virtual void SetSpecularPower(float power) = 0**

### **Remarks:**

Allows the specular exponent of the material to be set

### **Parameters:**

#### **float power**

The specular amount

### **Prototype:**

**virtual void SetNumTexStages(DWORD numStages)= 0**

### **Remarks:**

This sets the internal size for the table that will hold the Texture information for the material. If you are not using any Texture stages then this should be set

to zero, otherwise it should match exactly the number of textures being used.

**Parameters:**

**DWORD numStages**

The number of Texture stages

**Prototype:**

**virtual bool SetTexture(DWORD stage, DWORD\_PTR pTexture) = 0**

**Remarks:**

This allows a texture to be loaded to the graphics device. In this case the texture is loaded/created by the GFX via calls to `IHardwareRenderer::BuildTexture()` which will return a `DWORD_PTR`, which is an internal representation of the texture. The texture was allocated it will return true.

**Parameters:**

**DWORD stage**

The stage to hold the texture

**DWORD\_PTR pTexture**

A texture pointer returned by `IHardwareRenderer::BuildTexture()`

**Prototype:**

**virtual bool SetTexture(DWORD stage, LPCSTR filename) = 0**

**Remarks:**

This allows a texture to be loaded to the graphics device. The file is assumed to exist. If successful it will return true.

**Parameters:**

**DWORD stage**

The stage to hold the texture

**LPCSTR filename**

A string containing the filename of the texture to load

**Prototype:**

**virtual bool SetTextureUVWSource(DWORD stage, DWORD type)  
= 0**

**Remarks:**

This specifies where the Texture Coordinates will be retrieved. Most of the time the mesh will supply them so the type would be UVSOURCE\_HWGEN. However a Viewport Shader could create them dynamically so would supply. If successful it will return true.

**Parameters:**

**DWORD stage**

The stage to set

**DWORD type**

The UVW Source. It can be any of the following :-

**UVSOURCE\_MESH**

**UVSOURCE\_XYZ**

**UVSOURCE\_MESH2**

**UVSOURCE\_WORLDXYZ**

**UVSOURCE\_FACEMAP**

**UVSOURCE\_HWGEN**

**Prototype:**

**virtual bool SetTextureMapChannel(DWORD stage, DWORD  
numChan) = 0**

**Remarks:**

This specifies what mapping channel from the mesh the Texture Coordinates will be retrieved. This is used in the stripping code so that the VertexBuffer is populated with the correct TVs. If successful it will return true.

**Parameters:**

**DWORD stage**

The stage to set

**DWORD type**

The mapping channel to use

**Prototype:**

**virtual bool SetTextureCoordIndex(DWORD stage, DWORD index)  
= 0**

**Remarks:**

This specifies what texture coordinate the stage will use. This value is used in the DirectX call **SetTextureStageState(stage, D3DTSS\_TEXCOORDINDEX, index)**. If successful it will return true.

**Parameters:**

**DWORD stage**

The stage to set

**DWORD index**

The texture coordinate index to use

**Prototype:**

**virtual bool SetTextureTransformFlag(DWORD stage, DWORD  
flag) = 0**

**Remarks:**

This specifies what texture flag the stage will use. This value is used in the DirectX call **SetTextureStageState(stage, D3DTSS\_TEXTURETRANSFORMFLAGS, flag)**. If successful it will return true.

**Parameters:**

**DWORD stage**

The stage to set

**DWORD flag**

The DirectX flag to set

**Prototype:**

**virtual bool SetTextureTransform(DWORD stage, LPD3DXMATRIX pTransform)= 0**

**Remarks:**

This specifies what texture flag the stage will use. This value is used in the DirectX call **SetTransform((D3DTRANSFORMSTATETYPE) (D3DTS\_TEXTURE0+stage), pTransform)**. If successful it will return true.

**Parameters:**

**DWORD stage**

The stage to set

**LPD3DXMATRIX pTransform**

The DirectX matrix to set

**Prototype:**

**virtual bool SetTextureColorOp(DWORD stage, DWORD colorOp)= 0**

**Remarks:**

**This method is a direct replica if the DirectX color operators used with D3DTSS\_COLOROP. Please refer to the DirectX documentation for further information**

**Parameters:**

**DWORD stage**

The stage to set

**DWORD colorOp**

A member of the DirectX enumerated type **D3DTEXTUREOP**

**Prototype:**

**virtual bool SetTextureColorOpArg(DWORD stage, DWORD argNum, DWORD colorArg) = 0**

**Remarks:**

This method is a direct replica of the DirectX Texture Argument flag used with D3DTSS\_COLOROPARG. The argNum defines which argument to set.  
**Please refer to the DirectX documentation for further information**

**Parameters:**

**DWORD stage**

The stage to set

**DWORD argNum**

The argument index to set. If this is set to 1 then **D3DTSS\_COLORPARG1** will be addressed

**DWORD colorArg**

The Argument to set.

**Prototype:**

**virtual bool SetTextureAlphaOp (DWORD stage, DWORD alphaArg)= 0**

**Remarks:**

This method is a direct replica of the DirectX alpha blending operators used with D3DTSS\_ALPHAOP. **Please refer to the DirectX documentation for further information**

**Parameters:**

**DWORD stage**

The stage to set

**DWORD alphaArg**

A member of the DirectX enumerated type **D3DTEXTUREOP**

**Prototype:**

**virtual bool SetTextureAlphaArg(DWORD stage, DWORD argNum, DWORD AlphaArg) = 0**

**Remarks:**

This method is a direct replica of the DirectX Texture Alpha Argument flag

used with D3DTSS\_APLHAPARG. The argNum defines which argument to set. **Please refer to the DirectX documentation for further information**

**Parameters:**

**DWORD stage**

The stage to set

**DWORD argNum**

The alpha argument index to set. If this is set to 1 then **D3DTSS\_ALPHAARG1** will be addressed

**DWORD colorArg**

The Alpha Argument to set.

**Prototype:**

**virtual bool SetVertexShader(DWORD\_PTR pVertexShader)=0**

**Remarks:**

This method allows a DirectX vertex shader to be loaded. It is used in conjunction with `IHardwareRenderer::LoadVertexShader` which will return a `DWORD_PTR` of internal storage for the shader. There are methods in [IHardwareRenderer](#) that provide a means to supply the constants used in the shader.

**Parameters:**

**DWORD\_PTR pVertexShader**

The vertex shader to load.

**Prototype:**

**virtual bool SetPixelShader(DWORD\_PTR pPixelShader)=0**

**Remarks:**

This method allows a DirectX pixel shader to be loaded. It is used in conjunction with `IHardwareRenderer::LoadPixelShader` which will return a

DWORD\_PTR of internal storage for the shader. There are methods [IHardwareRenderer](#) that provide a means to supply the constants used in the shader.

**Parameters:**

**DWORD\_PTR pPixelShader**

The vertex shader to load

**Prototype:**

**virtual bool SetEffect(DWORD\_PTR pEffect)=0**

**Remarks:**

This method allows a DirectX effect. It is used in conjunction with `IHardwareRenderer::LoadEffectFile` which will return a `DWORD_PTR` of internal storage for the file. There are methods in [IHardwareRenderer](#) that provide a means to connect the application to the effects file.

**Parameters:**

**DWORD\_PTR pEffect**

The effect file to load

## Class RotationValue

### Description:

**This class is only available in release 5 or later.**

This class is to hold different representations of the rotation. In particular, it holds rotation value as represented by Euler angles or quaternion. Different types of rotation controllers may use different representations. To avoid losing information due to converting from one representation to another, we can use RotationValue to hold the result.

For example, the Skin pose feature reads rotation of a node from the rotation controller and stores the result in RotationValue (c.f. `maxsdk\include\iSkinPose.h`).

```
// Assuming node is a valid pointer to INode.  
ISkinPose* skpose = ISkinPose::GetISkinPose(*node);  
// skpose should not be null for Max version 5.0 or later.  
RotationValue rv = skpose->SkinRot();
```

It is guaranteed that rv keeps the original representation of the controller.

Being asked of Euler angles, RotationValue will return 3 float numbers in the format of Point3. There must be an association between numbers and axes.

There are two classes of Euler angle types. In one class, the rotation axes are not repeated (non-repetitive). They are enum's from kXYZ to kZYX. In the other class, one of the rotation axes is repeated (repetitive). They are enum's from kXYX to kZXZ. For convenience,

enum `kReptd` is used to denote the starting one: `kRept == kXYX`.

For non-repetitive Euler angles, there are two well-defined methods to associate three ordered angles, to three axes.

First, we can associate angles with x-, y-, and z-, axes, respectively. The first angle, for example, is always associated with the x-axis, no matter where it appears in the Euler order. Suppose

**Point3 a(0.1, 0.2, 0.3)**

then `a.x` (`==0.1`), `a.y` (`==0.2`), `a.z` (`==0.3`), are the angles of the x-axis, y-axis, and z-axis, respectively, no matter whether the order (type) of the Euler angles is `kXYZ` or `kZXY`.

Let's call this way of association by axis (name).

Second, we can associate them by position: the first angle, from left, is always associated with the first axis in the Euler angle order. For examples, the first angle is applied to the x-axis for `kXYZ` and `kXZY`, but to the y-axis for `kYXZ` and `kYZX`, etc. Suppose `a` is a `Point3`, `a[0]` (`==a.x`), `a[1]` (`==a.y`), `a[2]` (`==a.z`), are the angles of the z-axis, x-axis, and y-axis, respectively, for Euler type `kZXY`.

Let's call this way of association by order.

For repetitive Euler type, the association by axis is ambiguous because one axis may appear twice in the Euler axes. In this case, "by order" is well defined.

This class uses the association of by axis for non-repetitive types and by order for repetitive type. Suppose,

**Point3 a = rv.Euler(RotationValue::kZZXZ) // repetitive Euler type**

Then, a[0] and a[2] are both applied to the Z axis, but a[0] corresponds to the first z-axis from left, a[2] corresponds to the second z-axis (third axis) from left, and a[1] corresponds to the x-axis.

## **Methods:**

### **Prototype:**

```
static bool IsEuler(int rep) { }
```

### **Remarks**

Conveniency (static) method to test whether an integer corresponds to an enum of rotation representations used in RotationValue of type Euler angles

### **Prototype:**

```
static bool IsRepetitive(int rep) { }
```

### **Remarks:**

Conveniency (static) method to test whether in cases where a call to **IsEuler(int rep)** is made, and the return value is of type Euler angles, this method call returns whether it has repetitive axes (such as **XYX**).

### **Prototype:**

```
static bool IsQuat(int rep) }
```

### **Remarks:**

Conveniency (static) method to test whether an integer corresponds to an enum of rotation representations used in RotationValue of type Quaternion.

**Prototype:**

```
void Set(const Point3& a, EulerType et) { }
```

**Remarks:**

Set an object of RotationValue to an Euler angle representation. Angles are assumed in radians. To set to Euler angles of x, y, z, of order XYZ, do, for example:

```
RotationValue a;
```

```
a.Set(Point3(x, y, z), RotationValue::kXYZ);
```

**Prototype:**

```
void Set(const Quat& q) { }
```

**Remarks:**

Set an object of RotationValue to a quaternion representation.

**Prototype:**

```
RotationValue();
```

**Remarks:**

Constructor

**Prototype:**

```
RotationValue(const Point3& a, EulerType et);
```

**Remarks:**

Constructor

**K Prototype:**

```
RotationValue(const Quat& q);
```

**Remarks:**

Constructor.

**Prototype:**

**RotationValue(const RotationValue& src);**

**Remarks:**

Copy constructor.

**Prototype:**

**Point3 Euler(EulerType et =kXYZ) const {}**

**Remarks:**

Used to get the rotation in specific representation. Suppose rv is a RotationValue, to get it in terms of Euler angles of order XYZ:

**rv.Euler();**

in order ZXY:

**rv.Euler(RotationValue::kZXY);**

or, to get it in quaternion:

**(Quat)rv;**

to get it in matrix form:

**(Matrix3)rv**

**Prototype:**

**operator Quat() const {}**

**Remarks:**

Please see method Euler() for fuller explanation.

**Prototype:**

**operator Matrix3() const;**

**Remarks:**

Please see method Euler() for fuller explanation.

**Prototype:**

**void PreApplyTo(Matrix3& m) const;**

**Remarks:**

Given a matrix, m, we can apply the rotation, rv, of RotationValue from left side (PreApplyTo)

**rv.PreApplyTo(m) == ((Matrix3)rv) \* m**

**Prototype:**

**void PostApplyTo(Matrix3& m) const;**

**Remarks:**

Given a matrix, m, we can apply the rotation, rv, of RotationValue from right side (PoseApplyTo)

**rv.PoseApplyTo(m) == m \* (Matrix3)rv**

**Prototype:**

**void PostRotate(const AngAxis& aa);**

**Remarks:**

To apply a rotation, aa, as represented as AngAxis to a RotationValue, rv, from the right side,

**rv.PostRotate(aa)**

The internal representation of rv after applying to it will not be change. Mathematically,

**(Matrix3)rv.PostRotate(aa) == ((Matrix3)rv) \* MatrixOf(aa)**

If rv is in Euler angles, this method will try to keep the Euler angles from jumping at the borders of (+/-)180 degrees.

**Prototype:**

```
int NativeRep() const { return mRep; }
```

**Remarks:**

Used to get the internal representation and returns the representation type. If it is a Euler angle type, the first three numbers of the Quat returned from GetNative() are to be interpreted as Euler angles.

**Prototype:**

```
Quat GetNative() const { return mQ; }
```

**Remarks:**

Used to get the internal representation and returns the actual float numbers.

## Class ISkinPose : public FPMixinInterface

### Description:

Interface class for setting and getting a special, non-animated, transformation pose, [SkinPose](#).

### Methods:

### Prototype:

**static ISkinPose\* GetISkinPose(INode& n)**

### Remarks:

Method to obtain the interface pointer for a given INode.

### Parameters:

**INode& n**

The node having the transformation pose.

**Return Value:**

**ISkinPose\***

A pointer to this interface class.

**Prototype:**

```
virtual Point3 SkinPos() const = 0;
```

**Remarks:**

Method to obtain the position part of the transformation pose.

**Return Value:**

**Point3**

The position of the pose expressed as 3-vector (Point3).

**Prototype:**

**virtual RotationValue SkinRot() const = 0;**

**Remarks:**

Method to obtain the rotation part of the transformation pose.

**Return Value:****RotationValue**

The rotation of the pose expressed as an RotationValue, which can be quaternion or Euler angle type.

**Prototype:**

**virtual ScaleValue SkinScale() const = 0;**

**Remarks:**

Method to obtain the scale part of the transformation pose.

**Return Value:**

**ScaleValue**

The scale of the pose expressed as possibly unequal values along the principal axes of a coordinate system whose orientation is defined by a quaternion.

**Prototype:**

```
virtual void SetSkinPos(const Point3&) = 0;
```

**Remarks:**

Method to set the pose position to a given point.

**Parameters:**

**Point3**

The desired point for the pose.

**Prototype:**

```
virtual void SetSkinRot(const RotationValue&) = 0;
```

**Remarks:**

Method to set the pose rotation to a given value.

**Parameters:****RotationValue**

The rotation of the pose expressed as an Euler angle or quaternion type.

**Prototype:**

```
virtual void SetSkinRot(const Point3&) = 0;
```

**Remarks:**

Method to set the pose rotation to a given Euler angle.

**Parameters:****Point3**

The rotation of the pose expressed as an Euler angle vector.

**Prototype:**

```
virtual void SetSkinScaleFactors(const Point3&) = 0;
```

**Remarks:**

Method to set the potentially differing pose scale factors.

**Parameters:**

**Point3**

The point containing the scale factors x, y, and z.

**Prototype:**

**virtual void SetSkinScaleOrient(const Quat&) = 0;**

**Remarks:**

Method to set the orientation of the scale factor axes.

**Parameters:**

**Quat&**

The quaternion specifying the orientation of the scale factor axes.

**Prototype:**

**virtual bool IsSkinPosEnabled() const = 0;**

**Remarks:**

Method to determine if the pose position is enabled.

**Return Value:**

**bool**

If true, the pose position is enabled.

If false, the pose position is disabled.

**Prototype:**

**virtual bool IsSkinRotEnabled() const = 0;**

**Remarks:**

Method to determine if the pose rotation is enabled.

**Return Value:**

**bool**

If true, the pose rotation is enabled.

If false, the pose rotation is disabled.

**Prototype:**

**virtual bool IsSkinScaleEnabled() const = 0;**

**Remarks:**

Method to determine if the pose scale is enabled.

**Return Value:**

**bool**

If true, the pose scale is enabled.

If false, the pose scale is disabled.

**Prototype:**

**virtual bool SkinPoseMode() const = 0;**

**Remarks:**

Member function yielding a Boolean whether the node is in the Skin Pose mode. In this mode, the node will assume the skin pose, subject to “enabled” flags of the three (position, rotation, and scale) parts, as its transformation,

rather than from the normal channel, the transform controller.

**Return Value:**

**bool**

If true, the node transformation is in the skin pose mode.

If false, it is in the normal mode.

**Prototype:**

**virtual void EnableSkinPos(bool) = 0;**

**Remarks:**

Sets the state of the pose position transform.

**Parameters:**

**bool**

If true, the pose position transform is enabled.

If false, the pose position transform is disabled.

**Prototype:**

**virtual void EnableSkinRot(bool) = 0;**

**Remarks:**

Sets the state of the pose rotation transform.

**Parameters:**

**bool**

If true, the pose rotation transform is enabled.

If false, the pose rotation transform is disabled.

**Prototype:**

**virtual void EnableSkinScale(bool) = 0;**

**Remarks:**

Sets the state of the pose scale transform.

**Parameters:**

**bool**

If true, the pose scale transform is enabled.

If false, the pose scale transform is disabled.

**Prototype:**

**virtual void SetSkinPoseMode(bool) = 0;**

**Remarks:**

Puts the node transform in the skin pose or normal mode.

**Parameters:**

**bool**

If true, the node transform is put in the skin pose mode.

If false, the node transform resumes to the normal mode.

**Prototype:**

**virtual void SetSkinPose(TimeValue) = 0;**

**Remarks:**

Sets the state of all three non-animated skin pose transforms, subject to the

“enabled” flags, to the animated normal pose at a particular time.

**Parameters:**

**TimeValue**

Time at which the animated normal pose is used as the target to set the skin pose transforms.

**Prototype:**

**virtual void AssumeSkinPose(TimeValue) = 0;**

**Remarks:**

Sets the state of all three animated normal pose transforms, subject to the “enabled” flags, at a particular time, to the non-animated skin pose transforms.

**Parameters:**

**TimeValue**

Time at which to set the normal pose transforms.

**Prototype:**

**virtual void TMSetValue(TimeValue, SetXFormPacket&) = 0;**

**Remarks:**

This is a utility method used to set value to the node transform. According to whether it is in the skin pose mode and the three “enabled” flags, it will set value to the TM controller or the skin pose transforms.

**Parameters:**

**TimeValue**

Time at which to set the pose transform.

## **SetXFormPacket&**

Controller values for the transform.

### **Prototype:**

**Point3 SkinRotAngles() const;**

### **Remarks:**

Method to obtain the Euler angles of the pose rotation.

### **Return Value:**

**Point3**

The Euler angles about x, y, and z.

### **Prototype:**

**Point3 SkinScaleFactors () const;**

### **Remarks:**

Method to obtain the scale factors of the pose transformation.

### **Return Value:**

**Point3**

The scale factors along the x, y, and z axes.

### **Prototype:**

**Quat SkinScaleOrient() const;**

**Remarks:**

Method to obtain the scale factor orientation of the pose transformation.

**Return Value:****Quat**

The axis and angle of the scale factor orientation.

**Prototype:**

**void SetSkinScale(const ScaleValue&);**

**Remarks:**

Method to set the scale values for the pose transform.

**Parameters:****ScaleValue&**

The scale of the pose expressed as possibly unequal values along the principal axes of a coordinate system whose orientation is defined by a quaternion.

**Prototype:**

**void SetSkinRotAngles(const Point3&);**

**Remarks:**

Method to set the rotation angles for the pose transform.

**Parameters:****Point3&**

The Euler angles of the pose rotation.

**Prototype:**

**bool ShowSkinPos() const;**

**Remarks:**

Method to determine if the pose position component is enabled and the node is in the skin pose mode.

**Return Value:**

**bool**

If true, the position of the node transform will come from the skin pose.

If false, the position of the node transform comes from the normal TM controller.

**Prototype:**

**bool ShowSkinRot() const;**

**Remarks:**

Method to determine if the pose rotation component is enabled and the node is in the skin pose mode.

**Return Value:**

**bool**

If true, the rotation of the node transform will come from the skin pose.

If false, the rotation of the node transform comes from the normal TM controller.

**Prototype:**

**bool ShowSkinScale() const;**

**Remarks:**

Method to determine if the pose scale component is enabled and the node is in the skin pose mode.

**Return Value:****bool**

If true, the scale of the node transform will come from the skin pose.

If false, the scale of the node transform comes from the normal TM controller.

**Prototype:****bool IsACompEnabled() const;****Remarks:**

Method to determine if one or more pose components; position, rotation, or scale are enabled.

**Return Value:****bool**

If true, one or more components are enabled.

If false, none of the components are enabled.

**Prototype:****virtual bool NeedToSave() const = 0;****Remarks:**

Method to indicate that a post transform component has changed from the default and needs to be saved.

**Return Value:**

**bool**

If true, a pose component has changed.

If false, no pose component has changed.

**Prototype:**

```
virtual IOResult Save(ISave*) const = 0;
```

**Remarks:**

Method to write pose data to a file.

**Parameters:**

**ISave\***

Pointer for use in calling write methods.

**Return Value:****IOResult**

If IO\_OK, the method succeeded.

If IO\_ERROR, the method was unsuccessful.

**Prototype:**

**virtual IOResult Load(ILoad\*) = 0;**

**Remarks:**

Method to read pose data from a file.

**Parameters:**

**ILoad\***

Pointer for use in calling read methods.

**Return Value:****IOResult**

If IO\_OK, the method succeeded.

If IO\_ERROR, the method was unsuccessful.

**Prototype:**

**virtual void Copy(const ISkinPose&) = 0;**

**Remarks:**

Method to copy data members from an existing ISkinPose instance to the current one.

**Parameters:**

**ISkinPose&**

Reference to instance of this class to copy from.

**Prototype:**

```
virtual const void* ObjectOf(void*) const = 0;
```

**Remarks:**

Determines whether this is a const object of a particular subclass derived from ISkinPose. It is used for the internal implementation purpose.

**Parameters:**

**void\***

Pointer to the subclass identifier to test.

**Return Value:**

**void\***

Const pointer to subclass.

**Prototype:**

**virtual void\* ObjectOf(void\*) const = 0;**

**Remarks:**

Determines whether this is an object of a particular subclass derived from ISkinPose. It is used for the internal implementation purpose.

**Parameters:**

**void\***

Pointer to the subclass identifier to test.

**Return Value:**

**void\***

Pointer to subclass.

**Prototype:**

**FPInterfaceDesc\* GetDesc();**

**Remarks:**

Method to obtain the function publishing interface description.

**Return Value:**

**FPInterfaceDesc \***

Pointer to the interface descriptor.

## Class IUnwrapMod2

### Description:

**This class is only available in release 5 or later.**

The new class allows for Normal, Flatten, and Unfold mapping. You can bring them up through a dialog or through a script command. All these tools basically work the same. They are either applied to the current selected faces or the whole object if no faces are selected.

As this class has been fully developed with the new Function-Published System (FPS), all of its methods have a one-to-one correspondence with MaxScript commands. For the sake of brevity, references to MaxScript commands mean that there is an attendant C++ method in iunwrap.h that is prepended by lowercase **fn**. Examples of this are as follows ( MaxScript and its attendant C++ method):

copy **fpCopy()**;

normalMapNoParams **fnNormalMapNoParams()**.

There are three distinct modes of unmapping:

### Normal Mapping

Normal Mapping is mapping based solely are the normals provided. This is identically to box mapping except you customize what normals you want to project on. There are three methods to apply a "normalMap" which applies a normal map based on the parameters passed in (see the methods below) , "normalMapNoParams" which applies a normal map using the default setting, and "normalMapDialog" which brings up a dialog that lets you set the setting. Right now there are 6 types of default mapping Back/Front, Left/Right, Top/Bottom, Box Mapping No Top, Box Mapping, and Diamond mapping. Just apply the mapping to a teapot to see the differences they are pretty obvious once you see the effect. Parameters in the Normal Mapping dialog are:

- Mapping Type which are the 6 mapping types listed above.
- Spacing which determines how much space there is between each cluster.
- Normalize Cluster will normalize the cluster from 0 to 1
- Rotate Clusters will rotate the clusters so they take the least amount of area.
- Align By Width will sort the clusters by their width otherwise it will use their heights.

Hitting the Save As Default will take the current setting and set them as defaults for the next time you bring up the dialog and when you use the script command **"normalMapNoParams"**.

### **Flatten Mapping**

Flatten Mapping is similar to normal mapping, except it uses an angle threshold to define the clusters and the clusters will always be contiguous faces. This type of mapping will generate mapping that does not overlap so it is useful for baking textures and lighting. Just like normal mapping this comes in 3 flavors "flattenMap" which applies a normal map based on the parameters passed in (see the methods below) , "flattenMapNoParams" which applies a map using the default setting, and "flattenMapDialog" which brings up a dialog that lets you set the setting. The parameters for the Flatten Map Dialog are:

Face Angle Threshold - when building contiguous faces this the angle used to determine whether a face is part of that cluster. The larger this angle the larger the cluster will be, but you will get more distortion since the texture faces area will start deviating from their geometric face area.

- Spacing which determines how much space there is between each cluster.
- Normalize Cluster will normalize the cluster from 0 to 1
- Rotate Clusters will rotate the clusters so they take the least amount of area.
- Fill Holes will fill clusters with holes. It places smaller cluster in the gaps of larger clusters.

## Unfold Mapping

Unfold Mapping where as the Normal and Flatten Mapping basically use a lot of planar mapping to get their results, this is an actual unfolding algorithm. It guarantees that all texture faces will have the exact same proportions as their geometric equivalents, but you may get faces that overlap. This type of mapping is only good on meshes that are very regular things like cylinders etc. This has the same type of function calls as the above "unfoldMap" which applies an unfold map based on the parameters passed in (see the methods below) , "unfoldMapNoParams" which applies a map using the default setting, and "unfoldMapDialog" which brings up a dialog that lets you set the setting. The parameters for the Unfold Map Dialog are:

- Unfold Type which consists of Walk to closest face and Walk to farthest face. This determines the order of which face gets unfolded. Under almost all conditions you want to walk to the closest face.
- Normalize Cluster will normalize the cluster from 0 to 1

Copy/Paste allows you to copy a texture face/faces from one part or mesh to another or to a new mesh. These functions are extremely topology dependant so if you copy faces onto faces that have a different topology or face order you will get unpredictable results. There is a "copy" script which takes the current selected faces and puts them in the copy buffer. There is the "Paste" and "PasteInstance" commands which do the pasting to current selected faces. "Paste" take one parameter called rotate which determines if every time you paste to the same face whether the tvs are reoriented. For instance if you paste a quad onto a another quad there are actually 4 possible ways you can paste it onto it. If rotate is on, every time you paste it will go onto the next variation. "Paste Instance" forces the faces that are being pastes to use the vertices that the copy buffer use (NOTE you cannot paste instance across objects). This similar to doing a regular paste and then selecting all the overlapping vertices and doing a weld. There is no rotate option with this method. As noted above Pasting faces that have a different topology or face order will result in unpredictable results.

Stitching allows you find all the texture vertices that are assigned to the same geometric vertex and bring them all to the same spot and weld them together. This allows you take faces that geometrically contiguous, but not texture face contiguous and line them up. There is a `stitchVerts` command which takes parameters, `stitchVertsNoParams` which uses the current defaults, and a `stitchVertsDialog` which brings up a dialog to apply a stitch and set defaults. The Stitch Tool params are:

- **Align Cluster** which if checked and the edges to be stitched are on separate clusters it will try to align the clusters and then stitch the vertices.
- **Bias** determines which direction the vertices will move (to or from the source). At a Bias of 0.0 the vertices will move to the source and 1.0 they will move to the target.

Note stitching will respect the soft selection.

## **Methods:**

*Nb: the following are MaxScript commands, however, referring to the above qualifier, their C++ method name is the same as in MaxScript but prepended with `fn`.*

*The format for input/output parameters is as follows:*

*<output type> <name of function> <input type params 1...n>*

**copy** - this takes the selected faces and places them in the copy buffer

**paste**

rotate TYPE\_BOOL if this on, every time you paste to the same selection it will try a different variation

This paste the current copy buffer onto the current face selection.

**pasteInstance** this forces the faces that are being pastes to use the vertices that

the copy buffer use (NOTE you cannot paste instance across objects). This is similar to doing a regular paste and then selecting all the overlapping vertices and doing a weld. There is no rotate option with this method.

### **SetDebugLevel**

level TYPE\_INT level of debug info 0 means no debug info, the higher the value the more spam you will see in your script window  
Debugging tool so I can control the amount of debug info that goes to the listener and script window.

**TYPE\_BOOL getTileMap** - returns whether the background is tiled.

### **setTileMap**

tile TYPE\_BOOL - whether or not to tile the background  
This allows you to set the tile state of the background

**TYPE\_INT getTileMapLimit** - returns the max number of tiles to use in a direction

### **setTileMapLimit**

limit TYPE\_INT - the number of tile to limit in a direction  
This allows you to set the tile limit

**TYPE\_FLOAT getTileMapBrightness** returns the brightness of the tiles.

### **setTileMapBrightness** brightness

TYPE\_FLOAT - the brightness of the tiled maps  
This allows you to set the brightness of the tiled maps

New Maxscript functions

**TYPE\_BOOL getShowMap** returns the state of the show map button

### **TYPE\_VOID setShowMap**

showMap TYPE\_BOOL the state you want to set the show map button  
Lets you set the state of the show map button

**TYPE\_BOOL getLimitSoftSel** returns whether the soft selection limit is on/off

### **SetLimitSoftSel**

limit TYPE\_BOOL state the to set the soft selection limit  
Allows you to set the soft selection limit state

**TYPE\_INT getLimitSoftSelRange** returns the edge limit for the for the soft selection

### **setLimitSoftSelRange**

range TYPE\_INT this is how far out in edges that soft selection will expand to  
This lets you set the edge limit range for soft selection

### **TYPE\_FLOAT getVertexWeight**

index TYPE\_INT the index of the vertex you want to inspect  
returns the soft selection weight of a particular vertex

### **setVertexWeight**

index TYPE\_INT the index of the vertex you want to change  
weight TYPE\_FLOAT the soft selection weight you want to set the vertex to

This lets you set the soft selection weight of a particular vertex. Note once you set the weight of a vertex, it is tagged as being modified and will not change value unless you unmodify or call setVertexWeight on it again.

### **TYPE\_BOOL isWeightModified**

index TYPE\_INT the index of the vertex you want to inspect  
This returns whether a vertex is modified or not.

### **modifyWeight**

index TYPE\_INT the index of the vertex you want to change  
modify TYPE\_BOOL the modified state of the vertex

This lets you set the modified state of vertex. If a vertex is modified it ignores regular UI soft selection and the vertex soft selection weight is locked to its current state and can only be changed by the `setVertexWeight` method.

**TYPE\_BOOL getGeomSelectElementMode** returns whether you are in element mode for face selection in the viewport.

### **setGeomSelectElementMode**

mode TYPE\_BOOL - the state that you want to put viewport selection in  
Lets you set the viewport element mode.

**TYPE\_BOOL getGeomPlanarThresholdMode** return the whether you are in planar select mode.

### **setGeomPlanarThresholdMode**

mode TYPE\_BOOL the state that you want to put planar selection mode  
Lets set the planar selection mode.

**TYPE\_FLOAT getGeomPlanarThreshold** return the planar selection angle threshold

### **setGeomPlanarThreshold**

angle TYPE\_FLOAT the angle threshold  
Lets you set the angle threshold for the planar selection mode.

**TYPE\_INT getWindowX** returns the current X position of the Unwrap Edit window

**TYPE\_INT getWindowY** returns the current Y position of the Unwrap Edit window

**TYPE\_INT getWindowW** returns the current width of the Unwrap Edit window

**TYPE\_INT getWindowH** returns the current height of the Unwrap Edit window

**TYPE\_BOOL getIgnoreBackFaceCull** returns the state of the Ignore Back Face mode

### **setIgnoreBackFaceCull**

ignoreBackFaceCull TYPE\_BOOL state of the Ignore Back Faces mode  
Lets you set the Ignore Back Face mode

**TYPE\_BOOL getOldSelMethod** returns whether the system is in the old selection mode. Where drag selection always uses back faces and single pick mode ignore back faces.

### **SetOldSelMethod**

oldSelMethod TYPE\_BOOL the state of the old selection method  
This lets you set the system back to the old selection method. Where drag selection always uses back faces and single pick mode ignore back faces.  
This will override the Ignore Back Faces mode.

### **SelectByMatID**

matID TYPE\_INT the matID of the face that you want to select  
This lets you select faces by material ids.

### **selectBySG**

sg TYPE\_INT the smoothing group that you want to select  
This lets you select faces by smoothing group

**TYPE\_VOID expandGeomFaceSelection** - expands your current viewport face selection

**TYPE\_VOID contractGeomFaceSelection** - contracts your current viewport face selection

**TYPE\_BOOL getAlwaysEdit** - This returns whether the always edit mode is on. This mode will always bring up the edit dialog when the Unwrap rollup

window is displayed.

### **TYPE\_VOID setAlwaysEdit**

always TYPE\_BOOL the state that you want to set the always edit mode to.

This lets you set the always edit mode. This mode will always bring up the edit dialog when the Unwrap rollup window is displayed.

**TYPE\_BOOL getShowVertexConnections** this returns whether vertex connection indices are displayed. Vertex Connections are TV vertices that share the same geometric vertices

### **TYPE\_VOID setShowVertexConnections**

show TYPE\_BOOL whether to display the vertex connection data.  
This lets you toggle the vertex connection data.

**TYPE\_BOOL getFilterSelected** this returns the state of the Filter Selected Faces button

### **TYPE\_VOID setFilterSelected**

filter TYPE\_BOOL the filter state  
This lets you set the Filter Selected Faces button

**TYPE\_BOOL getSnap** this returns the snap state.

### **TYPE\_VOID setSnap**

snap TYPE\_BOOL the snap state  
This lets you set the snap state.

**TYPE\_BOOL getLock** this returns the lock selection state

### **TYPE\_VOID setLock**

lock TYPE\_BOOL state of the lock selection.  
This lets you set the state of the lock selection

### **TYPE\_VOID pack**

method `TYPE_INT` - 0 is a linear packing algorithm fast but not that efficient, 1 is a recursive algorithm slower but more efficient.  
spacing `TYPE_FLOAT` - the gap between cluster in percentage of the edge distance of the square  
normalize `TYPE_BOOL` - whether the clusters will be fit to 0 to 1 space.  
rotate `TYPE_BOOL` - whether a cluster will be rotated so it takes up less space.  
fillholes `TYPE_BOOL` - whether smaller clusters will be put in the holes of the larger cluster.  
This lets you pack the texture vertex elements so that they fit within a square space.

**`TYPE_VOID packNoParams`** - this packs the clusters using the default parameters.

**`TYPE_VOID packDialog`** - this brings up a dialog that lets the user set the parameters and then packs the clusters.

**`TYPE_INT getTVSubObjectMode`** sets the current texture subobject mode  
1 vertices, 2 edges, 3 faces.

**`TYPE_VOID setTVSubObjectMode`**

mode `TYPE_INT` - the subobject mode 1 vertices, 2 edges, 3 faces.  
Lets you set the tv subobject mode.

**`TYPE_BITARRAY getSelectedFaces`** returns the selected face list

**`TYPE_VOID selectFaces`**

Selection `TYPE_BITARRAY` selection that you want to make the face selection

This lets you set the face selection

**`TYPE_BOOL IsFaceSelected`**

Index `TYPE_INT` the index of the face you want to check  
This lets you check to see if a face is selected.

**TYPE\_INT getFillMode** - returns the fill mode type for face selections. The fill modes are as follows.

**FILL\_MODE\_OFF 1**  
**FILL\_MODE\_SOLID 2**  
**FILL\_MODE\_BDIAGONAL 3**  
**FILL\_MODE\_CROSS 4**  
**FILL\_MODE\_DIAGCROSS 5**  
**FILL\_MODE\_FDIAGONAL 6**  
**FILL\_MODE\_HORIZONTAL 7**  
**FILL\_MODE\_VERTICAL 8**

**TYPE\_VOID setFillMode**

mode TYPE\_INT - the fill mode that you want to set  
This lets you set the fill mode for selected faces.

**MoveSelected, RotateSelected, RotateSelectedCenter, ScaleSelectedCenter, and ScaleSelected** are identical to their vertex counter parts but are applied to the current selection.

**TYPE\_VOID MoveSelected**

Offset TYPE\_POINT3

**TYPE\_VOID RotateSelectedCenter**

Angle TYPE\_FLOAT

**TYPE\_VOID RotateSelected**

Angle TYPE\_FLOAT

Axis TYPE\_POINT3

**TYPE\_VOID ScaleSelectedCenter**

Scale TYPE\_FLOAT

Dir TYPE\_INT

**TYPE\_VOID ScaleSelected**

Scale TYPE\_FLOAT

Dir TYPE\_INT

Axis TYPE\_POINT3

**TYPE\_BITARRAY getSelectedEdges** returns the selected edge list

### **TYPE\_VOID selectEdges**

Selection TYPE\_BITARRAY selection that you want to make the edge selection

This lets you set the edge selection

### **TYPE\_BOOL IsEdgeSelected**

Index TYPE\_INT the index of the edge you want to check

This lets you check to see if a face is selected.

**TYPE\_BOOL getDisplayOpenEdges** returns whether the open edges will display

### **TYPE\_VOID setDisplayOpenEdges**

displayOpenEdges TYPE\_BOOL the state of the open edge display

This lets you set the open edge display

**TYPE\_POINT3 getOpenEdgeColor** returns the color used for the open edges

### **TYPE\_VOID setOpenEdgeColor**

color TYPE\_POINT3 the color to be used for open edges

This lets you set the open edge color

**TYPE\_BOOL getUVEdgeMode** returns whether you are in the UV Edge Selection mode. This mode will try to automatically select all the U or V edges when you select an edge. Since this is based on edges, the regular tri mesh may produce incorrect results since the hidden edges are not taken into account.

### **TYPE\_VOID setUVEdgeMode**

uvEdgeMode TYPE\_BOOL the state of the you want to set the UV Edge mode

This lets you set the UV Edge mode

**TYPE\_VOID uvEdgeSelect** - this is a command that will take your current edge selection and try to expand out along the U and V directions. Works best when you only have one edge selected.

**TYPE\_BOOL getOpenEdgeMode** returns whether you are in the Open Edge Selection mode. This mode will try to automatically select all the opens edges when you select an open edge.

### **TYPE\_VOID setOpenEdgeMode**

uvOpenMode TYPE\_BOOL the state of the you want to set the Open Edge mode

This lets you set the Open Edge mode

**TYPE\_VOID openEdgeSelect** - this is a command that will take your current selection and try to expand all the open edges in it.

**TYPE\_VOID vertToEdgeSelect** - this command takes your vertex selection and converts it to the edge selection.

**TYPE\_VOID vertToFaceSelect** - this command takes your vertex selection and converts it to the face selection.

**TYPE\_VOID edgeToVertSelect** - this command takes your edge selection and converts it to the vertex selection.

**TYPE\_VOID edgeToFaceSelect** - this command takes your edge selection and converts it to the face selection.

**TYPE\_VOID faceToVertSelect** - this command takes your face selection and converts it to the vertex selection.

**TYPE\_VOID faceToEdgeSelect** - this command takes your face selection and converts it to the edge selection.

**TYPE\_BOOL getDisplayHiddenEdges** return whether hidden edges of a tri mesh are displayed.

### **TYPE\_VOID setDisplayHiddenEdges**

displayHiddenEdges TYPE\_BOOL - the state that you want to the hidden

display to be.

This lets you turn on/off whether tri mesh hidden edges are displayed.

**TYPE\_POINT3 getHandleColor** - returns the color that will be used to display patch handles.

**TYPE\_VOID setHandleColor**

color TYPE\_POINT3 - the color that you want to set patch handles to

This lets you set the color that will be used to display patch handles.

**TYPE\_BOOL getFreeFormMode** this toggle the free form mode on and off. This mode is similar to the Photoshops free form mode. You select any where inside the bounding rectangle to move the selection. You select the corners to scale the selection, and you you select the edge centers to rotate the selection. You can also move the center cross which is your rotation pivot point.

**TYPE\_VOID setFreeFormMode**

freeFormMode TYPE\_BOOL the state of the free form mode.

Lets you turn on/off the free from mode.

**TYPE\_POINT3 getFreeFormColor** returns the color of the free form gizmo.

**TYPE\_VOID setFreeFormColor**

color TYPE\_POINT3 the color that you want the gizmo to be

This lets you set the color of the Free Form Gizmo

**TYPE\_VOID ScaleSelectedXY**

ScaleX TYPE\_FLOAT the x scale factor

ScaleY TYPE\_FLOAT the y scale factor

Axis TYPE\_POINT3 the axis to scale around

This lets you nu scale the current selection around an axis

**TYPE\_VOID SnapPivot**

Pos TYPE\_INT - the pivot position where

1 is the center

- 2 is the lower left of the selection
- 3 is the lower center of the selection
- 4 is the lower right of the selection
- 5 is the right center of the selection
- 6 is the upper right of the selection
- 7 is the upper center of the selection
- 8 is the upper left of the selection
- 9 is the left center of the selection

This lets you quickly snap the free form gizmo pivot the bounding rectangle.

**TYPE\_POINT3 getPivotOffset** - returns the pivot offset of the free form gizmo. This is an offset from the center of the selection.

**TYPE\_VOID setPivotOffset**

offset TYPE\_POINT3 the offset of the free form gizmo pivot  
This lets you set the offset of the free form gizmo pivot.

**TYPE\_POINT3 fnGetSelCenter** this returns the selection center so you can compute the pivot offset from a world uv position.

**TYPE\_BOOL getPolygonMode**

This returns whether the polygon mode for sub object face mode is on. Polygon Mode will just select across triangles across hidden edges of a triangle.

**TYPE\_VOID setPolygonMode**

mode TYPE\_BOOL the state you want to set the Polygon Mode  
This lets you set the state the of the Polygon Mode

**TYPE\_VOID PolygonSelect**

This is command that will take your current selection and expand it to include all polygons.

**TYPE\_VOID sketch**

IndexList TYPE\_INT\_TAB the indices of the points you want to move.  
PositionList TYPE\_POINT3\_TAB the list of a points you want to align your vertices to.

This lets you align texture vertices to a series of points

### **TYPE\_VOID sketchNoParams**

This puts you in sketch mode using the default parameters.

### **TYPE\_VOID sketchDialog**

This brings up the sketch options dialog.

### **TYPE\_VOID sketchReverse**

This will reverse the order of the select vertices that are being used for sketch when you use the Use Current Selection option.

### **TYPE\_INT getHitSize**

This returns the hit size whne you do a single click in pixels.

### **TYPE\_VOID SetHitSize**

size TYPE\_INT the size you want to set the hit size to  
This lets you set the hit size of a single click.

### **TYPE\_BOOL getResetPivotOnSelection**

This will return whether the Transform Gizmo will reset when the selection is changed.

### **TYPE\_VOID SetResetPivotOnSelection**

reset TYPE\_BOOL the state you want to set the reset pivot on selection  
This lets you set the Reset the Pivot On Selection.

### **TYPE\_BOOL getAllowSelectionInsideGizmo**

This returns whether a user can select sub objects inside the gizmo or not. If this is FALSE the user is in move mode when inside the gizmo unless they are over the pivot. If this is TRUE the user is in move mode when they are inside and over a selected sub object otherwise they are in select mode.

### **TYPE\_VOID Set AllowSelectionInsideGizmo**

select TYPE\_BOOL - the state you want to set the Allow Selection Inside Gizmo to be

This lets you set the Allow Selection Inside Gizmo flag

### **TYPE\_VOID SaveCurrentSettingsAsDefault**

This takes the current state of Unwrap UVW and save it to the plugin cfg directory into a file called unwrapUVW.ini. The next time the user creates an Unwrap UVW modifier this ini file will be used to set the defaults.

### **TYPE\_VOID LoadDefault**

This will load the unwrapUVW.ini defaults into the current Unwrap UVW modifier.

### **TYPE\_BOOL getShowShared**

This returns whether shared sub objects are displayed. Shared sub objects are texture vertices or edges that share the same geometric vertex or edge.

### **TYPE\_VOID setShowShared**

select TYPE\_BOOL whether to display shared sub objects or not.  
This lets you toggle the Show Shared flag.

### **TYPE\_POINT3 getSharedColor**

This returns the color that will be used to show shared sub objects.

### **TYPE\_VOID setSharedColor**

color TYPE\_POINT3 the color to be used for shared sub objects  
This lets you set the color to be used for shared sub objects.

### **TYPE\_VOID showIcon**

index TYPE\_INT index of the icon to be display/hidden. The icons are as follows:

- 1 - Move Mode
- 2 - Rotate Mode
- 3 - Scale Mode

- 4 - Transform Mode
- 5 - Mirror Tool
- 6 - Expand Selection
- 7 - Contract Selection
- 8 - Soft Selection Falloff
- 9 - Soft Selection Space
- 10 - Soft Selection Strength
- 11 - Break
- 12 - Target Weld
- 13 - Weld Selected
- 14 - Update Map
- 15 - Show Map
- 16 - UV/VW/UW space
- 17 - Properties Dialog
- 18 - Map Drop List
- 19 - U Spinner
- 20 - V Spinner
- 21 - W Spinner
- 22 - Lock Sub Object
- 23 - Hide/Show
- 24 - Freeze/Thaw
- 25 - Filter Selected
- 26 - Mat Ids
- 27 - Pan Mode
- 28 - Zoom Mode
- 29 - Zoom Region Mode
- 30 - Fit Command
- 31 - Snap

show TYPE\_BOOL whether to show or hide this icon

This lets you turn on/off icons in the Unwrap UVW Edit dialog

### **TYPE\_BOOL getSyncSelectionMode**

Returns whether the viewport and the dialog selections are synced

### **TYPE\_VOID setSyncSelectionMode**

sync TYPE\_BOOL

Lets you set whether the viewport and the dialog selections are synced

### **TYPE\_VOID syncTVSelection**

This is a command that syncs the dialog to the viewport

### **TYPE\_VOID syncGeomSelection**

This is a command that syncs the viewport to the dialog

### **TYPE\_POINT3 getBackgroundColor**

returns the color of the background in the dialog

### **TYPE\_VOID setBackgroundColor**

color TYPE\_POINT3

Lets you set the color of the background in the dialog

### **TYPE\_VOID updateMenuBar**

Forces the menu bar to update

### **TYPE\_BOOL getBrightnessAffectsCenterTile**

This returns whether the brightness value affects the center tile

### **TYPE\_VOID setBrightnessAffectsCenterTile**

bright TYPE\_BOOL

This lets you set whether the brightness value affects the center tile

### **TYPE\_BOOL getBlendTileToBackground**

This returns whether the tiled images are blended to the background color or black

### **TYPE\_VOID setBlendTileToBackground**

blend TYPE\_BOOL,

This lets you set whether the tiled images are blended to the background color or black

### **TYPE\_BOOL getPaintSelectMode**

This returns whether you are in paint select mode

**TYPE\_VOID setPaintSelectMode**

paint TYPE\_BOOL,

This lets you set whether you are in paint select mode

**TYPE\_INT getPaintSelectSize**

Returns the size of the paint select brush this is clamped between 1 and 15.

**TYPE\_VOID setPaintSelectSize**

size TYPE\_INT

This lets you set the size of the paint select brush this is clamped between 1 and 15.

**TYPE\_VOID PaintSelectIncSize**

This increments the brush size by one

**TYPE\_VOID PaintSelectDecSize**

This decrements the brush size by one

**TYPE\_INT GetTickSize**

Returns the size of a selected vertex tick

**TYPE\_VOID SetTickSize**

size TYPE\_INT size of the tick

This lets you set the size of a vertex tick

## Class INodeExposure

class INodeExposure: public FPMixinInterface

### Description:

**This class is only available in release 5 or later.**

This interface provides the ability for a node to define whether it is visible in any of max's dialog boxes. This interface will be extended and used by more of 3ds max's core utilities, but **currently ONLY TrackView and the Select Object/HideObject dialog box use this interface.** By default this interface is not available through the default nodes, it needs to be added.

To get a pointer to this interface from a node the following code can be used.

```
INodeExposure* iNE = (INodeExposure*)node->GetInterface(NODEEXPOSURE_INTERFACE_TOAPPEND)
```

This will add a new INodeExposure interface to the node if it is not present. The next time you use this technique it will only return the interface and not create another new interface.

### Data Members:

```
enum {  
    kSelectObjects, kSchematicView, kMaxscript,  
    kMerge, kMergeAnimation, kReplace,  
    kKeyEditor, kCurveEditor, kRangeEditor,  
};
```

This enum provides access to the different supported UI elements. It is used with various methods of the class to get/set the UI flags.

### Methods:

**Prototype:**

**bool IsExposedInTrackView() const**

**Remarks:**

This will return true if it is visible in TrackView or false if it is not. This will return TRUE is the node is exposed in **ANY** of the TrackView states, kKeyEditor, kCurveEditor and kRangeEditor. It will return FALSE if **ALL** are set to false.

**Prototype:**

**void SetExposedInTrackView(bool state)**

**Remarks:**

This allows the state of the TrackView exposure flag to be set by the user. This will set the flag for all three TrackView flags. See comment in IsExposedInTrackView().

**Prototype:**

**bool IsExposedInKeyEditor() const**

**Remarks:**

Specifies whether the node is visible in the Key Editor of TrackView

**Prototype:**

**void SetExposedInKeyEditor(bool state)**

**Remarks:**

This allows the state of the Key Editor of Trackview exposure flag to be set by the user.

**Prototype:**

**bool IsExposedInCurveEditor() const**

**Remarks:**

Specifies whether the node is visible in the Function Curve Editor of Track

View

**Prototype:**

**void SetExposedInCurveEditor(bool state)**

**Remarks:**

This allows the state of the Function Curve Editor of Track view exposure flag to be set by the user.

**Prototype:**

**bool IsExposedInRangeEditor() const**

**Remarks:**

Specifies whether the node is visible in the Key Range Editor of Track View

**Prototype:**

**void SetExposedInRnageEditor(bool state)**

**Remarks:**

This allows the state of the Key Range Editor of Track view exposure flag to be set by the user.

**Prototype:**

**bool IsExposedInSelectObjects() const**

**Remarks:**

This will return true if it is visible in Selected Objects/HideObjects Dialog box otherwise false

**Prototype:**

**void SetExposedInSelectObjects (bool state)**

**Remarks:**

This will set the flag for the exposure in Selected Objects/HideObjects dialog box

**Parameters:**

**bool state**

The value to set the flag

**Prototype:**

**virtual bool IsExposed(int ui) const =0**

**Remarks:**

This will return the exposure state of the UI element being queried

**Parameters:**

**int ui**

The UI flag to query. This should be a value from the UI enum – See data members section

**Prototype:**

**virtual void SetExposed(bool state) const =0**

**Remarks:**

This will set the state of all the UI elements to the state passed into the method

**Parameters:**

**bool state**

The state to set the nodes UI exposure

**Prototype:**

**virtual void SetExposed(bool state, int ui) const =0**

**Remarks:**

This will set the state of the individual UI element to the state passed into the method

**Parameters:**

**bool state**

The state to set the nodes UI exposure

**int ui**

The UI element to set

## Class INodeLayerProperties

```
class INodeLayerProperties : public FPMixinInterface
```

### Description:

**This class is only available in release 5 or later.**

This class defines an interface for accessing a node's global illumination properties.

An instance of this interface can be retrieved using the following line of code (assuming 'node' is of type INode\*):

```
static_cast<INodeGIProperties*>(node-  
>GetInterface(NODELAYERPROPERTIES_INTERFACE))  
#define NODELAYERPROPERTIES_INTERFACE  
Interface_ID(0x44e025f8, 0x6b071e44)
```

```
// Provides access to the nodes layer and bylayer bits
```

```
public:
```

```
virtual ILayerProperties* getLayer (void) = 0;
```

```
virtual void setLayer (FPInterface *) = 0;
```

```
virtual BOOL getDisplayByLayer () = 0;
```

```
virtual void setDisplayByLayer (BOOL) = 0;
```

```
virtual BOOL getRenderByLayer () = 0;
```

```
virtual void setRenderByLayer (BOOL) = 0;
```

```
virtual BOOL getMotionByLayer () = 0;
```

```
virtual void setMotionByLayer (BOOL) = 0;
```

```
virtual BOOL getColorByLayer () = 0;
```

```
virtual void setColorByLayer (BOOL) = 0;
```

```
virtual BOOL getGlobalIlluminationByLayer () = 0;
```

```
virtual void setGlobalIlluminationByLayer (BOOL) = 0;
```

```
};
```

## **Class IAssembly : public FPMixinInterface**

### **Description:**

This interface class allows for setting and retrieving assembly membership information to or from nodes. All methods are implemented by the system (Max). Client code can query an INode for this interface:

```
INode* n;  
IAssembly* a = GetAssemblyInterface(n);
```

### **Methods:**

Notes for Set methods:

Nodes can be both assembly members and heads at the same time.

### **Prototype**

```
virtual void SetAssemblyMember(BOOL b) = 0;
```

### **Remarks:**

Method for setting state of assembly member. To close an assembly member call SetAssemblyMemberOpen(FALSE), as documented below.

### **Parameters:**

**BOOL b**

Specifies a new state for an assembly member.

If TRUE the node is set as an assembly member.

If FALSE removes a closed or open assembly member from membership. An open member will be closed first and then have it's membership flag removed.

**Prototype**

**virtual void SetAssemblyMemberOpen(BOOL b) = 0;**

**Remarks:**

Method for opening or closing an assembly member. It should only be called on members.

**Parameters:**

**BOOL b**

Specifies the state of the assembly member.

If TRUE the assembly member is opened.

If FALSE the assembly member is closed.

**Prototype:**

**virtual void SetAssemblyHead(BOOL b) = 0;**

**Remarks:**

Method to designate an assembly member as the assembly head.

**Parameters:**

**BOOL b**

Specifies the head state of the member.

If TRUE the node is set as the assembly head.

If FALSE an open or closed assembly head becomes a non-head. If the head is open, it is first closed and then the head flag is removed. To close an assembly head call `SetAssemblyHeadOpen(FALSE)`, as documented below.

**Prototype:**

**virtual void SetAssemblyHeadOpen(BOOL b) = 0;**

**Remarks:**

Method for opening or closing an assembly head. It should only be called on an assembly head.

**Parameters:****BOOL b**

Specifies the state of the assembly head.

If TRUE the assembly head is opened.

If FALSE the assembly head is closed.

Notes for Query methods:

To detect closed assembly members or assembly heads, check both the member/head flag and the open member/head flag i.e. `IsAssemblyHead() && !IsAssemblyMemberOpen()`.

**Prototype:**

**virtual BOOL IsAssemblyMember() const = 0;**

**Remarks:**

Method to determine membership in an assembly. It will work with either open or closed members.

**Return Value:**

If TRUE, node is a member the assembly.

If FALSE, node is not a member of the assembly.

**Prototype:**

**virtual BOOL IsAssemblyHead() const = 0;**

**Remarks:**

Method to determine if a node is an assembly head. It works with either open or closed heads.

**Return Value:**

If TRUE, node is an assembly head.

If FALSE, node is not an assembly head.

**Prototype:**

**virtual BOOL IsAssemblyMemberOpen() const = 0;**

**Remarks:**

Method to determine if an assembly member is open.

**Return Value:**

If TRUE, the assembly member is open.

If FALSE, the assembly member is not open.

**Prototype:**

**virtual BOOL IsAssemblyHeadOpen() const = 0;**

**Remarks:**

Method to determine if an assembly head is open.

**Return Value:**

If TRUE, the assembly head is open.

If FALSE, the assembly head is not open.

**Prototype:**

```
virtual BOOL IsAssemblyHeadMemberOf(const IAssembly*  
const assemblyHead) const = 0;
```

**Remarks:**

Method to detect assemblies within assemblies. It checks whether this assembly node is a head node and is also a member of the assembly headed by the node passed in as a parameter.

**Return Value:**

If TRUE, node is both a head node and is a member of another assembly.

If FALSE, node is neither a head node nor a member of another assembly.

**Prototype:**

```
virtual IOResult Save(ISave* isave) = 0;
```

**Remarks:**

| for implementing persistence of the underlying object.

**Parameters:**

**ISave\* isave**

Pointer for write methods.

**Return Value:**

**IO\_OK**, the call succeeded.

**IO\_ERROR**, the call was unsuccessful.

**Prototype:**

**virtual IOResult Load(ILoad\* iload) = 0;**

**Remarks:**

Read method for implementing persistence of the underlying object.

**Parameters:**

**ILoad\* iload**

Pointer for read methods.

**Return Value:**

**IO\_OK**, the call succeeded.

**IO\_ERROR**, the call was unsuccessful.

## **Class IAssembly2 : public IAssembly**

### **Description:**

This new version of the assembly interface extends IAssembly. Developers are encouraged to use this version of the assembly interface.

Client code can query an INode for this interface:

```
INode* n;
```

```
IAssembly2* a = GetAssemblyInterface2(n);
```

### **Methods:**

Notes:

These methods should be called on assembly heads only. Calling them on members will not affect the display of the bounding box. The bounding box is displayed in red (by default) around an open assembly. Turning it off can reduce viewport clutter; it won't affect the functionality of the assembly (the way the assembly works).

Calling these methods on an assembly head, affects the display of the bounding box only on that assembly.

**Prototype:**

```
virtual void SetAssemblyBBoxDisplay(BOOL b) = 0;
```

### **Remarks:**

Implemented by the System. Method to control the display of an assembly's world space bounding box.

### **Parameters:**

**BOOL b**

If TRUE, display the bounding box.

If FALSE, do not display the bounding box.

**Prototype:**

**virtual BOOL GetAssemblyBBoxDisplay() = 0;**

**Remarks:**

Implemented by the system. Method to retrieve the value of the bounding box display flag.

**Return Value:**

**BOOL**

If TRUE, the bounding box display is enabled.

If FALSE, the bounding box display is disabled.

## Class IEditNormalsMod

See Also: [Class MNMESH](#) , [Class MNNormalSpec](#)

```
class IEditNormalsMod : public FPMixinInterface
```

### Description:

This class is available in release 5.0 and later only.

This class is an interface used by the scripiter and the SDK to access the Edit Normals modifier. See the documentation for that modifier for background on the normals and the basic operations like Break and Unify.

All but the last two of these methods are available via the scripiter with commands like:

```
numNormals =  
$.modifiers[#Edit_Normals].EditNormalsMod.GetNumNormals ()  
$.modifiers[#Edit_Normals].EditNormalsMod.SetSelection #  
{1..numNormals}  
$.modifiers[#Edit_Normals].EditNormalsMod.Unify ().
```

### Methods:

.

### Prototype:

```
int EnfnGetSelLevel ();
```

**Remarks:**

"Get" accessor for selection level - one of these values:  
EN\_SL\_OBJECT, EN\_SL\_NORMAL, EN\_SL\_VERTEX,  
EN\_SL\_EDGE, EN\_SL\_FACE

.

**Prototype:**

**int EnfnSetSelLevel (int selLevel);**

**Remarks:**

"Set" accessor for selection level - one of these values:  
EN\_SL\_OBJECT, EN\_SL\_NORMAL, EN\_SL\_VERTEX,  
EN\_SL\_EDGE, EN\_SL\_FACE

.

**Prototype:**

**bool EnfnMove (Point3& offset, TimeValue t);**

**Remarks:**

Moves the ends of currently selected normals by the offset indicated  
- then renormalizes them to unit length.

Note that the time is currently ignored, as the Edit Normals modifier  
is not yet animatable

**Prototype:**

**bool EnfnRotate (Quat & rotation, TimeValue t);**

**Remarks:**

Rotates currently selected normals by the rotation indicated.

Note that the time is currently ignored, as the Edit Normals modifier is not yet animatable

**Prototype:**

**bool EnfnBreakNormals (BitArray \*normalSelection=NULL, INode \*pNode=NULL)**

**Remarks:**

Breaks the indicated normals into separate normals for each face.

**Parameters:**

**BitArray \*normalSelection=NULL**

An optional selection set to use. If NULL, the current selection is used.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**bool EnfnUnifyNormals (BitArray \*normalSelection=NULL, INode \*pNode=NULL)**

**Remarks:**

Unifies the indicated normals so there's at most one normal per vertex. (Basically causes normals to be shared across faces at a vertex.)

**Parameters:**

**BitArray \*normalSelection=NULL**

An optional selection set to use. If NULL, the current selection is used.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**bool EnfnResetNormals (BitArray \*normalSelection=NULL, INode \*pNode=NULL)**

**Remarks:**

Makes the indicated normals completely non-explicit and unspecified. Generates a rebuild & computation to determine the topology and direction of the newly unspecified normals.

**Parameters:**

**BitArray \*normalSelection=NULL**

An optional selection set to use. If NULL, the current selection is used.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**bool EnfnSpecifyNormals (BitArray \*normalSelection=NULL, INode \*pNode=NULL)**

**Remarks:**

Specifies the normals indicated to be fixed to the faces they're currently used by.

**Parameters:**

**BitArray \*normalSelection=NULL**

An optional selection set to use. If NULL, the current selection is used.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**bool EnfnMakeNormalsExplicit (BitArray  
\*normalSelection=NULL, INode \*pNode=NULL)**

**Remarks:**

Make the indicated normals explicit, so they won't be based on underlying face normals.

**Parameters:**

**BitArray \*normalSelection=NULL**

An optional selection set to use. If NULL, the current selection is used.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**bool EnfnCopyNormal (int normalID, INode \*pNode=NULL)**

**Remarks:**

Copies the indicated normal into the Edit Normals modifier's copy/paste buffer.

**Parameters:**

**int normalID**

The ID of the normal we want to copy.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**bool EnfnPasteNormal (BitArray \*normalSelection=NULL, INode \*pNode=NULL)**

**Remarks:**

Pastes the normal currently in the Edit Normals modifier's copy/paste buffer into the normals indicated, making them specified and explicit.

**Parameters:**

**BitArray \*normalSelection = NULL**

An optional selection set to use. If NULL, the current selection is used.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**BitArray \*EnfnGetSelection (INode \*pNode=NULL)**

**Remarks:**

Returns a pointer to the current selection.

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**bool EnfnSetSelection (BitArray & selection, INode \*pNode=NULL)**

**Remarks:**

Sets the normal selection to the selection given.

**Parameters:**

**BitArray & selection**

The desired selection

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this

parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Return Value:**

True if the selection was changed; false if the new selection was the same as the old selection.

**Prototype:**

**bool EnfnSelect (BitArray & selection, bool invert=false, bool select=true, INode \*pNode=NULL)**

**Remarks:**

Selects, deselects, or inverts the selection of the normals indicated.

**Parameters:**

**BitArray & selection**

The normals whose selection we are trying to change

**bool invert=false**

If true, indicates that the normals in <selection> should have their selection status inverted

**bool select=true**

If <invert> is true, this is ignored. Otherwise, if true, the normals indicated should be selected; if false, they should be deselected.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Return Value:**

True if the selection changed, false otherwise.

**Prototype:**

**void EnfnConvertVertexSelection (BitArray & vertexSelection, BitArray & normalSelection, INode \*pNode=NULL)**

**Remarks:**

Converts a vertex selection into a selection of normals, by setting bits on normals based at selected faces.

**Parameters:**

**BitArray & vertexSelection**

The vertex selection we're converting

**BitArray & normalSelection**

The output normal selection

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this

parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Return Value:**

True if the selection changed, false otherwise.

**Prototype:**

**void EnfnConvertEdgeSelection (BitArray & edgeSelection, BitArray & normalSelection, INode \*pNode=NULL)**

**Remarks:**

Converts an edge selection into a selection of normals, by setting bits for normals used on either end and either side of selected edges.

**Parameters:**

**BitArray & edgeSelection**

The edge selection we're converting

**BitArray & normalSelection**

The output normal selection

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate

call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**void EnfnConvertFaceSelection (BitArray & faceSelection, BitArray & normalSelection, INode \*pNode=NULL)**

**Remarks:**

Converts a face selection into a selection of normals, by setting bits on normals used by selected faces.

**Parameters:**

**BitArray & faceSelection**

The face selection we're converting

**BitArray & normalSelection**

The output normal selection

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**int EnfnGetNumNormals (INode \*pNode=NULL)**

**Remarks:**

Returns the current number of normals.

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**Point3 \*EnfnGetNormal (int normalID, INode \*pNode=NULL, TimeValue t=0)**

**Remarks:**

Returns a pointer to the normal indicated.

**Parameters:**

**int normalID**

The index of the normal

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**TimeValue t=0**

This is currently unused - but might be important if we add animation capabilities to Edit Normals in the future.

**Prototype:**

```
void EnfnSetNormal (int normalID, Point3 &direction, INode *pNode=NULL, TimeValue t=0)
```

**Remarks:**

Sets the indicated normal to a specific value. NOTE that this does not set the "explicitness" of this normal. If the normal is not made explicit, it will be restored to its default value the next time non explicit normals are recomputed.

**Parameters:****int normalID**

The index of the normal

**Point3 & direction**

The desired normal direction. If not already normalized to a length of 1, this method will normalize it.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**TimeValue t=0**

This is currently unused - but might be important if we add animation capabilities to Edit Normals in the future.

**Prototype:**

**bool EnfnGetNormalExplicit (int normalID, INode \*pNode=NULL)**

**Remarks:**

Controls whether a given normal is built from smoothing groups or set to an explicit value

(Also makes the normal specified for all faces using this normal.)

**Parameters:**

**int normalID**

The index of the normal

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**void EnfnSetNormalExplicit (int normID, bool value, INode \*pNode=NULL)**

**Remarks:**

Makes the indicated normal explicit (or not). If setting the normal to explicit, it will also be set to "specified" on all faces using it. If setting it to non-explicit, the modifier recomputes all non-explicit normals to bring it up to date.

**Parameters:**

**int normalID**

The index of the normal

**bool value**

True to make this normal explicit, false to make it non-explicit.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**int EnfnGetNumFaces (INode \*pNode=NULL)**

**Remarks:**

Returns the number of faces in the normal specification..

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**int EnfnGetFaceDegree (int face, INode \*pNode=NULL)**

**Remarks:**

Returns the degree of the face indicated. (3 for triangle, 4 for quad, etc.)

**Parameters:****int face**

The desired face.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:****int EnfnGetNormalID (int face, int corner, INode \*pNode=NULL)****Remarks:**

Gets the index of the normal in the indicated corner of the indicated face

**Parameters:****int face**

The desired face.

**int corner**

The desired corner, in the range of 0 to `EnfnGetFaceDegree(face)-1`.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**void EnfnSetNormalID (int face, int corner, int normalID, INode \*pNode=NULL)**

**Remarks:**

Sets the index of the normal in the indicated corner of the indicated face

**Parameters:**

**int face**

The desired face.

**int corner**

The desired corner, in the range of 0 to `EnfnGetFaceDegree(face)-1`.

**int normalID**

The index of the desired normal

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**bool EnfnGetFaceNormalSpecified (int face, int corner, INode \*pNode=NULL)**

**Remarks:**

Indicates whether a particular corner of a particular face is specified to use a specific normal or not.

**Parameters:**

**int face**

The desired face.

**int corner**

The desired corner, in the range of 0 to EnfnGetFaceDegree(face)-1.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in

the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

```
void EnfnSetFaceNormalSpecified (int face, int corner, bool specified, INode *pNode=NULL)
```

**Remarks:**

Controls whether a corner of a face uses a specific normal ID, or builds normals based on smoothing groups. If called to set a corner to unspecified, it generates a rebuild of nonspecified normals and a recomputation of nonexplicit normals at next update.

**Parameters:**

**int face**

The desired face.

**int corner**

The desired corner, in the range of 0 to EnfnGetFaceDegree(face)-1.

**bool specified**

True to specify, false to set as unspecified.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**int EnfnGetNumVertices (INode \*pNode=NULL)**

**Remarks:**

Returns the number of vertices in the current mesh cache..

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**int EnfnGetVertexID (int face, int corner, INode \*pNode=NULL)**

**Remarks:**

Returns the vertex used in a corner of a face, in the current mesh cache. (Useful for determining the "base" of the normal used in that corner of that face.)

**Parameters:****int face**

The desired face.

**int corner**

The desired corner, in the range of 0 to `EnfnGetFaceDegree(face)-1`.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**Point3 EnfnGetVertex (int vertexID, INode \*pNode=NULL, TimeValue t=0)**

**Remarks:**

Returns the location of the vertex indicated (in the current mesh cache).

**Parameters:**

**int vertexID**

The desired vertex.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**TimeValue t=0**

This is currently unused - but might be important if we add animation capabilities to Edit Normals in the future.

**Prototype:**

**int EnfnGetNumEdges (INode \*pNode=NULL)**

**Remarks:**

Returns the number of edges in the current mesh cache.

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**int EnfnGetEdgeID (int faceIndex, int sideIndex, INode \*pNode=NULL)**

**Remarks:**

Returns the index of the edge used on a particular side of a particular face, in the current mesh cache

**Parameters:**

**int faceIndex**

The desired face.

**int side**

The desired side, in the range of 0 to EnfnGetFaceDegree(faceIndex)-1.

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate

call

for each node. If NULL, it assumes you want to modify normals in the "first" node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**int EnfnGetFaceEdgeSide (int faceIndex, int edgeIndex, INode \*pNode=NULL)**

**Remarks:**

Tells you which side of the face a given edge is on. (Can be useful for getting normal and vertex information around the edge.)

**Parameters:**

**int faceIndex**

The desired face.

**int edgeIndex**

The desired edge

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Return Value:**

The side of the face, in the range of 0 to `EnfnGetFaceDegree(faceIndex)-1`, or -1 if the edge was not found on this face

**Prototype:**

```
int EnfnGetEdgeVertex (int edgeIndex, int end, INode *pNode=NULL)
```

**Remarks:**

Returns the vertex at the end of the edge.

**Parameters:**

**int edgeIndex**

The desired edge.

**int end**

The desired end - either 0 or 1

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

```
int EnfnGetEdgeFace (int edgeIndex, int side, INode  
*pNode=NULL) { return 0; }
```

**Remarks:**

Tells you what face is on a particular side of a particular edge.

**Parameters:**

**int edgeIndex**

The index of the edge in the MNMesh's edge array.

**int side**

Indicates which side of the edge you want the face from. (Values: 0 or 1.)

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Return Value:**

The index of the desired face, or -1 if there's an error or if there is no face on that side.

**Prototype:**

**int EnfnGetEdgeNormal (int edgeIndex, int end, int side, INode \*pNode=NULL)**

**Remarks:**

Returns the normal associated with a particular end and side of this edge.

**Parameters:**

**int edgeIndex**

The index of the edge in the MNMesh's edge array.

**int end**

Indicates which end of the edge should be used. (Values: 0 or 1.)

**int side**

Indicates which side of the edge should be used - the edge may have different normals used by the faces on either side. (Values: 0 or 1.)

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Return Value:**

The index of the desired normal..

**Prototype:**

```
void EnfnRebuildNormals (INode *pNode=NULL) { }
```

**Remarks:**

Forces the modifier to rebuild all non-specified normals from the face smoothing groups. Note that this can change the number of normals in some cases, and often changes their order

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

```
void EnfnRecomputeNormals (INode *pNode=NULL) { }
```

**Remarks:**

Forces the modifier to recompute all non-explicit normals.

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**Prototype:**

**MNMesh \*EnfnGetMesh (INode \*pNode=NULL, TimeValue t=0)**

**Remarks:**

Returns a pointer to the cached copy of the MNMesh held by the EditNormalsModData. This is a "stripped-down" copy of the last mesh that was output by the modifier. It contains no maps, vertex or edge data, or normals. It's mainly used as a temporary "parent" to the localdata's MNNormalSpec in operations such as Display.

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this

parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

**TimeValue t=0**

This is currently unused - but might be important if we add animation capabilities to Edit Normals in the future.

**Prototype:**

**MNNormalSpec \*EnfnGetNormals (INode \*pNode=NULL,  
TimeValue t=0)**

**Remarks:**

Returns a pointer to the MNNormalSpec used by the EditNormalsModData.

This MNNormalSpec is not part of any particular MNMesh, rather it's used as the local data of the EditNormalsMod to indicate what normals should be applied to the mesh coming up the pipe. Its "parent" pointer is generally set to NULL, and should be set to a mesh like the one you get from EnfnGetMesh before you do certain operations. (See class MNNormalSpec for details on which methods require an accurate "parent" pointer.

**Parameters:**

**INode \*pNode = NULL**

If the Edit Normals modifier is instanced across several nodes, this parameter can be used to indicate which node you mean to modify normals in.

If you want to modify normals in all nodes, you must make a separate call

for each node. If NULL, it assumes you want to modify normals in the "first"

node. (This is fine if the modifier is only applied to one node, as is usually the case.)

### **TimeValue t=0**

This is currently unused - but might be important if we add animation capabilities to Edit Normals in the future.

## External Icons

See Also: [Class MaxIcon](#), [Class ICustButton](#)

Starting with 3ds max version 4.0 most of the icons used in the software have been externalized, meaning that these alpha-composited background icons now reside as \*.bmp files in the \UIIcons folder and are loaded during startup. Some of the image processing that takes place on these icons will be outlined below.

Resource based icons can be converted effectively to external icons and stored as 24-bit image files with associated 24-bit alpha mask image files. The files follow a fairly standard naming convention so for the following outline the material editor icons and image files will be used as an example. These can be found in the \UIIcons folder as;

**MeditImages\_a.bmp** The 24-bit icon alpha mask

**MeditImages\_i.bmp** The 24-bit icon images

One difference with ordinary icon formats is that these use an alpha mask instead of an XOR mask. An XOR mask needs to be inverted in order to create a proper alpha mask out of it. The process for this is fairly straightforward; copy the images directly to a file (i.e. myicons\_i.bmp), invert the mask file and storing that in the associated file (i.e. myicons\_a.bmp). This process will result in icons which are identical to the XOR masked versions. Of course, you can also tweak the alpha mask to give it fuzzy edges and other effects you might like for your icons to help composite with the background color in a seamless and smooth way.

In the plugin code you will need to create image lists for the icons. Once again, taking the material editor as an example, you can create the image list in the following manner;

```
hMeditImages = ImageList_Create(BUTW, BUTH, ILC_COLOR24 |  
ILC_MASK, 5, 0);  
LoadMAXFileIcon("MeditImages", hMeditImages, kBackground,
```

**FALSE);**

The first thing we do is create an image list with 24-bit images and a mask. Secondly, the call to **LoadMAXFileIcon()** loads the images into the image list. See below for more information on this function;

If you want your icons to respond when the user customizes colors you will need to implement a color change callback to reload the icons. This is not required, but it does make your UI play nicely with the color customization system. You need to create a callback as outlined below;

```
static void ColorChangeNotifyProc(void* param, NotifyInfo* pInfo)
{
    ImageList_RemoveAll(hMeditImages);
    LoadMAXFileIcon("MeditImages", hMeditImages, kBackground,
FALSE);
}
```

Then you will need to register it after you load the icons for the first time;

```
RegisterNotification(ColorChangeNotifyProc, NULL,
NOTIFY_COLOR_CHANGE);
```

This way your buttons will always look correct when the user starts customizing colors.

**Function:**

```
BOOL LoadMAXFileIcon(TCHAR* pFile, HIMAGELIST
hImageList, ColorId color, BOOL disabled);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method will load the images into the image list.

**Parameters:**

**TCHAR\* pFile**

The prefix of the file name for the icon BMP files. The path is not included. The system will look in the UIIcons directory, and append "\_i.bmp" for the image file and "\_a.bmp" for the alpha mask file.

**HIMAGELIST hImageList**

The image list that the icons will be loaded into. It appends the images at the end of the list.

**ColorId color**

The id of the background color used to composite. All the available "ColorId" values are defined in **MAXSDK\INCLUDE\iColorMan.h**. Normally you will use "kBackground" which is the color id for the background color for all MAX controls.

**BOOL disabled**

A boolean that tells the system whether to produce the disabled or enabled versions of the icons. This controls what sort of image processing is done on the icons. We store a set of scale values for the saturation, value and transparency for both enabled and disabled icons. This flag determines which set of values we use when loading the icons. If your original image list contains both enabled and disabled versions of its icons, then you can just pass "FALSE" for this parameter and use the image list as you did before. If you want the system to automatically create the disabled versions of your icons, then you can call this with "TRUE".

**Return Value:**

TRUE if successful, otherwise FALSE.

# RPF Files and the G-Buffer

## Overview

The RPF format is the 3ds max Rich Pixel file Format that supports the ability to include arbitrary image channels.

RPF files replace RLA files as the format of choice for rendering animations requiring further post-production or effects work. Many channels available in RPF files are exclusive to this format. The following is a list of pointers (e.g. a required reading list) to information in the SDK documentation pertaining to the storage type and format used for the RPF format.

### [Class GBuffer](#)

The G-Buffer stores multiple layers at each pixel.

### [List of Image \(G-Buffer\) Channels](#)

This section provides an overview of the image channels available with the GBuffer which relate directly to the multiple layer pixel information stored in the RPF files.

### [List of G-Buffer Channels Indices](#)

This section provides a quick overview of the various data size and channel names relating to the G-Buffer and layered pixel storage in RPF files.

### [List of G-Buffer Channel Types](#)

This section provides the list of channel types and the size associated with these channels per pixel.

### [Structure GBufData](#)

This structure is used by the Class GBufReader and Class GBufWriter.

### [Structure RealPixel](#)

This structure represents an **rgbe** based pixel where **e** is the base 2 exponent of the maximum RGB component while **rgb** are the mantissas of RG and B relative to **e**. This will compress the essential data of a floating point based

color into 32 bits.

The actual RPF file source code can be found in the SDK samples directory under

**\MAXSDK\SAMPLES\IO\RLA\RLA.CPP**

The files still refer to RLA instead of RPF for legacy purposes.

## Class FilterList

class FilterList

### Description:

A class whose sole purpose is for building up a filter list to pass to the Windows API functions **GetSaveFileName()** and **GetOpenFileName()**. It automatically puts in the embedded nulls and two terminating nulls. All methods are implemented by the system.

Example usage:

```
FilterList filterList;  
filterList.Append(_T("MAX files(*.max)"));  
filterList.Append(_T("*.max"));  
ofn.lpstrFilter = filterList;  
GetSaveFileName(&ofn)
```

### Data Members:

```
#define LISTBUFLLEN 2048
```

public:

```
TCHAR buf[LISTBUFLLEN];  
int length;
```

### Methods:

#### Prototype:

```
FilterList();
```

#### Remarks:

Constructor. Sets **buf** to all zeros and sets the **length** to 0.

#### Return Value:

A new FilterList object.

#### Prototype:

```
void Append(TCHAR *name);
```

#### Remarks:

Appends the string passed to **buf**.

**Prototype:**

**operator TCHAR \*()**

**Remarks:**

Returns the address of **buf**.

## Class Expr

See Also: [Class Point3](#), [List of Expression Types](#), [List of Expression Variable Types](#), [List of Expression Return Codes](#), [Character Strings](#).

class Expr

### Description:

This class may be used by developers to parse mathematical expressions. The expression is created as a character string using a straightforward syntax. Expressions consist of operators (+, -, \*, /, etc.), literal constants (numbers like 180, 2.718, etc.), variables (single floating point values or vector (**Point3**) values), and functions (mathematical functions that take one or more arguments and return a result). The return value from the expression may be a floating point value or a vector. There are many built in functions, operators and constants available for use.

All methods of this class are implemented by the system.

Developers wishing to use these APIs should #include **\MAXSDK\INCLUDE\EXPRLIB.H** and should link to **\MAXSDK\LIB\EXPR.LIB**.

Sample code using these APIs is shown below, and is also available as part of the expression controller in **\MAXSDK\SAMPLES\CONTROLLERS\EXPRCTRL.CPP**.

Variables may be defined and used in expressions. Variable names are case sensitive, and must begin with a letter of the alphabet, but may include numbers. They may be any length. To create a named variable, you use the method **defVar()**. This takes a name and returns a register number. Defining the variable creates storage space in a list of variables maintained by the parser, and the register number is used as an array index into the variable value arrays passed into the expression evaluation method (**eval()**).

To use the variable in an expression just use its name. For example if you define a variable named **radius**, you can use it in an expression like: **2\*pi\*radius**. To give the variable a value, you define two arrays of variables and pass them to the evaluation method (**eval()**). There is one array for scalar variables, and one for vector variables. You pass these arrays along with the number of variables in each list. See the sample code below for an example.

The order of calling the methods of this class to evaluate an expression is as

follows:

Declare an expression instance (**Expr expr;**)

Define the expression (**char e1[] = "2\*pi\*radius";**).

Define any variables (**expr.defVar(SCALAR\_VAR, \_T("radius"));**)

Load the expression (**expr.load(e1);**)

Evaluate the expression (**expr.eval(...);**)

There are no restrictions on the use of white space in expressions -- it may be used freely to make expressions more readable. In certain instances, white space should be used to ensure non-ambiguous parsing. For example, the **x** operator is used for to compute the cross product of two vectors. If a developer has several vectors: **Vec**, **Axis** and **xAxis** and wanted to compute the cross product, **VecxAxis** is ambiguous while **Vec x Axis** is not.

All the necessary information to evaluate an expression is completely stored within an expression object. For example, if you are passed a pointer to an expression object for which some variables have been defined that you knew the value of, you could get all the information you needed from the expression object to completely evaluate the expression. This includes the expression string, variable names, variable types, and variable register indices.

For complete documentation of the built in functions please refer to the 3ds max User's Guide under Using Expression Controllers. Below is an overview of the operators, constants and functions that are available:

## **Expression Operators:**

### **Scalar Operators**

Operator Use Meaning

+ p+q addition

- p-q subtraction

- -p additive inverse

\* p\*q multiplication

/ p/q division

^ p^q power (p to the power of q)

\*\* p\*\*q same as p^q

## **Boolean Operators**

= p=q equal to

< p<q less than

> p>q greater than

<= p<=q less than or equal to

>= p>=q greater than or equal to

| p|q logical OR

& p&q logical AND

## **Vector Operators**

+ V+W addition

- V-W subtraction

\* p\*V scalar multiplication

V\*p "

\* V\*W dot product

x VxW cross product

/ V/p scalar division

. V.x first component (X)

. V.y second component (Y)

. V.z third component (Z)

## **Built-In Constants:**

**pi** 3.1415...

**e** 2.7182...

**TPS** 4800 (ticks per second)

## **Expression Functions:**

### **Trigonometric Functions**

The angles are specified and returned in degrees.

**sin(p)** sine

**cos(p)** cosine

**tan(p)** tangent

**asin(p)** arc sine

**acos(p)** arc cosine

**atan(p)** arc tangent

### **Hyperbolic Functions**

**sinh(p)** hyperbolic sine

**cosh(p)** hyperbolic cosine

**tanh(p)** hyperbolic tangent

### **Conversion between Radians and Degrees**

**radToDeg(p)** takes p in radians and returns the same angle in degrees

**degToRad(p)** takes p in degrees and returns the same angle in radians

### **Rounding Functions**

**ceil(p)** smallest integer greater than or equal to p.

**floor(p)** largest integer less than or equal to p.

### **Standard Calculations**

**ln(p)** natural (base e) logarithm

**log(p)** common (base 10) logarithm

**exp(p)** exponential function --  $\exp(e) = e^p$

**pow(p, q)** p to the power of q --  $p^q$

**sqrt(p)** square root

**abs(p)** absolute value

**min(p, q)** minimum -- returns p or q depending on which is smaller

**max(p, q)** maximum -- returns p or q depending on which is larger

**mod(p, q)** remainder of p divided by q

### **Conditional**

**if (p, q, r)** works like the common spreadsheet "if" -- if p is nonzero then "if" returns q, otherwise r.

### **Vector Handling**

**length(V)** the length of V

**unit(V)** returns a unit vector in the same direction as V.

**comp(V, I)** i-th component, where I=0, 1, or 2.

`comp([5,6,7],1) = 6`

## Special Animation Functions

**noise(p, q, r)** 3D noise -- returns a randomly generated position.

p, q, and r are random values used as a seed.

### Sample Code:

The following code shows how the expression parser can be used. This code evaluates several expressions and displays the results in a dialog box. Both scalar and vector variables are used. One expression contains an error to show how error handling is done.

```
void Utility::TestExpr() {  
    // Declare an expression instance and variable storage  
    Expr expr;  
    float sRegs[2]; // Must be at least getVarCount(SCALAR_VAR);  
    Point3 vRegs[2]; // Must be at least getVarCount(VECTOR_VAR);  
    float ans[3];  
    int status;  
  
    // Define a few expressions  
    char e0[] = "2+2";  
    char e1[] = "2.0 * pi * radius";  
    char e2[] = "[1,1,0] + axis";  
    char e3[] = "[sin(90.0), sin(radToDeg(0.5*pi)), axis.z]";  
    char e4[] = "2+2*!@#$$%"; // Bad expression  
  
    // Define variables  
    int radiusReg = expr.defVar(SCALAR_VAR, _T("radius"));  
    int axisReg = expr.defVar(VECTOR_VAR, _T("axis"));  
    // Set the variable values  
    sRegs[radiusReg] = 50.0f;  
    vRegs[axisReg] = Point3(0.0f, 0.0f, 1.0f);  
    // Get the number of each we have defined so far  
    int sCount = expr.getVarCount(SCALAR_VAR);  
    int vCount = expr.getVarCount(VECTOR_VAR);  
  
    // Load and evaluate expression "e0"  
    if (status = expr.load(e0))  
        HandleLoadError(status, expr);
```

```

else {
    status = expr.eval(ans, sCount, sRegs, vCount, vRegs);
    if (status != EXPR_NORMAL)
        HandleEvalError(status, expr);
    else
        DisplayExprResult(expr, ans);
}
// Load and evaluate expression "e1"
if (status = expr.load(e1))
    HandleLoadError(status, expr);
else {
    status = expr.eval(ans, sCount, sRegs, vCount, vRegs);
    if (status != EXPR_NORMAL)
        HandleEvalError(status, expr);
    else
        DisplayExprResult(expr, ans);
}
// Load and evaluate expression "e2"
if (status = expr.load(e2))
    HandleLoadError(status, expr);
else {
    status = expr.eval(ans, sCount, sRegs, vCount, vRegs);
    if (status != EXPR_NORMAL)
        HandleEvalError(status, expr);
    else
        DisplayExprResult(expr, ans);
}
// Load and evaluate expression "e3"
if (status = expr.load(e3))
    HandleLoadError(status, expr);
else {
    status = expr.eval(ans, sCount, sRegs, vCount, vRegs);
    if (status != EXPR_NORMAL)
        HandleEvalError(status, expr);
    else
        DisplayExprResult(expr, ans);
}

```

```

// Load and evaluate expression "e4"
if (status = expr.load(e4))
    HandleLoadError(status, expr);
else {
    status = expr.eval(ans, sCount, sRegs, vCount, vRegs);
    if (status != EXPR_NORMAL)
        HandleEvalError(status, expr);
    else
        DisplayExprResult(expr, ans);
}
}

// Display the expression and the result
void Utility::DisplayExprResult(Expr expr, float *ans) {
    TCHAR msg[128];

    if (expr.getExprType() == SCALAR_EXPR) {
        _stprintf(msg, _T("Answer to \"%s\" is %.1f"),
            expr.getExprStr(), *ans);
        Message(msg, _T("Expression Result"));
    }
    else {
        _stprintf(msg, _T("Answer to \"%s\" is [%%.1f, %%.1f, %%.1f]"),
            expr.getExprStr(), ans[0], ans[1], ans[2]);
        Message(msg, _T("Expression Result"));
    }
}

// Display the load error message
void Utility::HandleLoadError(int status, Expr expr) {
    TCHAR msg[128];

    if(status == EXPR_INST_OVERFLOW) {
        _stprintf(_T("Inst stack overflow: %s"),
            expr.getProgressStr());
        Message(msg, _T("Error"));
    }
}

```

```

else if (status == EXPR_UNKNOWN_TOKEN) {
    _sprintf(msg, _T("Unknown token: %s"),
        expr.getProgressStr());
    Message(msg, _T("Error"));
}
else {
    _sprintf(msg,
        _T("Cannot parse \"%s\". Error begins at last char of: %s"),
        expr.getExprStr(), expr.getProgressStr());
    Message(msg, _T("Error"));
}
}

```

```

// Display the evaluation error message
void Utility::HandleEvalError(int status, Expr expr) {
    TCHAR msg[128];

```

```

    _sprintf(msg, _T("Can't parse expression \"%s\""),
        expr.getExprStr());
    Message(msg, _T("Error"));
}

```

```

// Display the specified message and title in a dialog box
void Utility::Message(TCHAR *msg, TCHAR *title) {
    MessageBox(ip->GetMAXHwnd(),
        (LPCTSTR) msg, (LPCTSTR) title,
        MB_ICONINFORMATION|MB_OK);
}

```

**Methods:**

**Prototype:**

**Expr()**

**Remarks:**

Constructor. Internal data structures are initialized as empty.

**Prototype:**

**~Expr()**

**Remarks:**

Destructor. Any currently defined variables are deleted.

**Prototype:**

**int load(char \*s);**

**Remarks:**

This method is used to load an expression for parsing. An error code is returned indicating if the expression was loaded. A successfully loaded expression is then ready for evaluation with the **eval()** method.

**Parameters:**

**char \*s**

The expression to load.

**Return Value:**

See [List of Expression Return Codes](#).

**Prototype:**

**int eval(float \*ans, int sRegCt, float \*sRegs, int vRegCt=0, Point3 \*vRegs=NULL);**

**Remarks:**

This method is used to evaluate the expression loaded using **load()**. It returns either a scalar or vector result.

**Parameters:**

**float \*ans**

The numeric result of the expression is returned here, i.e. the answer . For scalar values this is a pointer to a single float. For vector values, **ans[0] is x, ans[1] = y, ans[2] = z**. You can determine which type of result is returned using the method **getExprType()**.

**int sRegCt**

The number of items in the **sRegs** array of scalar variables.

**float \*sRegs**

Array of scalar variables.

**int vRegCt=0**

The number of items in the **vRegs** array of vector variables.

**Point3 \*vRegs=NULL**

Array of vector variables.

**Return Value:**

See [List of Expression Return Codes](#).

**Prototype:**

**int getExprType();**

**Remarks:**

Returns the type of expression. See [List of Expression Types](#).

**Prototype:**

**TCHAR \*getExprStr();**

**Remarks:**

Returns a pointer to the currently loaded expression string.

**Prototype:**

**TCHAR \*getProgressStr();**

**Remarks:**

If there was an error parsing the expression, this method returns a string showing what portion of the expression was parsed before the error occurred.

**Prototype:**

**int defVar(int type, TCHAR \*name);**

**Remarks:**

Defines a named variable that may be used in an expression.

**Parameters:**

**int type**

The type of variable. See [List of Expression Variable Types](#).

**TCHAR \*name**

The name of the variable. This name must begin with a letter, may include numbers and may be any length.

**Return Value:**

The register number (into the **sRegs** or **vRegs** array passed to **eval()**) of the

variable.

**Prototype:**

**int getVarCount(int type);**

**Remarks:**

This method returns the number of variables defined of the specified type. When you call **eval()** on an expression, you must make sure that the variable arrays (**sRegs** and **vRegs**) are at least the size returned from this method.

**Parameters:**

**int type**

See [List of Expression Variable Types](#).

**Prototype:**

**TCHAR \*getVarName(int type, int i);**

**Remarks:**

Returns the name of the variable whose index is passed, or NULL if the variable could not be found.

**Parameters:**

**int type**

The type the variable. See [List of Expression Variable Types](#).

**int i**

The register number of the variable.

**Prototype:**

**int getVarRegNum(int type, int i);**

**Remarks:**

When you define a variable with **defVar()**, you get a back a register number. If your code is set up in such a way that saving that register number is not convenient in the block of code that defines it, you can use this method later on to find out what that return value had been. For example, one piece of code might have:

```
expr->defVar(SCALAR_VAR, "a"); // not saving return  
value...
```

```
expr->defVar(SCALAR_VAR, "b");
```

and then right before evaluating the expression, you might have some code such as:

```
for(i = 0; i < expr->getVarCount(SCALAR_VAR); i++)  
    if(_tcscmp("a", expr->getVarName(SCALAR_VAR, i) == 0)  
        aRegNum = expr->getVarRegNum(SCALAR_VAR, i);
```

Of course, this is a bit contrived -- most real examples would probably have tables to store the variable names, register numbers, etc. and thus would not need to call this method. It is available however, and this makes the expression object self-contained in that everything you need to evaluate an expression with variables (other than the variable values themselves) is stored by the expression object.

**Parameters:**

**int type**

See [List of Expression Variable Types](#).

**int i**

The variable index returned from the method **defVar()**.

**Return Value:**

The register index for the variable whose type and index are passed.

**Prototype:**

**BOOL deleteAllVars();**

**Remarks:**

Deletes all the variables from the list maintained by the expression.

**Return Value:**

TRUE if the variables were deleted; otherwise FALSE.

**Prototype:**

**BOOL deleteVar(TCHAR \*name);**

**Remarks:**

Deletes the variable whose name is passed from the list maintained by the expression. Register numbers never get reassigned, even if a variable gets deleted. For example, if you delete variables 0-9, and keep variable 10, you're going to need to pass in an array of size at least 11 to the **eval()** method, even though the first 10 slots are unused.

**Parameters:**

**TCHAR \*name**

The name of the variable to delete.

**Return Value:**

TRUE if the variable was deleted; otherwise FALSE (the name was not found).

## Class CommandModeChangedCallback

See Also: [Class Interface](#), [Class CommandMode](#).

class CommandModeChangedCallback

### Description:

This is the callback object for **Interface::RegisterCommandModeChangedCallback()**.

### Methods:

### Prototype:

```
virtual void ModeChanged(CommandMode *oldM,  
    CommandMode *newM)=0;
```

### Remarks:

This method is called when the user changes command modes.

### Parameters:

**CommandMode \*oldM**

The command mode that was replaced.

**CommandMode \*newM**

The new command mode.

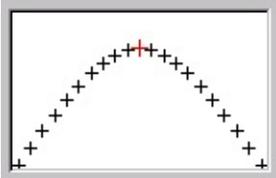
## Class TCBGraphParams

See Also: [Custom Controls](#).

class TCBGraphParams

### Description:

The TCB Graph control displays a tension/continuity/bias graph in the control.



If you are going to use the TCB Graph control you must initialize it by calling

```
void InitTCBGraph(HINSTANCE hInst);
```

from **DLLMain()**

The value to use in the Class field of the Custom Control Properties dialog is:

**TCBGraph**

Send this message to the graph control with **lParam** pointing to a

**TCBGraphParams** structure to set the graph parameters.

**WM\_SETTCBGRAPHPARAMS**

For example:

```
TCBGraphParams gp;
```

```
gp.tens = 0.0f; gp.bias = 0.0f; gp.cont = 0.0f;
```

```
gp.easeFrom = 0.0f; gp.easeTo = 0.0f;
```

```
HWND hGraph = GetDlgItem(hDlg, IDC_TCB_GRAPH);
```

```
EnableWindow(hGraph, TRUE);
```

```
SendMessage(hGraph, WM_SETTCBGRAPHPARAMS, 0,
```

```
(LPARAM)&gp);
```

```
UpdateWindow(hGraph);
```

Note that this control is not derived from **ICustControl** and thus does not have **Enable()**, **Disable()**, etc. methods.

### Data Members:

public:

```
float tens, cont, bias, easeFrom, easeTo;
```

The tension, continuity, bias, ease from and ease to parameters. Each value

may range from 0.0 to 1.0.

## Class AppDataChunk

See Also: [Class Animatable](#), [Class Class\\_ID](#).

```
class AppDataChunk
```

### Description:

This class represents an individual AppData chunk. All methods of this class are implemented by the system.

### Data Members:

public:

**Class\_ID classID;**

The Class\_ID of the owner of this chunk.

**SClass\_ID superClassID;**

The SuperClassID of the owner of this chunk.

**DWORD subID;**

An extra ID that lets the owner identify its sub chunks.

**DWORD length;**

The length of the data in bytes.

**void \*data;**

The chunk data itself.

### Methods:

### Prototype:

```
AppDataChunk(Class_ID cid, SClass_ID sid, DWORD sbid,  
DWORD len, void *d);
```

### Remarks:

Constructor. The data members are initialized to the values passed. Note that the **data** pointer should be allocated with standard **malloc()** since it will be freed using **free()** in the destructor.

### Parameters:

**Class\_ID cid**

The Class\_ID of the owner of this chunk.

**SClass\_ID sid**

The SuperClassID of the owner of this chunk.

**DWORD sbid**

An extra ID that lets the owner identify its sub chunks.

**DWORD len**

The length of the data in bytes.

**void \*d**

The chunk data itself.

**Prototype:**

**AppDataChunk();**

**Remarks:**

Constructor. The length is set to 0 and the data pointer set to NULL.

**Prototype:**

**~AppDataChunk()**

**Remarks:**

Destructor. The AppData is freed using **free(data)**. Since this is how the data is freed, the plug-in must use **malloc()** to allocate the memory.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

The system implements this method to load the AppDataChunk from disk.

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

The system implements this method to save the AppDataChunk to disk.

## Class ObjectDataReaderCallback

See Also: [Class TriObject](#).

class ObjectDataReaderCallback

### Description:

3D Studio DOS allowed developers to store APP\_DATA with objects and nodes in the scene. When the 3DS DOS file is imported into 3ds max, and no plug-in has registered to convert it, then it is just hung off the object (or INode in the case of KXP app data).

A 3ds max plug-in can register itself to read a particular APP\_DATA chunk when a 3DS DOS file is loaded. If a chunk is encountered that matches a registered plug-in, that plug-in will be asked to create an instance of itself based on the contents of the APP\_DATA chunk. The plug-in callback is given an opportunity to read the chunk and create an object other than a **TriObject** based on the contents of the chunk and the original object.

A plug-in that wants to process app data registers a new class derived from this class by calling **void**

**RegisterObjectAppDataReader(ObjectDataReaderCallback \*cb);**

The system then maintains a list of these **ObjectDataReaderCallbacks**.

### Methods:

### Prototype:

**virtual char \*DataName()=0;**

### Remarks:

Implemented by the Plug-In.

Returns the name that identifies the app data chunk. When the 3DS Import plug-in is loading objects, it will look for app data. For each app data chunk that the object has, it will go through the list of registered callbacks and call this method looking for a name match. When it does find a match it will call **ReadData()** on the callback that matched.

### Prototype:

**virtual Object \*ReadData(TriObject \*obj, void \*data, DWORD len)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to read the app data and create an instance of an object based on the data and the original mesh object. For example, the 3D Surfer plug-in from 3DS DOS creates a patch object based on the app data parameters and returns a pointer to it.

**Parameters:****TriObject \*obj**

The original mesh object the app data was assigned to. If no callback was registered, this would be the object that would get created.

**void \*data**

Points to the particular app data chunk handled by the registered callback.

**DWORD len**

The length of the app data chunk.

**Return Value:**

The Object created by the plug-in to hold the appdata. This would be the object created to take the place of the TriObject. For example, consider a 3DS object that had appdata embedded in it which represented the patch object from which the mesh was created (like 3D Surfer for example). This method would take that data and created a patch object so that the user could work with the object as a patch object in 3ds max instead of a tri object.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to delete this callback object. When the user exits 3ds max, this method is called on each of the registered callbacks. So if the callback was allocated dynamically, it could free itself in this implementation.

**The following functions are not part of any class but are available for use:**

**Prototype:**

**void RegisterObjectAppDataReader(ObjectDataReaderCallback \*cb);**

**Remarks:**

Implemented by the System.

This method allows a plug-in to register a callback that will be called when reading a 3DS file that has a particular appdata chunk. The plug-in callback is given an opportunity to read the chunk and create an object other than a TriObject based on the contents of the chunk and the original object.

**Parameters:**

**ObjectDataReaderCallback \*cb**

The callback to read the chunk and create an object.

**Prototype:**

**Object \*ObjectFromAppData(TriObject \*obj, char \*name, void \*data, DWORD len);**

**Remarks:**

This function is used internally by the 3DS Import plug-in.

Note the following about 3DS App Data. If app data is encountered and no plug-in has registered to convert it, then it is just hung off the object (or INode in the case of KXP app data). For object app data, TriObject's super class and class ID are used to identify the chunk and the sub ID is set to 0. For node app data, INode's super class and class ID are used to identify the chunk and the sub ID is set to 0.

This single 3ds max app data chunk will contain the entire 3DS app data chunk, which may have sub chunks (see the IPAS SDK). The following routines will aid in parsing 3DS app data.

**Prototype:**

**void GetIDStr(char \*chunk, char \*idstring);**

**Remarks:**

This function gets the ID string out of an **XDATA\_ENTRY** chunk and null terminates it.

**Parameters:**

**char \*chunk**

A pointer to the chunk.

**char \*idstring**

The ID string.

**Prototype:**

**int FindAppDataChunk(void \*appd, DWORD len, char \*idstring);**

**Remarks:**

Returns the offset into **appd** of the specified chunk or -1 if it is not found.

**Parameters:**

**void \*appd**

The is the entire app data chunk containing all the sub-chunks.

**DWORD len**

This is the length of the entire app data chunk.

**char \*idstring**

The ID of the chunk to find.

**Return Value:**

The offset into **appd** of the specified chunk or -1 if it is not found.

**Prototype:**

**void \*GetAppDataChunk(void \*appd, DWORD len, char \*idstring);**

**Remarks:**

This function is similar to **FindAppDataChunk()** above, but returns a pointer to the chunk or NULL if it is not found.

**Parameters:**

**void \*appd**

The is the entire app data chunk containing all the sub-chunks.

**DWORD len**

This is the length of the entire app data chunk.

**char \*idstring**

The ID of the chunk to find.

**Return Value:**

A pointer to the chunk or NULL if it is not found.

**Prototype:**

```
int SetAppDataChunk(void **pappd, DWORD &len, void *chunk);
```

**Remarks:**

This function adds the chunk to the appdata chunk, preserving existing chunks. **chunk** should point to the new chunk header followed by its data.

**Parameters:**

**void \*\*pappd**

The is the entire app data chunk containing all the sub-chunks.

**DWORD &len**

This is the length of the entire app data chunk.

**void \*chunk**

A pointer to the new chunk header.

**Return Value:**

Nonzero if the chunk was added; otherwise zero.

**Prototype:**

```
int DeleteAppDataChunk(void **pappd, DWORD &len, char *idstring);
```

**Remarks:**

Deletes a chunk from the appdata while preserving other chunks.

**Parameters:**

**void \*\*pappd**

The is the entire app data chunk containing all the sub-chunks.

**DWORD &len**

This is the length of the entire app data chunk.

**char \*idstring**

The ID of the chunk to delete.

**Return Value:**

Nonzero if the chunk was deleted; otherwise zero.

## List of Class IDs

See Also: [Class Class\\_ID](#), [List of Super Class IDs](#).

These are the Class\_IDs of the standard built-in classes. These IDs are defined in `\MAXSDK\INCLUDE\PLUGAPI.H`.

The following are global instances of several Class\_IDs defined by the system:

**triObjectClassID** - Triangle mesh Objects (TriObjects).

**defObjectClassID** - General deformable object. These are objects that provide points for modification.

**mapObjectClassID** - General texture-mappable object.

**patchObjectClassID** - Patch objects.

**genericShapeClassID** - Generic shapes.

**splineShapeClassID** - Spline shapes.

**linearShapeClassID** - Linear shapes.

**loftObjectClassID** - Loft object.

**derivObjClassID** - Object space derived objects.

**WSMDDerivObjClassID** - World space derived objects.

This Class\_ID is #defined for NURBS objects (**#define EDITABLE\_SURF\_CLASS\_ID Class\_ID(0x76a11646, 0x12a822fb)**):

**EDITABLE\_SURF\_CLASS\_ID** - NURBS objects.

Listed below are the first ULONG of the 8 byte ID. The second ULONG is 0 for all built-in classes (unless noted otherwise). For example a Class\_ID for a TriObject would read:

**Class\_ID(TRIOBJ\_CLASS\_ID, 0);**

Note that only **built-in** classes should have the second ULONG equal to 0. All plug-in developers should use both ULONGs.

The Class\_IDs are organized by their SuperClassIDs.

Subclass of all super classes

**STANDIN\_CLASS\_ID** - Stand-In

Subclasses of **REF\_MAKER\_CLASS\_ID**

**MTL\_LIB\_CLASS\_ID** - Mtl Library.

**MTLBASE\_LIB\_CLASS\_ID** - MtlBase Library.

**THE\_SCENE\_CLASS\_ID** - The Scene.

**MEDIT\_CLASS\_ID** - Materials Editor.

Subclasses of **GEOMOBJECT\_CLASS\_ID**

Built into core

**TRIOBJ\_CLASS\_ID** - TriObject

**PATCHOBJ\_CLASS\_ID** - PatchObject

Primitives

**BOXOBJ\_CLASS\_ID** - Box Primitive.

**SPHERE\_CLASS\_ID** - Sphere Primitive.

**CYLINDER\_CLASS\_ID** - Cylinder Primitive.

**CONE\_CLASS\_ID** - Cone Primitive.

**TORUS\_CLASS\_ID** - Torus Primitive.

**TUBE\_CLASS\_ID** - Tube Primitive.

**HEDRA\_CLASS\_ID** - Hedra Primitive.

**TEAPOT\_CLASS\_ID1** - The teapot is unique in that it uses both  
DWORDs in its class IDs.

**TEAPOT\_CLASS\_ID2** - The teapot is unique in that it uses both  
DWORDs in its class IDs.

**PATCHGRID\_CLASS\_ID** - Patch Grid

Particles

**RAIN\_CLASS\_ID** - Rain Particle System

**SNOW\_CLASS\_ID** - Snow Particle System

Subclasses of Object Snaps

**GRID\_OSNAP\_CLASS\_ID** - Class\_ID(0x62f565d6, 0x110a1f97)

Space Warp Objects

**WAVEOBJ\_CLASS\_ID** - Wave Space Warp Object

Shapes

**SPLINE3D\_CLASS\_ID** - Spline3D Shape

**NGON\_CLASS\_ID** - NGon

**DONUT\_CLASS\_ID** - Donut Shape

**STAR\_CLASS\_ID** - Start Shape

**RECTANGLE\_CLASS\_ID** - Rectangle Shape

**HELIX\_CLASS\_ID** - Hexlix Shape

**ELLIPSE\_CLASS\_ID** - Ellipse Shape

**CIRCLE\_CLASS\_ID** - Circle Shape

The basic lofter class

**LOFTOBJ\_CLASS\_ID** - This is the basic Loft object class

**LOFT\_DEFCURVE\_CLASS\_ID** - This is the class which defines loft object deformation curves. These are the Scale, Twist, Teeter, Bevel and Fit curves which deform the basic loft object mesh.

Standard 3ds max implementation of the lofter

**LOFT\_GENERIC\_CLASS\_ID**

Target objects

**TARGET\_CLASS\_ID** - Light and Camera target objects.

Morph objects

**MORPHOBJ\_CLASS\_ID** - Morph objects.

Subclasses of **CAMERA\_CLASS\_ID**

**SIMPLE\_CAM\_CLASS\_ID** - Free Camera

**LOOKAT\_CAM\_CLASS\_ID** - Target Camera

Subclasses of **LIGHT\_CLASS\_ID**:

**OMNI\_LIGHT\_CLASS\_ID** - Omni Light

**SPOT\_LIGHT\_CLASS\_ID** - Spot Light

**DIR\_LIGHT\_CLASS\_ID** - Directional Light

**FSPOT\_LIGHT\_CLASS\_ID** - Free Spot Light

**TDIR\_LIGHT\_CLASS\_ID** - Target Directional Light

Subclasses of **HELPER\_CLASS\_ID**

**DUMMY\_CLASS\_ID** - Dummy Object

**BONE\_CLASS\_ID** - Bones System

**TAPEHELP\_CLASS\_ID** - Tape Helper

**GRIDHELP\_CLASS\_ID** - Grid Helper

**POINTHELP\_CLASS\_ID** - Point Helper

**PROTHELP\_CLASS\_ID** - Protractor (Angle-Measuring) Helper

Subclasses of **MATERIAL\_CLASS\_ID**

**CMTL\_CLASS\_ID** - Top/Bottom Material.

**MULTI\_CLASS\_ID** - Multi Material

**DOUBLESIDED\_CLASS\_ID** - Double sided Material

**MIXMAT\_CLASS\_ID** - Mix Material

Subclasses of **TEXMAP\_CLASS\_ID**

**CHECKER\_CLASS\_ID** - Checker Texture

**MARBLE\_CLASS\_ID** - Marble 3D Texture

**MASK\_CLASS\_ID** - Mask Texture

**MIX\_CLASS\_ID** - Mix Texture

**NOISE\_CLASS\_ID** - Noise Texture

**GRADIENT\_CLASS\_ID** - Gradient Texture

**TINT\_CLASS\_ID** - Tint texture

**BMTEX\_CLASS\_ID** - Bitmap texture

**ACUBIC\_CLASS\_ID** - Reflect/refract

**MIRROR\_CLASS\_ID** - Flat mirror

**COMPOSITE\_CLASS\_ID** - Composite texture

**RGBMULT\_CLASS\_ID** - RGB Multiply texture

**FALLOFF\_CLASS\_ID** - Falloff texture

**OUTPUT\_CLASS\_ID** - Output texture

**PLATET\_CLASS\_ID** - Plate glass texture

**VCOL\_CLASS\_ID** - Vertex Color texture

Subclasses of **RENDERER\_CLASS\_ID**

**SREND\_CLASS\_ID** - Default scan-line renderer

Default material class (the Standard material)

**DMTL\_CLASS\_ID** - The Standard material, i.e. the default material.

Subclasses of **SOUNDOBJ\_CLASS\_ID**

**DEF\_SOUNDOBJ\_CLASS\_ID** - Default Sound Object

Subclasses of **OSM\_CLASS\_ID**

**SKEWOSM\_CLASS\_ID** - Skew Object Space Modifier

**BENDOSM\_CLASS\_ID** - Bend Object Space Modifier

**TAPEROSM\_CLASS\_ID** - Taper Object Space Modifier

**TWISTOSM\_CLASS\_ID** - Twist Object Space Modifier

**UVWMAPOSM\_CLASS\_ID** - UVW Map Object Space Modifier

**SELECTOSM\_CLASS\_ID** - Volume Selection Object Space Modifier

**MATERIALOSM\_CLASS\_ID** - Material Object Space Modifier

**SMOOTHOSM\_CLASS\_ID** - Smooth Object Space Modifier

**NORMALOSM\_CLASS\_ID** - Normal Object Space Modifier  
**OPTIMIZEOSM\_CLASS\_ID** - Optimize Object Space Modifier  
**EXTRUDEOSM\_CLASS\_ID** - Extrude Object Space Modifier  
**AFFECTREGION\_CLASS\_ID** - Affect Region Object Space Modifier  
**SUB\_EXTRUDE\_CLASS\_ID** - Face Extrude Object Space Modifier  
**TESSELLATE\_CLASS\_ID** - Tessellate Object Space Modifier  
**DELETE\_CLASS\_ID** - Delete Mesh Object Space Modifier  
**MESHSELECT\_CLASS\_ID** - Mesh Select Object Space Modifier  
**UVW\_XFORM\_CLASS\_ID** - UVW XForm Object Space Modifier  
**SURFREVOISM\_CLASS\_ID** - Lathe Object Space Modifier  
**DISPLACEOSM\_CLASS\_ID** - Displace Object Space Modifier  
**DISPLACE\_OBJECT\_CLASS\_ID** - Displace World Space Modifier Object  
**DISPLACE\_WSM\_CLASS\_ID** - Displace World Space Modifier  
**SINEWAVE\_OBJECT\_CLASS\_ID** - Ripple Object Space Modifier  
**SINEWAVE\_CLASS\_ID** - Ripple World Space Modifier Object  
**SINEWAVE\_OMOD\_CLASS\_ID** - Ripple World Space Modifier  
**LINWAVE\_OBJECT\_CLASS\_ID** - Wave Object Space Modifier  
**LINWAVE\_CLASS\_ID** - Wave World Space Modifier Object  
**LINWAVE\_OMOD\_CLASS\_ID** - Wave World Space Modifier  
**GRAVITYOBJECT\_CLASS\_ID** - Gravity World Space Modifier Object  
**GRAVITYMOD\_CLASS\_ID** - Gravity World Space Modifier  
**WINDOBJECT\_CLASS\_ID** - Gravity World Space Modifier Object  
**WINDMOD\_CLASS\_ID** - Gravity World Space Modifier  
**DEFLECTOBJECT\_CLASS\_ID** - Deflect World Space Modifier Object  
**DEFLECTMOD\_CLASS\_ID** - Deflect World Space Modifier  
**BOMB\_OBJECT\_CLASS\_ID** - Bomb World Space Modifier Object  
**BOMB\_CLASS\_ID** - Bomb World Space Modifier

The following are Class\_IDs for various controllers:

**LININTERP\_FLOAT\_CLASS\_ID** - Linear float controller.  
**LININTERP\_POSITION\_CLASS\_ID** - Linear position controller

**LININTERP\_ROTATION\_CLASS\_ID** - Linear rotation controller  
**LININTERP\_SCALE\_CLASS\_ID** - Linear scale controller  
**PRS\_CONTROL\_CLASS\_ID** - Position/Rotation/Scale Controller  
**LOOKAT\_CONTROL\_CLASS\_ID** - Lookat controller  
**HYBRIDINTERP\_FLOAT\_CLASS\_ID** - Bezier float controller.  
**HYBRIDINTERP\_POSITION\_CLASS\_ID** - Bezier position controller  
**HYBRIDINTERP\_ROTATION\_CLASS\_ID** - Bezier rotation controller  
**HYBRIDINTERP\_POINT3\_CLASS\_ID** - Bezier Point3 controller  
**HYBRIDINTERP\_SCALE\_CLASS\_ID** - Bezier scale controller  
**HYBRIDINTERP\_COLOR\_CLASS\_ID** - Bezier color controller  
**TCBINTERP\_FLOAT\_CLASS\_ID** - TCB Float Controller  
**TCBINTERP\_POSITION\_CLASS\_ID** - TCB Position Controller  
**TCBINTERP\_ROTATION\_CLASS\_ID** - TCB Rotation Controller  
**TCBINTERP\_POINT3\_CLASS\_ID** - TCB Point3 Controller  
**TCBINTERP\_SCALE\_CLASS\_ID** - TCB Scale Controller  
**PATH\_CONTROL\_CLASS\_ID** - Path Controller  
**EULER\_CONTROL\_CLASS\_ID** - Euler Angle Controller  
**EXPR\_POS\_CONTROL\_CLASS\_ID** - Expression Position Controller  
**EXPR\_P3\_CONTROL\_CLASS\_ID** - Expression Point3 Controller  
**EXPR\_FLOAT\_CONTROL\_CLASS\_ID** - Expression Float Controller  
**EXPR\_SCALE\_CONTROL\_CLASS\_ID** - Expression Scale Controller  
**EXPR\_ROT\_CONTROL\_CLASS\_ID** - Expression Rotation Controller  
**FLOATNOISE\_CONTROL\_CLASS\_ID** - Noise Float Controller  
**POSITIONNOISE\_CONTROL\_CLASS\_ID** - Noise Position Controller  
**POINT3NOISE\_CONTROL\_CLASS\_ID** - Noise Point3 Controller  
**ROTATIONNOISE\_CONTROL\_CLASS\_ID** - Noise Rotation Controller  
**SCALENOISE\_CONTROL\_CLASS\_ID** - Noise Scale Controller

**SURF\_CONTROL\_CLASSID** - Surface Position Controller

**LINKCTRL\_CLASSID** - Link Inheritance Controller

## List of Super Class IDs

See Also: [List of Class IDs](#).

The following Super Class IDs are defined in

`\MAXSDK\INCLUDE\PLUGAPI.H`: Note: **typedef `ulong` `SClass_ID`;**

**GEOMOBJECT\_CLASS\_ID** - Used by geometric objects.

**CAMERA\_CLASS\_ID** - Used by plug-in cameras.

**LIGHT\_CLASS\_ID** - Used by plug-in lights.

**SHAPE\_CLASS\_ID** - Used by spline shapes.

**HELPER\_CLASS\_ID** - Used by helper objects.

**SYSTEM\_CLASS\_ID** - Used by system plug-ins.

**OSM\_CLASS\_ID** - Used by Object Space Modifiers.

**WSM\_CLASS\_ID** - Used by Space Warp Modifiers (World Space Modifiers).

**WSM\_OBJECT\_CLASS\_ID** - Used by Space Warp Objects (World Space Modifier Objects).

**SCENE\_IMPORT\_CLASS\_ID** - Used by Scene Import plug-ins.

**SCENE\_EXPORT\_CLASS\_ID** - Used by Scene Export plug-ins.

**BMM\_STORAGE\_CLASS\_ID** - Bitmap storage.

**BMM\_FILTER\_CLASS\_ID** - Used by Image Filter plug-ins.

**BMM\_IO\_CLASS\_ID** - Used by Image Loading/Saving plug-ins (IO Modules).

**BMM\_DITHER\_CLASS\_ID** - Bitmap dithering.

**BMM\_COLORCUT\_CLASS\_ID** - Bitmap color cut.

**USERDATATYPE\_CLASS\_ID** - This is obsolete.

**MATERIAL\_CLASS\_ID** - Used by plug-in Materials.

**TEXMAP\_CLASS\_ID** - Used by plug-in Textures.

**UVGEN\_CLASS\_ID** - Used by the UVGen class that handles the UV coordinate interface.

**XYZGEN\_CLASS\_ID** - Used by the XYZGen class.

**TEXOUTPUT\_CLASS\_ID** - Used by the TextureOutput class.

**SOUNDOBJ\_CLASS\_ID** - Used by sound object plug-ins.

**FLT\_CLASS\_ID** - Used by image processing filter plug-ins.

**RENDERER\_CLASS\_ID** - Used by plug-in renderers.

**BEZFONT\_LOADER\_CLASS\_ID** - Used by bezier font loader plug-ins.

**REF\_MAKER\_CLASS\_ID** - These are items such as material libraries, the scene, and medit.

**ATMOSPHERIC\_CLASS\_ID** - Used by atmospheric plug-ins.

**UTILITY\_CLASS\_ID** - Used by utility plug-ins.

**TRACKVIEW\_UTILITY\_CLASS\_ID** - Used by Track View Utility plug-ins.

**FRONTEND\_CONTROL\_CLASS\_ID** - Used by Front End Controller plug-ins.

**MOT\_CAP\_DEV\_CLASS\_ID** - Used by Motion Capture plug-ins.

**MOT\_CAP\_DEVBINDING\_CLASS\_ID** - Used by Motion Capture Device Bindings.

**OSNAP\_CLASS\_ID** - Used by Object Snap plug-ins.

**TEXMAP\_CONTAINER\_CLASS\_ID** - In 3ds max 2.0 and later this new super class has been added. This is used by the Standard material to contain its Texmaps. The track view filter code has been modified so it now looks for this class and will filter it out when maps are being filtered out, instead of having special purpose code for the Standard Material. This will permit plug-in developers to put their Texmaps down in a sub-directory like the Standard material does.

**RENDER\_EFFECT\_CLASS\_ID** - Used by Render Effect plug-ins (R3 and later only).

**FILTER\_KERNEL\_CLASS\_ID** - Used by Anti-Aliasing Filter plug-ins (R3 and later only)

**SHADER\_CLASS\_ID** - Used by Shader plug-ins which plug-in to the Standard2 material (R3 and later only).

**COLPICK\_CLASS\_ID** - Used by color picker plug-ins (R3 and later only)

**SHADOW\_TYPE\_CLASS\_ID** - Used by shadow generators (R3 and later only)

**GUP\_CLASS\_ID** - Used by Global Utility Plug-Ins (R3 and later only)

**LAYER\_CLASS\_ID** - Used by VIZ layer (R3 and later only).

**SCHEMATICVIEW\_UTILITY\_CLASS\_ID** - Used by Schematic View (R3 and later only)

**SAMPLER\_CLASS\_ID** - Used by Sampler plug-ins (R3 and later only)

Super-class ID's used by various controllers

**CTRL\_FLOAT\_CLASS\_ID** - Used by float controllers.

**CTRL\_POINT3\_CLASS\_ID** - Used by Point3 controllers.

**CTRL\_MATRIX3\_CLASS\_ID** - Used by Matrix3 controllers.

**CTRL\_POSITION\_CLASS\_ID** - Used by position controllers.

**CTRL\_ROTATION\_CLASS\_ID** - Used by rotation controllers.

**CTRL\_SCALE\_CLASS\_ID** - Used by scale controllers.

**CTRL\_MORPH\_CLASS\_ID** - Used by morph controllers.

**CTRL\_SHORT\_CLASS\_ID** - This SuperClassID is obsolete.

**CTRL\_POINT2\_CLASS\_ID** - This SuperClassID is obsolete.

**CTRL\_QUAT\_CLASS\_ID** - This SuperClassID is obsolete.

**CTRL\_INTEGER\_CLASS\_ID** - This SuperClassID is obsolete.

**CTRL\_POS\_CLASS\_ID** - This SuperClassID is obsolete.

**CTRL\_COLOR\_CLASS\_ID** - This SuperClassID is obsolete.

**CTRL\_COLOR24\_CLASS\_ID** - This SuperClassID is obsolete.

**CTRL\_USERTYPE\_CLASS\_ID** - This SuperClassID is obsolete.

## Class ChangeForegroundCallback

See Also: [Class CommandMode](#), [Class IScene](#).

class ChangeForegroundCallback

### Description:

The purpose of this callback is to call **INode::FlagForeground()** for any nodes in the scene that are supposed to be in the foreground.

### Methods:

#### Prototype:

**virtual BOOL IsValid()=0;**

#### Remarks:

Implemented by the Plug-In.

Returns TRUE if this foreground callback is valid ; otherwise FALSE. When the system needs to redraw the viewports the system checks to see if the current foreground callback is the same one that was in place the last time. If it is the same the system will call this method to see if the state is valid. If the state is not valid (this method returns FALSE) then the foreground and background are re-built and then the **Validate()** method is called so this foreground callback may note that it is invalid.

#### Prototype:

**virtual void Invalidate()=0;**

#### Remarks:

Implemented by the Plug-In.

The system calls this method of the plug-in when the foreground state is no longer valid. The plug-in should set a flag internally to indicate it is invalid. For example if the current node selection set changes the system calls this method to mark it as invalid so that the next time the system goes to redraw the viewports it can determine that it would need to rebuild the foreground and the background.

#### Prototype:

**virtual void Validate()=0;**

**Remarks:**

Implemented by the Plug-In.

The system calls this method when the foreground state is valid. For example after the foreground and background buffers have been re-built this method is called. The plug-in should set a flag internally to indicate it is valid.

**Prototype:**

**virtual void callback(TimeValue t,IScene \*scene)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is used to enumerate the scene and flag nodes in the scene that need to go in the foreground.

**Parameters:**

**TimeValue t**

The time to flag the nodes.

**IScene \*scene**

This interface pointer provides methods to enumerate all nodes in the scene, or to flag nodes that are selected, animated or dependent on a given node.

## Class FPInterface (and Class FPMixinInterface)

See Also: [Class BaseInterface](#), [Class ClassDesc](#), [Class FPInterfaceDesc](#), [Class Interface\\_ID](#), [Class ActionTable](#), [List of FPStatus Values](#), [Function Publishing System](#).

```
class FPInterface : public BaseInterface
```

### Description:

This class is available in release 4.0 and later only.

The Function Publishing system makes use of this class. Functions are published in one or more Interfaces by a plug-in. Each interface is represented by an instance of a class derived from this base class.

Note that the Function Publishing class hierarchy is as follows:

**FPInterface:** This is the base class for all interfaces, the prime client type for using interfaces.

**FPInterfaceDesc:** This is the class which contains interface metadata.

**FPStaticInterface:** This is the class to use as the base class for defining static or core virtual interface classes.

**FPMixinInterface:** This is for use as the base class for defining object-based mixin interface classes, in this case you also use FPInterfaceDesc for mixin interface descriptors.

### Methods:

public:

### Prototype:

```
virtual FPInterfaceDesc* GetDesc() = 0;
```

### Remarks:

Returns a pointer to the class which contains the interface [= 4\)](#)  
[BSPSPopupOnMouseOver\(event\);">metadata](#).

### Prototype:

```
virtual FPStatus Invoke(FunctionID fid, TimeValue t=0,  
FPParams* params=NULL);
```

### Remarks:

**Parameters:****FunctionID fid**

The function ID of the function to invoke.

**TimeValue t=0**

The timevalue at which to invoke the function.

**FPParams\* params=NULL**

The FPParams to pass.

**Return Value:**

The FPStatus. See the [List of FPStatus Values](#) for details

**Prototype:**

```
virtual inline FPStatus Invoke(FunctionID fid, FPParams*  
params);
```

**Remarks:**

This method will invoke the specified function.

**Parameters:****FunctionID fid**

The function ID of the function to invoke.

**FPParams\* params**

The FPParams to pass.

**Return Value:**

The FPStatus. See the [List of FPStatus Values](#) for details

**Prototype:**

```
virtual FPStatus Invoke(FunctionID fid, TimeValue t, FPValue&  
result, FPParams* params=NULL);
```

**Remarks:**

This method will invoke the specified function.

**Parameters:****FunctionID fid**

The function ID of the function to invoke.

**TimeValue t**

The timevalue at which to invoke the function.

**FPValue& result**

A reference to the resulting FPValue.

**FPParams\* params=NULL**

The FPParams to pass.

**Return Value:**

The FPStatus. See the [List of FPStatus Values](#) for details

**Prototype:**

```
virtual inline FPStatus Invoke(FunctionID fid, FPValue& result,  
FPParams* params=NULL);
```

**Remarks:**

This method will invoke the specified function.

**Parameters:****FunctionID fid**

The function ID of the function to invoke.

**FPValue& result**

A reference to the resulting FPValue.

**FPParams\* params=NULL**

The FPParams to pass.

**Return Value:**

The FPStatus. See the [List of FPStatus Values](#) for details

**Prototype:**

```
virtual FunctionID FindFn(TCHAR* name);
```

**Remarks:**

This method returns a function ID based on the name of the function specified.

**Parameters:****TCHAR\* name**

The name of the function to retrieve the Function ID for.

**Prototype:**

**virtual BOOL IsEnabled(FunctionID actionID);**

**Remarks:**

This method allows you to check whether a specific action function is enabled, in which case the method will return TRUE. If the action function is not enabled FALSE will be returned.

**Parameters:**

**FunctionID actionID**

The function ID of the action you wish to check the enabled state for.

**Prototype:**

**virtual BOOL IsChecked(FunctionID actionID);**

**Remarks:**

This method allows you to check whether a specific action function is checked, in which case the method will return TRUE. If the action function is not checked FALSE will be returned.

**Parameters:**

**FunctionID actionID**

The function ID of the action you wish to check the checked state for.

**Prototype:**

**virtual BOOL IsVisible(FunctionID actionID);**

**Remarks:**

This method allows you to check whether a specific action function is visible, in which case the method will return TRUE. If the action function is not visible FALSE will be returned.

**Parameters:**

**FunctionID actionID**

The function ID of the action you wish to check the visibility state for.

**Prototype:**

**virtual FunctionID GetIsEnabled(FunctionID actionID);**

**Remarks:**

This method will return the isEnabled ID for the specified action function.

**Parameters:****FunctionID actionID**

The function ID of the action you wish to get the isEnabled ID for.

**Prototype:**

```
virtual FunctionID GetIsChecked(FunctionID actionID);
```

**Remarks:**

This method will return the isChecked ID for the specified action function.

**Parameters:****FunctionID actionID**

The function ID of the action you wish to get the isChecked ID for.

**Prototype:**

```
virtual FunctionID GetIsVisible(FunctionID actionID);
```

**Remarks:**

This method will return the isVisible ID for the specified action function.

**Parameters:****FunctionID actionID**

The function ID of the action you wish to get the isVisible ID for.

**Prototype:**

```
virtual bool Acquire(FPInterfaceCallback* fpicb=NULL);
```

**Remarks:**

This method is called when MAXScript makes a reference to this object. Call this method with a non-NULL **FPInterfaceCallback** so the interface can signal deletion (usually a mixin).

**Parameters:****FPInterfaceCallback\* fpicb=NULL**

A pointer to an FPInterfaceCallback class.

**Return Value:**

This method should return TRUE if it needs Release() called.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void Release(FPInterfaceCallback* fpicb=NULL);
```

**Remarks:**

This method is called when MAXScript deletes a reference to this object.

**Parameters:**

**FPInterfaceCallback\* fpicb=NULL**

A pointer to an FPInterfaceCallback class.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual ActionTable* GetActionTable();
```

**Remarks:**

This method returns a pointer to the ActionTable.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void EnableActions(BOOL onOff);
```

**Remarks:**

This method allows you to enable or disable the entire set of actions in the interface. You might want to use this method if the actions are only to be active during certain periods in the running of 3ds max. Usually, this control is achieved via ActionTable contexts.

**Parameters:**

**BOOL onOff**

TRUE to enable actions, FALSE to disable them.

**Default Implementation:**

**{ }**

## Class FPMixinInterface

See Also: [Class FPInterface](#), [Class FPInterfaceDesc](#), [Function Publishing System](#).

```
class FPMixinInterface : public FPInterface
```

### Description:

This class is available in release 4.0 and later only.

A "Mixin" interface provides a way for a plug-in to expose some of its functionality for use by other plug-ins or MAXScript. The notion of "Mixin" refers to the idea that the interface is a sub-class of the plug-in class and thus "mixed in" with it. Many classes in the SDK now inherit from FPMixinInterface in order to expose some of their functionality.

Developers should see the documentation for [Class FPInterface](#) for reference on this class as well.

For an overview of the Function Publishing System as a whole, see the Advanced Topics section [Function Publishing System](#).

### Methods:

public:

### Prototype:

```
virtual FPInterfaceDesc* GetDescByID(Interface_ID id);
```

### Remarks:

This method is used to directly implement **FPInterface::GetDesc()** in your public virtual base mixin class, like this;

```
FPInterfaceDesc* GetDesc() { return  
GetDescByID(THIS_INTERFACE_ID); }
```

Then implement a GetDescByID() in the implementing class to avoid link export issues.

### Parameters:

**Interface\_ID id**

The unique interface ID by which to get the FPInterfaceDesc.

### Default Implementation:

```
{ return &nullInterface; }
```

**Prototype:**

```
virtual Interface_ID GetID();
```

**Remarks:**

This method overrides GetID() in those interfaces that do not publish metadata but instead have a unique Interface\_ID for quick internal identification in implementation code that might be shared by a bunch of mixin interfaces.

**Return Value:**

The Interface\_ID.

**Default Implementation:**

```
{ return GetDesc()->ID; }
```

## Class FPInterfaceDesc

See Also: [Class FPInterface](#), [= 4\) BSPSPopupOnMouseOver\(event\);">Class ClassDesc](#), [Class Interface\\_ID](#), [Class FPEnum](#), [Class FPFunctionDef](#), [Class FPPropDef](#), [Class ActionTable](#), [Template Class Tab](#), [Function Publishing System](#).

```
class FPInterfaceDesc : public FPInterface
```

### Description:

This class is available in release 4.0 and later only.

This is the Function Publishing interface descriptor. This is usually a static instance of the implementation interface. The constructor for this class uses the same var-args technique used by the ParamBlockDesc2 constructor, enabling descriptive information for all the functions in the interface to be supplied in one constructor call.

The FPInterfaceDesc class, an FPInterface that contains the metadata for an interface, is a distinguished singleton instance per interface kept in a Tab<> in ClassDesc. This class is subclassed typically by static and core interfaces and instantiated by mixins to provide their metadata.

Note the following typedef: **typedef FPInterfaceDesc FPStaticInterface**

### Data Members:

public:

#### **Interface\_ID ID;**

The unique ID of the interface.

#### **TSTR internal\_name;**

The fixed internal name for the interface.

#### **StringResID description;**

The description string resource ID.

#### **ClassDesc\* cd;**

Points to the publishing plug-in's ClassDesc.

#### **USHORT flags;**

The flag bits. One or more of the following values:

#### **FP\_ACTIONS**

Marks this as an Action Interface, holding only UI modal, zero parameter action functions.

### **FP\_MIXIN**

Marks this as a Mixin Interface, it is implemented directly by the plug-in class, so the methods in it are virtual on the plugin's objects.

### **FP\_CORE**

Marks this as a 3ds max Core Interface, available through **GetCOREInterface(Interface\_ID)**.

### **FP\_STATIC\_METHODS**

This interface is used as a static method interface in MAXScript, properties are not directly callable

### **FP\_SCRIPTED\_CLASS**

Internal use only: Belongs to a scripted plug-in class.

### **FP\_TEMPORARY**

Internal use only: Temporary descriptor built during scene load.

### **Tab<FPFunctionDef\*> functions;**

A table of descriptors for individual functions in this interface.

### **Tab<FPPropDef\*> props;**

A Table of descriptors for individual properties in this interface.

### **Tab<FPEnum\*> enumerations;**

A table of any symbolic enums for the interface. This is a Tab<> of pointers to **FPEnum** class instances which themselves contain a Tab<> of name, code pairs.

The following data members are for scripted plug-ins if this interface belongs to a scripted plug-in class.

### **MSPluginClass\* pc;**

The scripted class if non-NULL (gc-protected by the scripted plugin class).

### **Rollout\* rollout;**

The rollout if specified (gc-protected by the scripted plugin class).

### **ActionTable\* action\_table;**

The table published for this action interface.

For more information, see [Class ActionTable](#).

## **Methods:**

public:

**Prototype:**

**FPInterfaceDesc();**

**Remarks:**

Constructor. No initialization is performed.

**Prototype:**

**FPInterfaceDesc(Interface\_ID id, TCHAR\* int\_name,  
StringResID descr, ClassDesc\* cd, ULONG flag, ...);**

**Remarks:**

Constructor.

**Parameters:**

**Interface\_ID id**

The unique ID of the interface.

**TCHAR\* int\_name**

The fixed internal name for the interface.

**StringResID descr**

A string resource ID containing the description for this interface class.

**ClassDesc\* cd**

A pointer to a ClassDesc class descriptor of the publishing plug-in.

**ULONG flag**

The flag bits. One or more of the following values: **FP\_ACTIONS**, **FP\_MIXIN**, **FP\_CORE**, **FP\_STATIC\_METHODS**, **FP\_SCRIPTED\_CLASS**, **FP\_TEMPORARY**. For a description see the data members descriptions.

...

This constructor takes a variable number of arguments representing the 'properties' sections and function definitions. For more information see the advanced topics on [Function Publishing](#).

**Prototype:**

**virtual void Init();**

**Remarks:**

This is a virtual method called by the varargs-based constructors for interface descriptors and static interfaces, so that they have an opportunity to do runtime initialization of any extra state data you add to these interfaces (usually to static interfaces). Since such interfaces are usually constructed with the built-in varargs constructor, there is no way to do custom initialization without a hook like the `Init()` call. Your static interface would provide an implementation of `Init()` to do any special initialization.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
void LoadDescriptor(Interface_ID id, TCHAR* int_name,  
StringResID descr, ClassDesc* cd, ULONG flag, ...);
```

**Remarks:**

This method relates to **`Init()`**. In some cases, you really do need to provide your own constructor or set of constructors for a static interface or descriptor, but you still want to be able to load it with all the interface metadata that the built-in varargs constructor does. You can do this by calling the **`LoadDescriptor()`** method at any point in your own constructors and it takes the same arguments as the built-in varargs constructor.

**Parameters:****Interface\_ID id**

The unique ID of the interface.

**TCHAR\* int\_name**

The fixed internal name for the interface.

**StringResID descr**

A string resource ID containing the description for this interface class.

**ClassDesc\* cd**

A pointer to a `ClassDesc` class descriptor of the publishing plug-in.

**ULONG flag**

The flag bits. One or more of the following values: **`FP_ACTIONS`**, **`FP_MIXIN`**, **`FP_CORE`**, **`FP_STATIC_METHODS`**,

**FP\_SCRIPTED\_CLASS, FP\_TEMPORARY.** For a description see the data members descriptions.

...

This method takes a variable number of arguments representing the 'properties' sections and function definitions. For more information see the advanced topics on [Function Publishing](#).

## Construction Utilities

### Prototype:

```
void SetClassDesc(ClassDesc* i_cd);
```

### Remarks:

This method sets the ClassDesc pointer associated FPInterfaceDesc class. You can only call this method once on a descriptor and then only if it has been constructed initially with a NULL cd. See the notes in the constructor.

### Parameters:

```
ClassDesc* i_cd
```

This points to the ClassDesc class descriptor to set.

### Prototype:

```
va_list check_fn(va_list ap, int id);
```

### Remarks:

This is used internally.

### Prototype:

```
va_list scan_fn(va_list ap, int id, int index);
```

### Remarks:

This is used internally.

### Prototype:

```
va_list check_prop(va_list ap, int id);
```

**Remarks:**

This is used internally.

**Prototype:**

```
va_list scan_prop(va_list ap, int id, int index);
```

**Remarks:**

This is used internally.

**Prototype:**

```
va_list check_enum(va_list ap, EnumID id);
```

**Remarks:**

This is used internally.

**Prototype:**

```
va_list scan_enum(va_list ap, EnumID id, int index);
```

**Remarks:**

This is used internally.

**Metadata Access****Prototype:**

```
FPInterfaceDesc* GetDesc();
```

**Remarks:**

This method returns a pointer to the descriptor for this Function Publishing interface descriptor.

**Default Implementation:**

```
{ return this; }
```

**Prototype:**

```
FPFunctionDef* GetFnDef(FunctionID fid);
```

**Remarks:**

This method returns a pointer to the function definition of a specific function identified by its ID. Calls to this method, given an **FPInterface\***, can be made indirectly through **FPInterface::GetDesc()**. For example;  
**FPFunctionDef\* fd = fpi->GetDesc()->GetFnDef(foo\_move);**

**Parameters:**

**FunctionID fid**

The unique function ID used to identify the function.

**Prototype:**

**ActionTable\* GetActionTable();**

**Remarks:**

This method returns a pointer to the action table.

**Default Implementation:**

**{ return action\_table; }**

## **Global Actions**

**Prototype:**

**void EnableActions(BOOL onOff);**

**Remarks:**

This method allows you to enable or disable the entire set of actions in the interface. You might want to use this method if the actions are only to be active during certain periods in the running of 3ds max. Usually, this control is achieved via ActionTable contexts.

**Parameters:**

**BOOL onOff;**

TRUE to enable actions, FALSE to disable actions.

## **Overridable HInstance and resource access from owning module**

**Prototype:**

**virtual HINSTANCE HInstance();**

**Remarks:**

This method will return a handle to the owning instance.

**Prototype:**

```
virtual TCHAR* GetRsrcString(StringResID id);
```

**Remarks:**

This method returns the string associated with a specified String Resource ID

**Parameters:**

**StringResD id**

The string resource ID for which you want to obtain the string.

## Class Interface\_ID

See Also: [Class FPInterface](#), [Function Publishing System](#).

class Interface\_ID

### Description:

This class is available in release 4.0 and later only.

This class is the interface ID for the Function Publishing System of 3ds max. This class is structurally very similar to a Class\_ID, containing two randomly-chosen longwords to provide a unique global ID. The various constructors assign a value to each of these. There are also methods to assign and retrieve the individual parts and operators to check for equality or inequality.

All the methods of this class are implemented by the system.

### Methods:

public:

### Prototype:

**Interface\_ID();**

### Remarks:

Constructor. The two parts of the ID are initialized to **0xffffffff**.

### Prototype:

**Interface\_ID(const Interface\_ID& iid);**

### Remarks:

Constructor. The two parts of the ID are initialized from the corresponding parts of the Interface\_ID passed.

### Parameters:

**const Interface\_ID& iid**

The ID whose parts are used to initialize this ID.

### Prototype:

**Interface\_ID(ulong aa, ulong bb);**

### Remarks:

Constructor. The two parts of the ID are initialized from the from the parts passed.

**Parameters:**

**ulong aa**

Passed to initialize the first part of the ID.

**ulong bb**

Passed to initialize the second part of the ID.

**Prototype:**

**ULONG PartA();**

**Remarks:**

Returns the first part of the ID.

**Prototype:**

**ULONG PartB();**

**Remarks:**

Returns the second part of the ID.

**Prototype:**

**void SetPartA( ulong aa );**

**Remarks:**

Sets the first part of the ID.

**Parameters:**

**ulong aa**

Passed to set the first part.

**Prototype:**

**void SetPartB( ulong bb );**

**Remarks:**

Sets the second part of the ID.

**Parameters:**

**ulong bb**

Passed to set the second part.

**Prototype:**

**int operator==(const Interface\_ID& iid) const;**

**Remarks:**

Equality operator. Returns nonzero if the two parts of the ID are equal to the ID passed; otherwise zero.

**Parameters:**

**const Interface\_ID& iid**

The ID to check.

**Prototype:**

**int operator!=(const Interface\_ID& iid) const;**

**Remarks:**

Inequality operator. Returns nonzero if either of the parts of the ID are NOT equal to the ID passed; otherwise zero.

**Parameters:**

**const Interface\_ID& iid**

The ID to check.

**Prototype:**

**Interface\_ID& operator=(const Interface\_ID& iid);**

**Remarks:**

Assignment operator.

**Parameters:**

**const Interface\_ID& iid**

The ID to assign from.

## Class FPFunctionDef

See Also: [Class FPInterface](#), [Class FPInterfaceDesc](#), [Class FPActionDef](#), [Class FPParamDef](#), [Template Class Tab](#), [Function Publishing System](#), [List of Param Type Choices](#) , [List of ParamType2 Choices](#).

class FPFunctionDef

### Description:

This class is available in release 4.0 and later only.

This class stores data about a single function of an FPInterface. A table of pointers to these objects is a data member of Class **FPInterfaceDesc**.

### Data Members:

public:

#### **FunctionID ID;**

The interface-local ID (unique to the interface only) used to identify the function in calls.

#### **TSTR internal\_name;**

The fixed internal name for the function.

#### **StringResID description;**

The description string resource.

#### **USHORT flags;**

The internal flag bits. One of more of the following values:

##### **FP\_ACTION**

Indicates this is an action function.

##### **FP\_HAS\_UI**

Indicates the action has UI specified.

##### **FP\_ICONRES**

Indicates icon via resource ID.

##### **FP\_ICONFILE**

Indicates icon via bmp file + index.

##### **FP\_HAS\_SHORTCUT**

Indicates has default keyboard shortcut.

##### **FP\_HAS\_KEYARGS**

Indicates function has some optional keyword args defined.

### **FP\_VAR\_ARGS**

Indicates a variable number of args, pass args directly in a FPParams instance.

### **FP\_NO\_REDRAW**

Do not flag need for viewport redraw when function is invoked, MAXScript defaults to flag redraw.

### **ParamType2 result\_type;**

The type of value returned by the function.

### **EnumID enumID;**

The ID of symbolic enumeration in owning interface if any.

### **FPActionDef\* action\_def;**

The extra metadata if function in an Action interface.

### **Tab<FPParamDef\*> params;**

The descriptors for parameters to this function.

### **Methods:**

public:

### **Prototype:**

**FPFunctionDef();**

### **Remarks:**

Constructor. The data members are initialized as follows:

**flags = 0; description = 0; action\_def = NULL; enumID = FP\_NO\_ENUM;**

## Class FPPropDef

See Also: [Class FPParamOptions](#), [List of Param Type Choices](#)

class FPPropDef

### Description:

This class is available in release 4.0 and later only.

The 'properties' section follows the function definitions. Each property has a single entry defining the function IDs for the getter and setter functions, a fixed internal property name, a descriptor string resource ID and the property type. If the property is read-only and there is no setter function, specify **FP\_NO\_FUNCTION** for the setter ID.

### Data Members:

public:

**FunctionID** **getter\_ID;**

The interface-local ID for getter method.

**FunctionID** **setter\_ID;**

The interface-local ID for setter method.

**TSTR** **internal\_name;**

The fixed, internal name.

**StringResID** **description;**

The description string resource ID.

**USHORT** **flags;**

The flag bits.

**ParamType2** **prop\_type;**

The property type.

**EnumID** **enumID;**

The ID of symbolic enumeration in owning interface if any.

**FPParamOptions\*** **options;**

Present if non-NULL, used for setter param.

### Methods:

public:

### Prototype:

**FPPropDef();**

**Remarks:**

Constructor. The data members are initialized as follows:

**flags = 0; description = 0; getter\_ID = setter\_ID =  
FPS\_NO\_SUCH\_FUNCTION; enumID = FP\_NO\_ENUM; options  
= NULL;**

## Class FPActionDef

See Also: [Class FPInterface](#), [Class FPInterfaceDesc](#), [Class FPActionDef](#), [Class FPParamDef](#), [Template Class Tab](#), [Class MaxIcon](#), [Function Publishing System](#), [List of ControlType2 Choices](#) , [List of ParamType2 Choices](#).

class FPActionDef

### Description:

This class is available in release 4.0 and later only.

FPActionDef, contains extra descriptor info for Action interface functions.

### Data Members:

public:

**TSTR internal\_cat;**

The fixed, internal category name.

**StringResID category;**

The localizable category string resource ID.

**FunctionID isEnabled\_id;**

The interface function IDs for the isEnabled predicate for this action.

**FunctionID isChecked\_id;**

The interface function IDs for the isChecked predicate for this action.

**FunctionID isVisible\_id;**

The interface function IDs for the isVisible predicate for this action.

**ResID icon\_resID;**

The icon as resource ID.

**TSTR icon\_file;**

The icon as UI .bmp filename, index pair, as per CUI icon specifications.

**short icon\_index;**

The index of the icon associated with the action.

**MaxIcon\* icon;**

The MaxIcon class data associated with this action.

**StringResID button\_text;**

The button text string resource ID, defaults to function description.

**StringResID tool\_tip;**

The tooltip string resource ID, defaults to function description.

**StringResID menu\_text;**

The menu item text string resource ID, defaults to buttonText or function description.

**ControlType2 ctrl\_type;**

The type of UI control, if f\_ui specified.

**ResID ctrl\_pbID;**

The control's host parammap pblock ID.

**MapID ctrl\_mapID;**

The control's host parammap map ID within the block.

**int ctrl\_id;**

The control dialog item ID.

**COLORREF ctrl\_hiCol;**

The highlight color if check button.

**ACCEL shortcut;**

The default keyboard shortcut.

**Methods:**

public:

**Prototype:**

**FPActionDef();**

**Remarks:**

Constructor.

This will initialize the members to their empty default values.

**Prototype:**

**~FPActionDef();**

**Remarks:**

Destructor.

## Class FPParamDef

See Also: [List of ParamType2 Choices](#), [Class FPParamOptions](#), [Template Class Tab](#).

```
class FPParamDef
```

### Description:

This class is available in release 4.0 and later only.

This class contains a descriptor for each published function, found in Tab<> in **FPInterface**.

### Data Members:

public:

**TSTR internal\_name;**

The internal name.

**StringResID description;**

The string resource ID of the description.

**ParamType2 type;**

The parameter type. See the [List of ParamType2 Choices](#) for details.

**EnumID enumID;**

ID of symbolic enumeration in owning interface if any.

**USHORT flags;**

The parameter definition flags;

**FPP\_HAS\_RANGE**

Indicates that the parameter definition contains a range.

**FPP\_HAS\_VALIDATOR**

Indicates that the parameter has a validator.

**FPP\_IN\_PARAM**

In flag used by \_BR ref types to decide when to pass in source values or hand back returns.

**FPP\_OUT\_PARAM**

Out flag used by \_BR ref types to decide when to pass in source values or hand back returns.

**FPP\_IN\_OUT\_PARAM**

In-Out flag used by \_BR ref types to decide when to pass in source values

or hand back returns.

### **FPP\_KEYARG**

If p\_keyArgDefault supplied, the client can use keyword args if supported for this param.

### **FPP\_INDEX**

Parameter values used as indexes, always 0-origin internally, allows client to map to other origins.

### **FPPParamOptions\* options;**

Present if non-NULL, a pointer to the parameter options object.

### **Methods:**

public:

### **Prototype:**

**FPPParamDef();**

### **Remarks:**

Constructor.

### **Default Implementation:**

```
{ description = 0; options = NULL; flags = FPP_IN_OUT_PARAM;  
enumID = FP_NO_ENUM; }
```

# Class FParams

See Also: [Class FPValue](#)

class FParams

## Description:

This class is available in release 4.0 and later only.

This parameters class contains a **Tab<>** of **FPValue**'s, being the actual parameters for an FP Function call.

## Data Members:

public:

**Tab<FPValue> params;**

The table of FPValue's contained in this class.

## Methods:

public:

## Prototype:

**FParams();**

## Remarks:

Constructor.

## Default Implementation:

{ }

## Prototype:

**FParams(int count, ...);**

## Remarks:

Constructor.

## Parameters:

**int count**

The number of parameter values to add.

...

This method takes a variable number of arguments representing the parameter

values that will be stored in the **params** table.

**Prototype:**

**~FPParams();**

**Remarks:**

Destructor.

**Prototype:**

**void Load(int count, ...);**

**Remarks:**

This method loads a number of parameter values.

**Parameters:**

**int count**

The number of parameter values to add.

...

This method takes a variable number of arguments representing the parameter values that will be stored in the **params** table.

## Class FPEnum

See Also: [Class FPInterface](#), [Function Publishing System](#)

class FPEnum

### Description:

This class is available in release 4.0 and later only.

This class contains an ID for the enumeration and a table of structures which contains a name and an integer code for each item. This is used by metadata clients to support symbolic values for **TYPE\_ENUM** types (ints).

One or more symbolic enums, similar to C++ enums, can now be added to an FPInterface's metadata, and individual int parameters and/or results for functions in that interface can be defined as **TYPE\_ENUM** and associated with one of the enum lists. Working in a similar manner as MAXScript, this allows metadata clients to support symbolic encodings for these parameters and results.

Enums are defined in the FPInterface descriptor following the function and property definitions as sets of string/code pairs. Each enum list is identified by a unique integer, similar to function IDs, which is used to associated a **TYPE\_ENUM** parameter or result with its enum. IDs for these would normally be defined somewhere near the function IDs for an interface. For example:

```
// function IDs
enum { bmm_getWidth, bmm_getHeight, bmm_getType,
bmm_copyImage, ...};
// enum IDs
enum { bmm_type, bmm_copy_quality, ...};
```

might be some of the IDs for a possible bitmap manager interface. The two enums provide symbolic codes for the bitmap type and copyImage quality defines in the "bitmap.h" SDK header, such as **BMM\_PALETTED**, **BMM\_TRUE\_32**, **COPY\_IMAGE\_RESIZE\_LO\_QUALITY**, etc. In the descriptor for the interface, any enum lists follow the function and property definitions. They are introduced by the special tag, 'enums', as in the following example:

```

static FPInterfaceDesc bmmfpi (
    BMM_INTERFACE, _T("bmm"), IDS_BMMI, NULL,
    FP_CORE,
    ...
    bmm_copyImage, _T("copyImage"), ...
        _T("copyType"), IDS_COPYTYPE, TYPE_ENUM,
    bmm_copy_quality,
    ...
    properties,
        geo_getType, geo_setType, _T("type"), 0, TYPE_ENUM,
    bmm_type,
        enums,
            bmm_type, 7,
                "lineArt", BMM_LINE_ART,
                "paletted", BMM_PALETTED,
                "gray8", BMM_GRAY_8,
                "gray16", BMM_GRAY_16 ,
                "true16", BMM_TRUE_16,
                "true32", BMM_TRUE_32,
                "true24", BMM_TRUE_64,
            bmm_copy_quality, 4,
                "crop", COPY_IMAGE_CROP,
                "resizeLo", COPY_IMAGE_RESIZE_LO_QUALITY,
                "resizeHi", COPY_IMAGE_RESIZE_HI_QUALITY,
                "useCustom", COPY_IMAGE_USE_CUSTOM,
        end
    );

```

In the above example, the enums are listed following the function & property definitions. They are introduced by the 'enums' tag and consist of an enum ID followed by a count of items, followed by that many string and code pairs. By

attaching them to the interface like this, any number of functions and properties in the interface can use them.

The above example also has function and property definitions showing the use of **TYPE\_ENUM**. The **copyImage** function takes a **copyType** parameter which uses the **bmm\_copy\_quality enum** and the type property uses the **bmm\_type enum**. In all situations where TYPE\_xxx types can be supplied in a descriptor, including the new property definitions, **TYPE\_ENUM** can be used to indicate an int by-value type. **TYPE\_ENUM**'s must always be followed by an enum ID. This is the only case in which the type is specified as a pair of values. **TYPE\_ENUM** parameters and results show up in MAXScript as # names. For example, if a bmm interface was in the variable 'bm1' and the bitmap type was **BMM\_GRAY\_16**,

```
bm1.type
```

```
-> #gray16
```

```
bm1.type = #true32 -- set it to #true24 (code is BMM_TRUE_24)
```

```
bm2 = bm1.copyImage #resizeHi
```

the integer **TYPE\_ENUM** codes are translated back-and-forth to symbolic # names by MAXScript using the definitions in the FPInterface descriptor's enums. If you need to access the enum metadata in an **FPInterfaceDesc**, it is available in the 'enumerations' data member. This is a **Tab<>** of pointers to **FPEnum** class instances which themselves contain a **Tab<>** of name, code pairs. See class **FPEnum** in `\MAXSDK\INCLUDE\FnPub.h` for details.

#### **Data Members:**

public:

```
EnumID ID;
```

```
ID for this enumeration
```

```
typedef struct {
```

```
    TCHAR*name;
```

This is the symbolic name for the enum.

**int code;**

This is the equivalent integer code.

**} enum\_code;**

**Tab<enum\_code> enumeration;**

The table of enumeration codes.

## Class FPInterfaceCallback

See Also: [Class FPInterface](#), [Function Publishing System](#)

class FPInterfaceCallback

### Description:

This class is available in release 4.0 and later only.

FPInterfaceCallback class is registered with an interface on Acquire() so that it can be notified when the interface goes away

### Methods:

public:

### Prototype:

**virtual void InterfaceDeleted()=0;**

### Remarks:

This method is called when the interface is deleted.

## Class FPValue

See Also: [List of ParamType2 Choices](#), [Template Class Tab](#).

class FPValue

### Description:

This class is available in release 4.0 and later only.

This class contains a single value used as part of the Function Publishing system. It's capable of holding any of the FnPub supported types. This value is used as a parameter and as a return value.

### Data Members:

public:

#### **ParamType2 type;**

This data member identifies which type of value is stored in the union below.

The following union contains a single value, pointer to a single value, or a pointer to a single table (Tab<>) of values.

#### **union**

{

**int i;**

**float f;**

**int\* iptr;**

**float\* fptr;**

**Point3\* p;**

**TimeValue t;**

**TCHAR\* s;**

**TSTR\* tstr;**

**PBBitmap\* bm;**

**Mtl\* mtl;**

**Texmap\* tex;**

**INode\* n;**

**ReferenceTarget\* r;**

**Matrix3\* m;**

```
AngAxis* aa;  
Quat* q;  
Ray* ray;  
Point2* p2;  
BitArray* bits;  
ClassDesc* cd;  
Mesh* msh;  
Object* obj;  
Control* ctrl;  
Interval* intvl;  
POINT* pt;  
HWND hwnd;  
IObject* iobj;  
FPInterface* fpi;  
void* ptr;  
Color* clr;  
FPValue* fpv;  
Value* v;
```

```
// Tab<>s of above  
Tab<int>* i_tab;  
Tab<float>* f_tab;  
Tab<Point3*>* p_tab;  
Tab<TimeValue>* t_tab;  
Tab<TCHAR*>* s_tab;  
Tab<TSTR*>* tstr_tab;  
Tab<PBBitmap*>* bm_tab;  
Tab<Mtl*>* mtl_tab;  
Tab<Texmap*>* tex_tab;  
Tab<INode*>* n_tab;
```

```
Tab<ReferenceTarget*>* r_tab;  
Tab<Matrix3*>* m3_tab;  
Tab<AngAxis*>* aa_tab;  
Tab<Quat*>* q_tab;  
Tab<Ray*>* ray_tab;  
Tab<Point2*>* p2_tab;  
Tab<BitArray*>* bits_tab;  
Tab<ClassDesc*>* cd_tab;  
Tab<Mesh*>* msh_tab;  
Tab<Object*>* obj_tab;  
Tab<Control*>* ctrl_tab;  
Tab<Interval*>* intvl_tab;  
Tab<POINT*>* pt_tab;  
Tab<HWND*>* hwnd_tab;  
Tab<IObject*>* iobj_tab;  
Tab<FPInterface*>* fpi_tab;  
Tab<void*>* ptr_tab;  
Tab<Color*>* clr_tab;  
Tab<FPValue*>* fpv_tab;  
Tab<Value*>* v_tab;  
};
```

**Methods:**

public:

**Prototype:**

**FPValue();**

**Remarks:**

Constructor

**Default Implementation:**

**{ Init(); }**

**Prototype:**

**FPValue(FPValue& from);**

**Remarks:**

Constructor.

**Parameters:**

**FPValue& from**

A reference to a FPValue to copy from.

**Default Implementation:**

```
{ Init(); *this = from; }
```

**Prototype:**

**FPValue(int type, ...);**

**Remarks:**

Constructor

**Default Implementation:**

```
{va_list ap; va_start(ap, type); ap = Loadva(type, ap);  
va_end(ap);}
```

**Prototype:**

**~FPValue();**

**Remarks:**

Destructor.

**Default Implementation:**

```
{ Free(); }
```

**Prototype:**

**void Free();**

**Remarks:**

This method will free up all memory used by the class.

**Prototype:**

```
void Init();
```

**Remarks:**

This method will Initialize FPValue class.

**Default Implementation:**

```
{ type = (ParamType2)TYPE_INT; s = NULL; }
```

**Prototype:**

```
FPValue& operator=(FPValue& sv);
```

**Remarks:**

Assignment operator.

**Prototype:**

```
inline void Load(int type, ...);
```

**Remarks:**

This method will load the FPValue class with the provided data.

**Parameters:**

```
int type
```

The FPValue parameter type to load.

...

This method takes a variable number of arguments.

**Default Implementation:**

```
{va_list ap; va_start(ap, type); ap = Loadva(type, ap);  
va_end(ap);}
```

**Prototype:**

```
inline void LoadPtr(int type, ...);
```

**Remarks:**

This method will load the FPValue class with the provided data.

**Parameters:**

```
int type
```

The FPValue parameter type to load.

...

This method takes a variable number of arguments.

**Return Value:**

```
{ va_list ap; va_start(ap, type);  
  ap = Loadva(type, ap, true); va_end(ap); }
```

## Class FPParamOptions

See Also: [Class FPValue](#), [List of ParamType2 Choices](#), [Template Class Tab](#).

class FPParamOptions

### Description:

This class is available in release 4.0 and later only.

This class contains the optional parameters holding specific descriptor information.

### Data Members:

public:

**FPValue range\_low;**

The low range if specified.

**FPValue range\_high;**

The high range if specified.

**FPValidator\* validator;**

The validator if specified.

**FPValue keyarg\_default;**

The default if value is optional keyword arg

**BYTE pos\_param\_count;**

The count of positional params in event of keyword arg presence

### Methods:

public:

### Prototype:

**FPParamOptions();**

### Remarks:

Constructor.

## Class FPValidator

See Also: [Class InterfaceServer](#), [Class FPInterfaceDesc](#), [Class FPValue](#), [Class Interface\\_ID](#), [Function Publishing System](#).

```
class FPValidator : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

An interface descriptor may contain validation information for individual parameters, so that clients (such as MAXScript) can validate values given as parameters to FPInterface calls, prior to making the call. The validation information can be in the form of a range of values for int and float types, or more generally, a validator object that is called to validate a parameter value. The validation info is specified in the FPInterface descriptor in optional tagged entries following the parameters to be validated. The two possible tags are **f\_range** and **f\_validator**. An instance of this class is used when **f\_validator** is specified.

Here's an example from a possible mixin interface to Cylinder:

```
static FPInterfaceDesc cylfpi (  
CYL_INTERFACE, _T("cylMixin"), 0, &cylinderDesc, FP_MIXIN,  
...  
cyl_setRadius, _T("setRadius"), 0, TYPE_VOID, 0, 1,  
_T("radius"), 0, TYPE_FLOAT,  
f_range, 0.0, 10000.0,  
cyl_setDirection, _T("setDirection"), 0, TYPE_VOID, 0, 1,  
_T("vector"), 0, TYPE_POINT3,  
f_validator, &cylValidator,  
...  
end  
);
```

The "vector" parameter in the above example has a validator object specified. This must be a pointer to an instance of a class derived from the new class, **FPValidator**, defined in **I\MAXSDK\INCLUDE\iFnPub.h**. This is a

virtual base class, containing a single method, **Validate()**, that is called to validate a prospective value for a parameter. You would typically subclass **FPValidator** in your code and provide an implementation of **Validate()** to do the validation.

**Methods:**

public:

**Prototype:**

**virtual bool Validate(FPInterface\* fpi, FunctionID fid, int paramNum, FPValue& val, TSTR& msg)=0;**

**Remarks:**

This method is called to validate the value **val** passed for the given parameter in the function whose ID is passed in the specified interface passed. If there are many parameters to validate this way, developers can choose to provide a separate subclass for each parameter or a single subclass and switch on the parameter identification supplied.

**Parameters:**

**FPInterface\* fpi**

Points to the interface the function is a part of.

**FunctionID fid**

The ID of the function within the interface above.

**int paramNum**

Identifies which parameter within the function above to validate.

**FPValue& val**

The value to validate.

**TSTR& msg**

Update this string with an error message if needed. The user of the Validator can then display this string.

**Return Value:**

Returns true if the value was valid; false if invalid.

## Class ClassDesc2

See Also: [Class ClassDesc](#), [Class Animatable](#), [Class ParamBlockDesc2](#), [Class IParamMap2](#), [Class IObjParam](#), [Class ParamMap2UserDlgProc](#), [Class IAutoMParamDlg](#).

```
class ClassDesc2 : public ClassDesc
```

### Description:

This class is available in release 3.0 and later only.

A subclass of ClassDesc which you specialize to provide a class descriptor for plug-in classes that will use the ParamBlock2 system. It contains a table of ParamBlockDesc2s for all the parameter blocks used in the plug-in and a number of sets of methods including access to the block descriptors, auto user interface management, auto param block2 construction, and access to any automatically-maintained ParamMap2s.

### Methods:

public:

### Prototype:

```
ClassDesc2();
```

### Remarks:

Constructor. The master ParamDlg and Effect ParamDlg pointers are set to NULL.

### Prototype:

```
~ClassDesc2();
```

### Remarks:

Destructor.

### Prototype:

```
void ResetClassParams(BOOL fileReset);
```

### Remarks:

This method may be called to restore all sticky parameters to their default values.

**Parameters:****BOOL fileReset**

This parameter is not used.

**Prototype:****int NumParamBlockDescs();****Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

Returns the number of parameter block2 descriptors used by this plug-in class.

**Prototype:****ParamBlockDesc2\* GetParamBlockDesc(int i);****Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

Returns a pointer to the 'i-th' parameter block2 descriptor.

**Parameters:****int i**

The zero based index of the parameter block2 descriptor to return.

**Prototype:****ParamBlockDesc2\* GetParamBlockDescByID(BlockID id);****Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

Returns a pointer to the parameter block2 descriptor as specified by its **BlockID**.

Note: **typedef short BlockID;**

**Parameters:**

**BlockID id**

The permanent ID for the parameter block.

**Prototype:**

**ParamBlockDesc2\* GetParamBlockDescByName(TCHAR\* name);**

**Remarks:**

Returns a pointer to the parameter block2 descriptor as specified by the descriptor's internal name.

**Parameters:**

**TCHAR\* name**

The internal name of the parameter block descriptor.

**Prototype:**

**void AddParamBlockDesc(ParamBlockDesc2\* pbd);**

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

Adds a parameter block2 to the list of those maintained by this class descriptor.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block2 descriptor of the parameter block2 to add.

**Prototype:**

**void ClearParamBlockDescs();**

**Remarks:**

Implemented by the System.

Removes all the parameter block 2 descriptors maintained by this plug-in.

**Prototype:**

```
void BeginEditParams(IObjParam *ip, ReferenceMaker* obj,  
ULONG flags, Animatable *prev);
```

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

This method is called to handle the beginning of the automatic command panel user interface management provided by the param map 2 system. This method is called by the plug-in from its **Animatable::BeginEditParams()** method. The parameters passed to that method are simply passed along to this method.

**Parameters:**

**IObjParam \*ip**

The interface pointer passed to the plug-in.

**ReferenceMaker\* obj**

Points to the plug-in class calling this method.

**ULONG flags**

The flags passed along to the plug-in in **Animatable::BeginEditParams()**.

**Animatable \*prev**

The pointer passed to the plug-in in **Animatable::BeginEditParams()**.

**Prototype:**

```
void EndEditParams(IObjParam *ip, ReferenceMaker* obj,  
ULONG flags, Animatable *prev);
```

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an

implementation of the method used by plug-ins using the ParamBlock2 system.

This method is called to handle the ending of the automatic command panel user interface management provided by the param map 2 system. This method is called by the plug-in from its **Animatable::EndEditParams()** method. The parameters passed to that method are simply passed along to this method.

**Parameters:**

**IObjParam \*ip**

The interface pointer passed to the plug-in.

**ReferenceMaker\* obj**

Points to the plug-in class calling this method.

**ULONG flags**

The flags passed along to the plug-in in **Animatable::EndEditParams()**.

**Animatable \*prev**

The pointer passed to the plug-in in **Animatable::EndEditParams()**.

**Prototype:**

**void InvalidateUI();**

**Remarks:**

Implemented by the System.

This invalidates the entire UI for **every** parameter map of the plug-in.

**Prototype:**

**void InvalidateUI(ParamBlockDesc2\* pbd);**

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

This is called if the user interface parameters needs to be updated. This invalidates the entire UI managed by the param map whose description is passed.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block descriptor for the rollup.

**Prototype:**

```
void InvalidateUI(ParamBlockDesc2* pbd, ParamID id, int  
tabIndex=-1);
```

**Remarks:**

Implemented by the System.

This is called if a certain user interface parameter of the specified parameter map needs to be updated. The parameter ID of the control is passed. If the parameter is a Tab<> then the index into the table of the parameter is passed.

**Parameters:****ParamBlockDesc2\* pbd**

Points to the parameter block descriptor for the rollup.

**ParamID id**

The permanent parameter ID of the parameter.

**int tabIndex=-1**

If the parameter is a Tab<> then this is the zero based index into the table. The default value of -1 indicates it is not a table.

**Prototype:**

```
void MakeAutoParamBlocks(ReferenceMaker* owner);
```

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

This method is called to create the parameter blocks for the plug-in.

**Parameters:****ReferenceMaker\* owner**

Points to the plug-in class calling this method.

**Prototype:**

**int NumParamMaps();**

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

Returns the number of automatically-maintained parameter map2s.

**Prototype:**

**IParamMap2\* GetParamMap(ParamBlockDesc2\* pbd, MapID map\_id = 0);**

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

Returns a pointer to the parameter map2 as specified by the parameter block2 pointer passed.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block descriptor2 associated with this parameter map.

**MapID map\_id**

This parameter is available in release 4.0 and later only.

Specifies the ID of the map/rollout to get.

**Prototype:**

**IParamMap2\* GetParamMap(ParamBlockDesc2\* pbd);**

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

Returns a pointer to the parameter map2 as specified by the parameter block2

pointer passed.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block descriptor2 associated with this parameter map.

**Prototype:**

```
void SetUserDlgProc(ParamBlockDesc2* pbd,  
ParamMap2UserDlgProc* proc=NULL);
```

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

This method allows the developer to provide special handling for controls not processed automatically by the parameter map (or those that need additional processing). The developer provides a dialog proc to process the messages from the controls. This method is used to tell the parameter map that the developer defined method should be called. The given proc will be called after default processing is done.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **SetUserDlgProc()** with default map ID of 0.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block descriptor for the parameter map.

**ParamMap2UserDlgProc\* proc=NULL**

Points to the class derived from **ParamMap2UserDlgProc** which handles the controls.

**Prototype:**

```
void SetUserDlgProc(ParamBlockDesc2* pbd, MapID map_id,  
ParamMap2UserDlgProc* proc=NULL);
```

**Remarks:**

This method is available in release 4.0 and later only.

This overload of **SetUserDlgProc()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to set the user dialog proc for. See original function for the rest of the description.

**Prototype:**

**ParamMap2UserDlgProc\* GetUserDlgProc(ParamBlockDesc2\* pbd, MapID map\_id = 0);**

**Remarks:**

Implemented by the System.

This is a method of the base class ClassDesc. This class provides an implementation of the method used by plug-ins using the ParamBlock2 system.

Returns a pointer to the user dialog proc associated with the parameter map as specified by the parameter block descriptor2 pointer.

**Parameters:**

**ParamBlockDesc2\* pbd**

Points to the parameter block descriptor for the parameter map.

**MapID map\_id**

This parameter is available in release 4.0 and later only.

Specifies the ID of the map/rollout to get the user dialog proc for.

**Prototype:**

**IAutoMParamDlg\* CreateParamDlgs(HWND hwMtlEdit, IMtlParams \*imp, MtlBase\* obj);**

**Remarks:**

Implemented by the System.

This method creates and returns a pointer to the object which handles the automatic processing of the user interface in the materials editor. This method loops over all parameter blocks which specify **AUTO\_UI** and makes the AutoMParamDlgs for them. The first one becomes the master and the others are added to it.

**Parameters:**

**HWND hwMtlEdit**

The window handle of the materials editor.

**IMtlParams \*imp**

The interface pointer provided for calling methods in 3ds max.

**MtlBase\* obj**

Points to the plug-in class calling this method.

**Prototype:**

**IAutoMParamDlg\* CreateParamDlg(BlockID id, HWND hwMtlEdit, IMtlParams \*imp, ReferenceTarget\* obj, MapID map\_id = 0);**

**Remarks:**

Implemented by the System.

This method creates and returns a pointer to the object which handles the automatic processing of the user interface in the materials editor. This method makes an AutoMParamDlg for the specified parameter block.

**Parameters:**

**BlockID id**

The permanent ID of the parameter block.

**HWND hwMtlEdit**

The window handle of the materials editor.

**IMtlParams \*imp**

The interface pointer provided for calling methods in 3ds max.

**MtlBase\* obj**

Points to the plug-in class calling this method.

**MapID map\_id**

This parameter is available in release 4.0 and later only.

Specifies the ID of the map/rollout in the parameter block to create AutoMParamDlg for.

**Prototype:**

**IAutoEParamDlg\* CreateParamDialogs(IRendParams \*ip, SpecialFX\* obj);**

**Remarks:**

Implemented by the System.

This method creates and returns a pointer to the object which handles the automatic processing of the user interface in the rendering effects dialog. This method loops over all parameter blocks which specify **AUTO\_UI** and makes the AutoMParamDlgs for them. The first one becomes the master and the others are added to it.

**Parameters:**

**IRendParams \*ip**

The interface pointer provided for calling methods in 3ds max.

**SpecialFX\* obj**

Points to the plug-in class calling this method. See [Class SpecialFX](#).

**Prototype:**

**IAutoEParamDlg\* CreateParamDialog(BlockID id, IRendParams \*ip, SpecialFX\* obj, MapID mapID=0);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method creates and returns a pointer to the object which handles the automatic processing of the user interface in the rendering effects dialog. This method makes an AutoEParamDlg for the specified parameter block.

**Parameters:**

**BlockID id**

The permanent ID of the parameter block.

**IRendParams \*ip**

The interface pointer provided for calling methods in 3ds max.

**SpecialFX\* obj**

Points to the plug-in class calling this method. See [Class SpecialFX](#).

**MapID map\_id**

Specifies the ID of the map/rollout in the parameter block to create AutoEParamDlg for.

**Prototype:**

**void MasterDlgDeleted(IAutoMParamDlg\* dlg);**

**Remarks:**

Implemented by the System.

This method is called when an AutoMParamDlg is deleted.

**Parameters:**

**IAutoMParamDlg\* dlg**

Pointer to the object which handles the automatic processing of the user interface in the materials editor.

**Prototype:**

**void MasterDlgDeleted(IAutoEParamDlg\* dlg);**

**Remarks:**

Implemented by the System.

This method is called when an AutoEParamDlg is deleted.

**Parameters:**

**IAutoEParamDlg\* dlg**

Pointer to the object which handles the automatic processing of the user interface in the rendering effects dialog.

**Prototype:**

**IAutoMParamDlg\* GetMParamDlg();**

**Remarks:**

Implemented by the System.

Returns the master dialog processing routine for the materials editor plug-in.

**Prototype:**

**IAutoEParamDlg\* GetEParamDlg();**

**Remarks:**

Implemented by the System.

Returns the master dialog processing routine for the rendering effects plug-in.

**Prototype:**

**void RestoreRolloutState();**

**Remarks:**

This method may be called to restore any saved rollout state (open/closed condition and scrolling position) for any parameter map maintained by the plug-in.

**Prototype:**

**ParamID LastNotifyParamID(ReferenceMaker\* owner,  
IParamBlock2\*& pb);**

**Remarks:**

This method scans all the parameter blocks in the owner and returns the ParamID and parameter block making the most recent change notification.

**Parameters:**

**ReferenceMaker\* owner**

The owner of the parameter blocks.

**IParamBlock2\*& pb**

The parameter block which made the most recent notification.

**Return Value:**

The parameter ID of the parameter which made the most recent notification.

**Prototype:**

**void Reset(ReferenceMaker\* owner, BOOL updateUI = TRUE,  
BOOL callSetHandlers = TRUE);**

**Remarks:**

This method may be called to reset all the parameters of all know parameter blocks to their default values and optionally update the user interface.

**Parameters:**

**ReferenceMaker\* owner**

The owner of this ClassDesc2.

**BOOL updateUI = TRUE**

If TRUE to user inteface is updated. If FALSE it's not.

**BOOL callSetHandlers = TRUE**

TRUE to call **PBAccessor::Set()** for all the parameters; otherwise FALSE.

**Prototype:**

**void GetValidity(ReferenceMaker\* owner, TimeValue t, Interval &valid);**

**Remarks:**

This method updates the validity interval passed with the cumulative validity interval of all the owner's parameter blocks.

**Parameters:**

**ReferenceMaker\* owner**

The owner of this ClassDesc2.

**TimeValue t**

The time about which to compute the interval.

**Interval &valid**

The interval to update.

**The following functions are not part of this class but are available for use.**

**Function:**

**IParamBlock2 \*CreateParameterBlock2(ParamBlockDesc2 \*pdesc, ReferenceMaker\* iowner);**

**Remarks:**

This method is used to create a parameter block2.

**Parameters:**

**ParamBlockDesc2 \*pdesc**

This is an array of parameter block descriptors.

**ReferenceMaker\* iowner**

Points to the owner of the parameter block.

**Return Value:**

A pointer to the created parameter block. On error NULL is returned.

**Function:**

**IParamBlock2\* UpdateParameterBlock2(ParamBlockDescID \*pdescOld, int oldCount, IParamBlock \*oldPB, ParamBlockDesc2\* pdescNew, IParamBlock2\* newPB=NULL);**

**Remarks:**

This function creates a new ParamBlock2, based on an existing ParamBlock of an earlier version. The new parameter block inherits any parameters from the old parameter block whose parameter IDs match. This may also be used to partially update an existing ParamBlock2.

**Parameters:**

**ParamBlockDescID \*pdescOld**

The array of parameter block descriptors which describes each parameter in the old parameter block.

**int oldCount**

The number of elements in the array above.

**IParamBlock \*oldPB**

The old parameter block.

**ParamBlockDesc2\* pdescNew**

Points to the new parameter block 2 descriptor.

**IParamBlock2\* newPB=NULL**

Points to an existing IParamBlock2 indicating that this paramblock should be filled in from the old ParamBlock, rather than creating a new one.

**Return Value:**

The new parameter block2.

**Function:**

**void SetPB2MacroRecorderInterface(MacroRecorder\* mri);**

**Remarks:**

This function for internal use only.

## Class ModContext

See Also: [Class LocalModData](#).

```
class ModContext : public BaseInterfaceServer
```

### Description:

The ModContext stores information about the space the modifier was applied in, and allows the modifier to store data it needs for its operation. All methods are implemented by the system.

### Data Members:

public:

#### **Matrix3 \*tm;**

This matrix represents the space the modifier was applied in. The modifier plug-in uses this matrix when it deforms an object. The plug-in first transforms the points with this matrix. Next it applies its own deformation. Then it transforms the points back through the inverse of this transformation matrix.

#### **Box3 \*box;**

The Bounding Box of the Deformation. This represents the scale of the modifier. For a single object it is the bounding box of the object. If the modifier is being applied to a sub-object selection it represents the bounding box of the sub-object selection. If the modifier is being applied to a selection set of objects, then this is the bounding box of the entire selection set. For a selection set of objects the bounding box is constant. In the case of a single object, the bounding box is not constant.

#### **LocalModData \*localData;**

A pointer to an instance of a class derived from the LocalModData class. This is the part of the ModContext that the plug-in developer controls. It is the place where a modifier may store application specific data.

### Methods:

### Prototype:

```
ModContext();
```

### Remarks:

Constructor. The transformation matrix, bounding box, and local data pointer

are initialized to NULL.

**Prototype:**

**~ModContext();**

**Remarks:**

Destructor. The tm, bounding box and local data are freed.

**Prototype:**

**ModContext(const ModContext& mc);**

**Remarks:**

Constructor. The tm, bounding box and local data are initialized to those of the specified ModContext.

**Parameters:**

**const ModContext& mc**

The ModContext to copy.

**Prototype:**

**ModContext(Matrix3 \*tm, Box3 \*box, LocalModData \*localData);**

**Remarks:**

Constructor. The tm, bounding box, and local data are initialized to those specified.

**Parameters:**

**Matrix3 \*tm**

The transform matrix to copy.

**Box3 \*box**

The bounding box to copy.

**LocalModData \*localData**

The local data that will be cloned.

## Class ObjectState

See Also: [Class Object](#), [Class Matrix3](#).

class ObjectState

### Description:

The ObjectState is the structure that flows up the geometry pipeline. It contains a matrix, a material index, some flags for channels, and a pointer to the object in the pipeline.

### Data Members:

public:

**Object \*obj;**

This is a pointer to the object in the pipeline. The validity interval of the object can be retrieved using **obj->ObjectValidity()**.

### Methods:

#### Prototype:

**ObjectState();**

#### Remarks:

Constructor. The object pointer is initialized to NULL.

#### Prototype:

**ObjectState(Object \*ob);**

#### Remarks:

Constructor. The object pointer is set to the object passed. The tm pointer is set to NULL and the tm and mtl validity intervals are set to **FOREVER**.

#### Parameters:

**Object \*ob**

The object to initialize the object pointer to.

#### Prototype:

**ObjectState(const ObjectState& os);**

#### Remarks:

Constructor. The object state is initialized to the object state passed.

**Parameters:**

**const ObjectState& os**

The object state to initialized to **os**.

**Prototype:**

**~ObjectState();**

**Remarks:**

Destructor. If the **tm** exists, it is deleted.

**Prototype:**

**void OSSetFlag(ulong f);**

**Remarks:**

Call this method to update the object state flags.

**Parameters:**

**ulong f**

The flags to set. The specified flags are ORed into the current state of the flags.

**Prototype:**

**void OSClearFlag(ulong f);**

**Remarks:**

Call this method to clear the specified object state flags.

**Parameters:**

**ulong f**

The flags to clear.

**Prototype:**

**ulong OSTestFlag(ulong f) const;**

**Remarks:**

Call this method to test the specified flags.

**Parameters:**

**ulong f**

The flags to test.

**Return Value:**

Nonzero if the specified flags are all set; otherwise 0.

**Prototype:**

**void OSCopyFlag(ulong f, const ObjectState& fromos);**

**Remarks:**

Copies the specified flag settings from the specified object state to this object state.

**Parameters:**

**ulong f**

The flags to copy.

**const ObjectState& fromos**

The source object state.

**Prototype:**

**Interval tmValid() const**

**Remarks:**

Returns the validity interval of the object state's transformation matrix.

**Prototype:**

**Interval mtlValid() const;**

**Remarks:**

Returns the validity interval of the object state's material.

**Prototype:**

**Interval Validity(TimeValue t) const;**

**Remarks:**

Returns the validity interval of the object state. If the object is not defined, this interval is NEVER. Otherwise it is the intersection of the tm validity interval,

the mtl validity interval and the interval returned from obj->ObjectValidity(t).

**Parameters:**

**TimeValue t**

Specifies the time to retrieve the validity interval.

**Return Value:**

The validity interval of the object state.

**Prototype:**

**int TMIsIdentity() const;**

**Remarks:**

Returns nonzero if the object state's transformation matrix is the identity matrix; otherwise zero.

**Prototype:**

**void SetTM(Matrix3\* mat, Interval iv);**

**Remarks:**

Sets the object state's transformation matrix to the specified Matrix3 and its validity interval to the interval passed. If the specified matrix is NULL, a new Matrix3 is allocated and is initialized to the identity matrix.

**Parameters:**

**Matrix3\* mat**

Specifies the matrix to set.

**Interval iv**

Specifies the validity interval to set.

**Prototype:**

**Matrix3\* GetTM() const;**

**Remarks:**

Returns the object state's transformation matrix.

**Prototype:**

**void SetIdentityTM();**

**Remarks:**

Sets the object state tm to the identity transformation matrix.

**Prototype:**

```
void ApplyTM(Matrix3* mat, Interval iv);
```

**Remarks:**

Applies the specified matrix to the object state tm. The object state tm is multiplied by the specified matrix. The specified interval is intersected with the object state tm validity interval.

**Parameters:**

**Matrix3\* mat**

The matrix to apply.

**Interval iv**

The interval to intersect with the object state's tm validity interval.

**Prototype:**

```
void CopyTM(const ObjectState &fromos);
```

**Remarks:**

Copies the object state tm (and its validity interval) from the specified object state's tm.

**Parameters:**

**const ObjectState &fromos**

The object state whose tm is to be copied.

**Prototype:**

```
void CopyMtl(const ObjectState &fromos);
```

**Remarks:**

Copies the object state material (and its validity interval) from the specified object state's material.

**Parameters:**

**const ObjectState &fromos**

The object state whose material is to be copied.

**Prototype:**

```
void Invalidate(ChannelMask channels, BOOL  
checkLock=FALSE);
```

**Remarks:**

Invalidates the specified channels of the object state's object.

**Parameters:**

**ChannelMask channels**

The channels of the object to invalidate.

**BOOL checkLock=FALSE**

If **checkLock** is TRUE and **OBJ\_CHANNELS** is one of the specified channels, the object the object state points to is not deleted if it is locked; otherwise it is deleted.

**Prototype:**

```
void DeleteObj(BOOL checkLock=FALSE);
```

**Remarks:**

Deletes the object state's object.

**Parameters:**

**BOOL checkLock=FALSE**

If **checkLock** is TRUE, the object the object state points to is not deleted if it is locked; otherwise it is always deleted.

**Operators:****Prototype:**

```
ObjectState& operator=(const ObjectState& os);
```

**Remarks:**

Assignment operator. The object pointer, flags, transformation matrix (and its validity interval), and material (and its validity interval) are copied from the specified object state.

## Class LocalModData

See Also: [Class ModContext](#).

```
class LocalModData : public InterfaceServer
```

### Description:

This class allows a modifier to store application specific data. A reference to a pointer to an instance of this class is passed in to **ModifyObject()** as part of the **ModContext**. The value of the pointer starts out as NULL, but the modifier can set it to point at an actual instance of a derived class. When the mod app is deleted, if the pointer is not NULL, the **LocalModData** will be deleted - the virtual destructor allows this to work.

### Methods:

#### Prototype:

```
virtual ~LocalModData()
```

#### Remarks:

Implemented by the Plug-In.

A plug-in using local data should implement this method to free its local data.

#### Prototype:

```
virtual LocalModData *Clone()=0;
```

#### Remarks:

Implemented by the Plug-In.

This method is called to allow a plug-in to copy its local data. It is called when the system is copying a **ModContext**.

#### Return Value:

The plug-in should return a pointer to a new instance of its LocalModData.

## List of Channel Bits

See Also: [The Geometry Pipeline System](#), [Object Modification](#), [Class Mesh](#), [List of Object Channels](#).

The pipeline is divided into the following channels:

### **GEOM\_CHANNEL**

The vertices of the object. Most modifiers only alter this channel.

### **TOPO\_CHANNEL**

The topology channel, i.e. the face or polygon structures. Smoothing groups and materials are also part of this channel. Edge visibility is also part of this channels since it is an attribute of the face structure.

### **TEXMAP\_CHANNEL**

The texture vertices and procedural mappings.

### **MTL\_CHANNEL**

This is no longer used. Materials are rolled into the Face data structure and are part of the topology channel.

### **SELECT\_CHANNEL**

The sub-object selection channel. An object's selection flows down the pipeline. What the selection is actually comprised of is up to the specific object type. For example, TriObjects have bits for face, edge and vertex selection. This channel is the actual **BitArray** used (like **selLevel** of the **Mesh** class).

### **SUBSEL\_TYPE\_CHANNEL**

This is the current level of selection. Every object that flows down the pipeline is at a certain level that corresponds to the Sub-Object drop down in the 3ds max user interface. This channel indicates which level the object is at. This is also specific to the object type. There are 32 bits to represent the level of selection. When all the bits are 0, the object is at object level selection.

### **DISP\_ATTRIB\_CHANNEL**

These are miscellaneous bits controlling the item's display. These bits are specific to the type of object. For the Mesh object these are the surface normal scale, display of surface normals, edge visibility and display flags.

### **EXTENSION\_CHANNEL**

This channel is available in release 4.0 and later only.

This is the channel used by extension channel objects.

## **VERTCOLOR\_CHANNEL**

This is the color per vertex channel. This is also used for the second texture mapping channel.

## **GFX\_DATA\_CHANNEL**

This channel is used internally by 3ds max for stripping. Plug-In developers don't need to specify this channel as being changed or used in their plug-ins.

## **TM\_CHANNEL**

This is the ObjectState TM that flows down the pipeline. This TM may be modified by modifiers.

## **GLOBMTL\_CHANNEL**

This is no longer used. Materials are rolled into the Face data structure and are part of the topology channel.

The following **#defines** may be used to specify groups of channels:

```
#define OBJ_CHANNELS (TOPO_CHANNEL |  
GEOM_CHANNEL | SELECT_CHANNEL |  
TEXMAP_CHANNEL | MTL_CHANNEL |  
SUBSEL_TYPE_CHANNEL | DISP_ATTRIB_CHANNEL |  
VERTCOLOR_CHANNEL | GFX_DATA_CHANNEL |  
DISP_APPROX_CHANNEL | EXTENSION_CHANNEL)  
#define ALL_CHANNELS  
(OBJ_CHANNELS|TM_CHANNEL|GLOBMTL_CHANNEL)
```

Note: Some of the sample code specifies these channels as **PART\_\*** as opposed to **\*\_CHANNEL**. For example, **PART\_GEOM|PART\_TOPO** instead of **GEOM\_CHANNEL|TOPO\_CHANNEL**. The proper usage is the **\*\_CHANNEL** version.

## List of Viewport Drawing Color Indices

See Also: Advanced Topics section [Getting and Setting User Preferences](#).

The following values may be used to indicate which color is returned by the various APIs for getting and setting user interface drawing colors.

**COLOR\_SELECTION**

**COLOR\_SUBSELECTION**

**COLOR\_FREEZE**

**COLOR\_GRID**

**COLOR\_GRID\_INTENS**

**COLOR\_SF\_LIVE**

**COLOR\_SF\_ACTION**

**COLOR\_SF\_TITLE**

**COLOR\_VP\_LABELS**

**COLOR\_VP\_INACTIVE**

**COLOR\_ARCBALL**

**COLOR\_ARCBALL\_HILITE**

**COLOR\_ANIM\_BUTTON**

**COLOR\_SEL\_BOXES**

**COLOR\_LINK\_LINES**

**COLOR\_TRAJECTORY**

**COLOR\_ACTIVE\_AXIS**

**COLOR\_INACTIVE\_AXIS**

**COLOR\_SPACE\_WARPS**

**COLOR\_DUMMY\_OBJ**

**COLOR\_POINT\_OBJ**

**COLOR\_POINT\_AXES**

**COLOR\_TAPE\_OBJ**

**COLOR\_BONES**

**COLOR\_GIZMOS**

**COLOR\_SEL\_GIZMOS**  
**COLOR\_SPLINE\_VECS**  
**COLOR\_SPLINE\_HANDLES**  
**COLOR\_PATCH\_LATTICE**  
**COLOR\_PARTICLE\_EM**  
**COLOR\_CAMERA\_OBJ**  
**COLOR\_CAMERA\_CONE**  
**COLOR\_CAMERA\_HORIZ**  
**COLOR\_NEAR\_RANGE**  
**COLOR\_FAR\_RANGE**  
**COLOR\_LIGHT\_OBJ**  
**COLOR\_TARGET\_LINE**  
**COLOR\_HOTSPOT**  
**COLOR\_FALLOFF**  
**COLOR\_START\_RANGE**  
**COLOR\_END\_RANGE**  
**COLOR\_VIEWPORT\_BKG**  
**COLOR\_TRAJ\_TICS\_1**  
**COLOR\_TRAJ\_TICS\_2**  
**COLOR\_TRAJ\_TICS\_3**  
**COLOR\_GHOST\_BEFORE**  
**COLOR\_GHOST\_AFTER**  
**COLOR\_12FIELD\_GRID**  
**COLOR\_START\_RANGE1**  
**COLOR\_END\_RANGE1**  
**COLOR\_CAMERA\_CLIP**  
**COLOR\_NURBS\_CV**  
**COLOR\_NURBS\_LATTICE**  
**COLOR\_NURBS\_CP**  
**COLOR\_NURBS\_FP**

**COLOR\_NURBS\_DEP**  
**COLOR\_END\_EFFECTOR**  
**COLOR\_END\_EFFECTOR\_STRING**  
**COLOR\_JOINT\_LIMIT\_SEL**  
**COLOR\_JOINT\_LIMIT\_UNSEL**  
**COLOR\_IK\_TERMINATOR**  
**COLOR\_SF\_USER**

The following indicies are available in R3 and later only.

**COLOR\_VERT\_TICKS**  
**COLOR\_XRAY**  
**COLOR\_GROUP\_OBJ**  
**COLOR\_MANIPULATOR\_X**  
**COLOR\_MANIPULATOR\_Y**  
**COLOR\_MANIPULATOR\_Z**  
**COLOR\_MANIPULATOR\_ACTIVE**  
**COLOR\_VPT\_CLIPPING**  
**COLOR\_DECAY\_RADIUS**  
**COLOR\_VERT\_NUMBERS**  
**COLOR\_CROSSHAIR\_CURSOR**  
**COLOR\_SV\_WINBK**  
**COLOR\_SV\_NODEBK**  
**COLOR\_SV\_SELNODEBK**  
**COLOR\_SV\_NODE\_HIGHLIGHT**  
**COLOR\_SV\_MATERIAL\_HIGHLIGHT**  
**COLOR\_SV\_MODIFIER\_HIGHLIGHT**  
**COLOR\_SV\_PLUGIN\_HIGHLIGHT**  
**COLOR\_SV\_SUBANIM\_LINE**  
**COLOR\_SV\_CHILD\_LINE**  
**COLOR\_SV\_FRAME**  
**COLOR\_SV\_SELTEXT**

**COLOR\_SV\_TEXT**  
**COLOR\_UNSEL\_TAB**  
**COLOR\_ATMOS\_APPARATUS**  
**COLOR\_SUBSELECTION\_HARD**  
**COLOR\_SUBSELECTION\_MEDIUM**  
**COLOR\_SUBSELECTION\_SOFT**  
**COLOR\_SV\_UNFOLD\_BUTTON**  
**COLOR\_SV\_GEOMOBJECT\_BK**  
**COLOR\_SV\_LIGHT\_BK**  
**COLOR\_SV\_CAMERA\_BK**  
**COLOR\_SV\_SHAPE\_BK**  
**COLOR\_SV\_HELPER\_BK**  
**COLOR\_SV\_SYSTEM\_BK**  
**COLOR\_SV\_CONTROLLER\_BK**  
**COLOR\_SV\_MODIFIER\_BK**  
**COLOR\_SV\_MATERIAL\_BK**

# Class HitRecord

See Also: [Class HitLog](#), [Class HitData](#).

class HitRecord

## Description:

This class provides a data structure used during sub-object hit-testing.

## Friend Class:

class HitLog

## Data Members:

public:

**Inode \*nodeRef;**

Points the node that was hit.

**ModContext \*modContext;**

Points to the ModContext of the modifier.

**DWORD distance;**

The 'distance' of the hit. To classify as a hit, the sub-object component must be within some threshold distance of the mouse. This distance is recorded in the hit record so that the closest of all the hits can be identified. What the distance actually represents depends on the rendering level of the viewport. For wireframe modes, it refers to the distance in the screen XY plane from the mouse to the sub-object component. In a shaded mode, it refers to the Z depth of the sub-object component. In both cases, smaller values indicate that the sub-object component is 'closer' to the mouse cursor.

**ulong hitInfo;**

A general unsigned long value. Most modifiers will just need this to identify the sub-object element. The edit mesh modifier uses the value to store the index of the vertex or face that was hit for example.

**HitData \*hitData;**

In case 4 bytes is not enough space to identify the sub-object element, this pointer is available. To use this, a developer would define a class derived from HitData that would contain the necessary data. The HitData class has one member function, a virtual destructor, so the derived class can be properly deleted when the HitRecord instance is deleted.

**Methods:**

**Prototype:**

**HitRecord();**

**Remarks:**

Constructor. The following initialization is performed: **next = NULL;**  
**modContext = NULL; distance = 0; hitInfo = 0; hitData = NULL;**

**Prototype:**

**HitRecord(INode \*nr, ModContext \*mc, DWORD d, long inf,  
HitData \*hitdat)**

**Remarks:**

Constructor. The data members are initialized to the data passed.

**Prototype:**

**HitRecord \*Next();**

**Remarks:**

Implemented by the System.

Each **HitRecord** maintains a pointer to another **HitRecord**. This method returns the next hit record.

**Prototype:**

**~HitRecord();**

**Remarks:**

Implemented by the System.

Destructor. If **HitData** has been allocated, it is deleted as well.

# Class HitRegion

See Also: [Data Types](#).

class HitRegion

## Description:

This class describes the properties of a region used for built-in hit testing of items in the interactive renderer.

## Data Members:

public:

### int type;

The region type. One of the following values:

#### **POINT\_RGN**

A single point.

#### **RECT\_RGN**

A rectangular region.

#### **CIRCLE\_RGN**

A circular region.

#### **FENCE\_RGN**

An arbitrary multi-point polygon region.

### int crossing;

If nonzero, elements that are contained within **or cross** the region boundary are hit. If zero, only those elements **entirely within** the boundary are hit. This is not used for point hit testing.

### int epsilon;

Specifies the distance in pixels outside the pick point within which elements may be and still be hit. This is not used for rect, circle, or fence hit testing.

### union {

**POINT** pt;

**RECT** rect;

**CIRCLE** circle;

**POINT** \*pts;

};

The storage for the region.

## Class HitLog

See Also: [Class HitRecord](#), [Class HitData](#).

class HitLog

### Description:

This class provides a data structure for keeping a log of hits during sub-object hit-testing. It provides a list of **HitRecords** that may be added to and cleared. A developer may also request the 'closest' hit record in the list.

### Methods:

#### Prototype:

**HitLog();**

#### Remarks:

Constructor. The list of HitRecords is set to NULL.

#### Prototype:

**~HitLog();**

#### Remarks:

Destructor. Clears the hit log.

#### Prototype:

**void Clear();**

#### Remarks:

Clears the log of hits by deleting the list of HitRecords.

#### Prototype:

**HitRecord\* First();**

#### Remarks:

Implemented by the System.

Returns the first **HitRecord**.

#### Prototype:

## **HitRecord\* ClosestHit();**

### **Remarks:**

Implemented by the System.

Returns the **HitRecord** that was 'closest' to the mouse position when hit testing was performed. This is the **HitRecord** with the minimum **distance**.

### **Prototype:**

```
void LogHit(INode *nr, ModContext *mc,  
            DWORD dist, ulong info, HitData *hitdat = NULL);
```

### **Remarks:**

Implemented by the System.

This method is called to log a hit. It creates a new **HitRecord** object using the data passed and adds it to the hit log.

### **Parameters:**

#### **INode \*nr**

The node that was hit.

#### **ModContext \*mc**

The ModContext of the modifier.

#### **DWORD dist**

The 'distance' of the hit. What the distance actually represents depends on the rendering level of the viewport. For wireframe modes, it refers to the distance in the screen XY plane from the mouse to the sub-object component. In a shaded mode, it refers to the Z depth of the sub-object component. In both cases, smaller values indicate that the sub-object component is 'closer' to the mouse cursor.

#### **ulong info**

Identifies the sub-object component that was hit.

#### **HitData \*hitdat = NULL**

If the info field is insufficient to indicate the sub-object component that was hit, pass an instance of the **HitData** class that contains the needed information.

# Class Light

See Also: [Data Types](#).

```
class Light : public BaseInterfaceServer
```

## Description:

This class describes the lights used in the interactive renderer. All methods of this class are implemented by the system.

## Methods:

## Prototype:

```
Light();
```

## Remarks:

Class constructor. Sets the default values of the light.

```
type = OMNI_LGT;  
attenType = NO_ATTEN;  
atten = (float)0.0;  
intensity = (float)1.0;  
angle = (float)30.0;  
color[0] = (float)1.0;  
color[1] = (float)1.0;  
color[2] = (float)1.0;
```

## Data Members:

public:

```
LightType type;
```

One of the following values:

```
OMNI_LGT
```

Omni-directional.

```
SPOT_LGT
```

Spot light.

```
DIRECT_LGT
```

Directional light.

```
AMBIENT_LGT
```

Ambient light - global illumination.

**Point3 color;**

The color of the light. Individual values are from 0.0 to 1.0 with 1.0 as white.

**Attenuation attenType;**

Attenuation is not currently implemented. A developer should pass **NO\_ATTEN**.

**float atten;**

Note: Attenuation is not currently implemented.

Light attenuation factor.

**float intensity;**

Light multiplier factor.

**float angle;**

Angle of cone for spot and cone lights in degrees.

**int shape;**

One of the following values:

**GW\_SHAPE\_RECT** - Rectangular spotlights.

**GW\_SHAPE\_CIRCULAR** - Circular spotlights.

**float aspect;**

The aspect ratio of the light.

**int overshoot;**

Nonzero indicates the light supports overshoot; otherwise 0.

**BOOL affectDiffuse;**

This data member is available in release 2.0 and later only.

This defaults to TRUE, but if the user set it to FALSE in the light modifier panel, then the

light is not supposed to illuminate the diffuse component of an object's material.

**BOOL affectSpecular;**

This data member is available in release 2.0 and later only.

This defaults to TRUE, but if the user set it to FALSE in the light modifier panel, then the

light is not supposed to illuminate the specular component of an object's material.

# Class Interval

See Also: Advanced Topics sections on [Intervals](#) and [Time](#).

class Interval

## Description:

An Interval is a class that represents a length of time. It has two private data members, start and end that are each TimeValues. A TimeValue is a single instant in time. For more explanation see the Advanced Topics section on [Intervals](#). All the methods of this class are implemented by the system.

Definitions:

```
#define FOREVER Interval(TIME_NegInfinity,  
TIME_PosInfinity)  
#define NEVER Interval(TIME_NegInfinity, TIME_NegInfinity)
```

## Methods:

### Prototype:

```
Interval(TimeValue s, TimeValue e )
```

### Remarks:

Constructor that assigns both the start and end times of the interval. If TimeValue **e** is less than TimeValue **s** the values are swapped before they are assigned.

### Parameters:

**TimeValue s**

Specifies the start time.

**TimeValue e**

Specifies the end time.

### Prototype:

```
Interval()
```

### Remarks:

Constructor that returns an **EMPTY** interval, i.e. having a start and end time equal to **TIME\_NegInfinity**

**Prototype:****int InInterval(const TimeValue t);****Remarks:**

Return Nonzero if the TimeValue passed is greater than or equal to the start value and less than or equal to the end value and not equal to **TIME\_NegInfinity**. Returns 0 otherwise.

**Parameters:****const TimeValue t****Return Value:**

Nonzero if the TimeValue passed is greater than or equal to the start value and less than or equal to the end value and not equal to **TIME\_NegInfinity**; otherwise 0.

**Prototype:****int InInterval(const Interval interval)****Remarks:**

Returns nonzero if the interval passed is contained within the interval; otherwise 0.

```
return InInterval( interval.Start() ) && InInterval( interval.End() );
```

**Parameters:****const Interval interval**

The interval to check.

**Return Value:**

Returns nonzero if the interval passed is contained within the interval; otherwise 0.

**Prototype:****int Empty()****Remarks:**

Returns 1 if the interval is **EMPTY**, i.e. has a start and end time equal to **TIME\_NegInfinity**. Returns 0 otherwise.

**Prototype:**

**void Set(TimeValue s, TimeValue e)**

**Remarks:**

Sets the start and end times for the interval.

**Parameters:**

**TimeValue s**

Start time for the interval.

**TimeValue e**

End time for the interval.

**Prototype:**

**void SetStart(TimeValue s)**

**Remarks:**

Sets the start value only.

**Parameters:**

**TimeValue s**

Start time for the interval.

**Prototype:**

**void SetEnd(TimeValue e)**

**Remarks:**

Sets the end value only.

**Parameters:**

**TimeValue e**

End time for the interval.

**Prototype:**

**void SetEmpty()**

**Remarks:**

Sets the interval to be **EMPTY**, i.e. having a start and end time equal to **TIME\_NegInfinity**.

**Prototype:**

**void SetInfinite()**

**Remarks:**

Sets the interval to be **FOREVER**, i.e. have a start time equal **TIME\_NegInfinity** and end time equal to **TIME\_PosInfinity**.

**Prototype:**

**void SetInstant(const TimeValue t)**

**Remarks:**

Sets both the start and end times to the time passed.

**Prototype:**

**TimeValue Start()**

**Remarks:**

Returns the start time of the interval.

**Prototype:**

**TimeValue End()**

**Remarks:**

Returns the end time of the interval.

**Prototype:**

**TimeValue Duration()**

**Remarks:**

Implemented by the System.

Returns the duration of the interval (end points included).

**Operators:**

**Prototype:**

**int operator==(const Interval& i)**

**Remarks:**

Checks for equality between two Intervals.

**Return Value:**

Nonzero if the intervals are equal; otherwise 0.

**Prototype:**

**Interval operator&(const Interval i) const;**

**Remarks:**

Intersects **Interval** and i. The interval will have a start time of the greater of the two interval start times, and an end value which is the lesser of the two end values. If the end time is less than the start time, both the start and end times are set to **TIME\_NegInfinity**.

**Return Value:**

An Interval that is the intersection of the intervals.

**Prototype:**

**Interval& operator&=(const Interval i)**

**Remarks:**

This updates the invoking interval so it will have a start time of the greater of the two interval start times, and an end value which is the lesser of the two end values.

**return (\*this = (\*this&i));**

**Prototype:**

**Interval& operator+=(const TimeValue t)**

**Remarks:**

Expands the **Interval** to include the TimeValue.

**if (t<start) start=t; if (t>end) end=t; return \*this;**

## Class IKChainActions

See Also: : [Class IKCmdOps](#), [Class IIKChainControl](#)

class IKChainActions : public FPStaticInterface

### Description:

**This class is only available in release 5 or later.**

The program interface to actions on IK chain nodes, which **IKCmdOps::CreateIKChain()** returns. An macro is defined to obtain a pointer to the interface:

```
IKChainActions* ikchainAction =  
GET_IKCHAIN_FP_INTERFACE;
```

It should be a valid pointer provided the IK system is successfully loaded.

The methods are mainly designed to connect to the UI and script. However, they can be called programmatically as well. The action methods, those that return **FPStatus**, assume that a unique IK chain node is currently selected and apply action to it. Before an action method is called, **IsSnapEnabled()** must be called to test whether a unique IK chain node is being selected. Such as:

```
if (ikchainAction->IsSnapEnabled())  
    ikchainAction->SetPreferredAngles();
```

### Methods:

#### Prototype:

```
virtual FPStatus SetPreferredAngles()
```

#### Remarks:

It sets the preferred angles of all bones on the chain to the corresponding joint

angles currently in force, which can be FK or IK depending the state of Enabled.

**Prototype:**

**virtual FPStatus IKSnapAction()**

**Remarks:**

Applies IK Snapping, which sets IK goal and other parameters, such as the swivel angle, according to the current state of the bone chain. Moreover, it invokes **SetPreferredAngles()**.

**Prototype:**

**virtual FPStatus FKSnapAction()**

**Remarks:**

Applies FK Snapping, which assigns the currently active joint angles to the corresponding FK angles. Moreover, it invokes **SetPreferredAngles()**.

**Prototype:**

**virtual FPStatus SnapAction()**

**Remarks:**

It applies **IKSnapAction()** if IK Enabled is currently off, and **FKSnapAction()** if IK Enabled is on.

**Prototype:**

**virtual FPStatus ToggleEnabled()**

**Remarks:**

Toggles IK Enabled state. If the AutoSnap parameter of the IK chain is true, it will invoke **SnapAction()** before toggling the Enabled state.

**Prototype:**

**virtual FPStatus AssumePreferredAngles()**

**Remarks:**

It assigns the preferred angles to the corresponding FK angles and then turn off IK Enabled.

## Class IKSolver

See Also: [Class BaseInterfaceServer](#), [Class ZeroPlaneMap](#), [Class LinkChain](#), [Inverse Kinematics](#)

```
class IKSolver : public BaseInterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This class represents the base class that IK Solver plugins should derive from.

The IK solver is a pure mathematical function and does not hold state, but instead just solves a given, self-contained, mathematical problem, e.g. the plugin solver does not have influence on when IK is invoked and what an IK problem is (what is the goal and what are the joints, etc.), but contributes to IK by providing an answer on how to solve. Structurally, it is independent of the SDK and, hence, can be built independently, except for some theoretically independent math library. See the Inverse Kinematics section for more detailed information.

### Methods:

public:

### Prototype:

```
virtual ~IKSolver();
```

### Remarks:

Destructor.

## Class Identity

### Prototype:

```
virtual SClass_ID SuperClassID();
```

### Remarks:

Plugins derived from this class are supposed to have

**IK\_SOLVER\_CLASS\_ID** as their super class ID. This method should not be overridden.

### Default Implementation:

```
{ return IK_SOLVER_CLASS_ID; }
```

**Prototype:**

```
virtual Class_ID ClassID() = 0;
```

**Remarks:**

Implemented by the plug-in.

Returns the class ID of the IK Solver plugin.

**Prototype:**

```
virtual void GetClassName(TSTR& s);
```

**Remarks:**

Implemented by the plug-in.

This method returns the class name of the IK Solver plugin. This name will appear in the solver list from which users can pick or assign IK chains.

**Parameters:**

**TSTR& s**

The class name string.

**Default Implementation:**

```
{ s = TSTR(_T("IKSolver")); }
```

## Solver Traits

**Prototype:**

```
virtual bool IsInteractive() const = 0;
```

**Remarks:**

Implemented by the plug-in.

This method indicates whether the IK Solver is a controller or an interactive manipulation tool. In the former, the relationship between the goal and the joints are permanent: joints are completely controlled by the goal. In the latter, the relationship is transient, existing only during interactive manipulation. In the end, IK solutions are registered at each joint, mostly likely as key-frames, and it no longer matters how joints have got their joint angles. Only non-

interactive, or controller, IK solvers are supported in R4. Note that Interactive solvers do not need an initial pose, instead it needs a current pose.

**Return Value:**

TRUE if the IK Solver is an interactive tool, otherwise FALSE.

**Prototype:**

**virtual bool IsHistoryDependent() const = 0;**

**Remarks:**

Implemented by the plug-in.

At a specific point in time, the history dependent solver will reach solutions not only based the state of the goal at the time, but also its previous states (i.e. history dependent). On the contrary, the history independent solver does its job based on the state of the goal just at the time. The procedural implication is that, when the goal is changed at time t, the IK system would have to invalidate joints at time t for the history independent solver, and at all times that are greater or equal to t for the history dependent solver. In R4, only history dependent solvers are used by the IK system.

**Return Value:**

TRUE if the IK Solver is history dependent, otherwise FALSE.

**Prototype:**

**virtual bool UseSlidingJoint() const = 0;**

**Remarks:**

Implemented by the plug-in.

This method indicates whether the IK Solver intends to use the sliding joint (translational degrees of freedom) of the IK chain.

**Return Value:**

TRUE if the sliding joint of the IK chain is used, otherwise FALSE.

**Prototype:**

**virtual bool UseSwivelAngle() const = 0;**

**Remarks:**

Implemented by the plug-in.

This method indicates whether the IK Solver intends to use the swivel angle parameter of the IK chain.

**Return Value:**

TRUE if the swivel angle of the IK chain is used, otherwise FALSE.

**Prototype:**

**virtual bool DoesOneChainOnly() const = 0;**

**Remarks:**

Implemented by the plug-in.

When two IK chains overlap, i.e., there is a joint belonging to both IK chains, some solvers are able to negotiate between the possibly contending goals and some are not. This method indicates if the IK Solver does a single chain only. For those IK Solvers that can only solve one chain at a time, the IK system will pass to the solvers one chain at a time in a definitive order. In R4, only solvers that "do one chain only" are used.

**Return Value:**

TRUE if the IK Solver does only one chain, otherwise FALSE.

**Prototype:**

**virtual bool IsAnalytic() const;**

**Remarks:**

Implemented by the plug-in.

This method determines whether the IK Solver is analytic or needs to go through iterations. Solutions of an analytic IK Solver are not dependent on position and rotation thresholds or a maximum number of iterations.

**Return Value:**

TRUE if the IK Solver is analytic, otherwise FALSE.

**Default Implementation:**

**{ return false; }**

**Prototype:**

**virtual bool DoesRootJointLimits() const;**

**Remarks:**

Implemented by the plug-in.

This method determines whether the IK Solver handles root joint limits. If the IK Solver does not do joint limits, the result will be simply clamped into joint limits by the IK system.

**Return Value:**

TRUE if the IK Solver does root joint limits, otherwise FALSE.

**Default Implementation:**

```
{ return false;}
```

**Prototype:**

```
virtual bool DoesJointLimitsButRoot () const;
```

**Remarks:**

Implemented by the plug-in.

This method determines whether the IK Solver handles joint limits. If the IK Solver does not do joint limits, the result will be simply clamped into joint limits by the IK system.

**Return Value:**

TRUE if the IK Solver does joint limits, otherwise FALSE.

**Default Implementation:**

```
{ return false;}
```

**Prototype:**

```
virtual bool SolveEERotation() const = 0;
```

**Remarks:**

Implemented by the plug-in.

This method determines whether the rotational part of the goal node will be used.

**Return Value:**

TRUE if the rotational part of the goal node will be used, otherwise FALSE to indicate that only the position of the goal node is taken as the IK goal while the rotation threshold will be irrelevant.

## Solution Parameters

### Prototype:

```
virtual const IKSys::ZeroPlaneMap* GetZeroPlaneMap(const Point3& a0, const Point3& n0) const;
```

### Remarks:

Implemented by the plug-in.

IK Solvers may have their own Zero Plane Map. If so, they must override this method. The IK system will need it to perform IK snapping, which is setting the swivel angle based on the current pose so that the pose is consistent with the swivel angle. A Zero-Plane map can depend on the initial pose, which is when the joint angles take into account the respective preferred angles. In this method, **a0** is to be substituted for by the end effector axis, which is a unit vector, and **n0** by the solver plane normal, also a unit vector, when the chain is at the initial pose. The IK system will call this function using **IIKChainControl::InitEEAxis()** and **IIKChainControl::InitPlane()** for the two arguments.

### Parameters:

**const Point3& a0**

The end effector axis unit vector.

**const Point3& n0**

The solver plane normal.

### Return Value:

A pointer to the ZeroPlaneMap.

### Default Implementation:

```
{ return NULL; }
```

### Prototype:

```
virtual float GetPosThreshold() const = 0;
```

### Remarks:

This method allows you to retrieve the position threshold.

**Prototype:**

**virtual float SetPosThreshold(float) const = 0;**

**Remarks:**

This method allows you to set the position threshold.

**Parameters:**

**float**

The position threshold value.

**Prototype:**

**virtual float GetRotThreshold() const = 0;**

**Remarks:**

This method allows you to retrieve the rotation threshold.

**Prototype:**

**virtual float SetRotThreshold(float) const = 0;**

**Remarks:**

This method allows you to set the rotation threshold.

**Parameters:**

**float**

The rotation threshold value.

**Prototype:**

**virtual unsigned GetMaxIteration() const = 0;**

**Remarks:**

This method allows you to retrieve the maximum number of iterations.

**Prototype:**

**virtual void SetMaxIteration(unsigned) = 0;**

**Remarks:**

This method allows you to set the maximum number of iterations.

**Parameters:**

## **unsigned**

The maximum number of iterations.

## **Solving IK**

### **Prototype:**

```
virtual ReturnCondition Solve(IKSys::LinkChain&) = 0;
```

### **Remarks:**

Implemented by the plug-in.

This is the method that the IK system will call when it's the time to update the joints according to the IK goal and other parameters. The derived class should override this method if **DoesOneChainOnly()** returns TRUE and **HistoryDependent()** returns FALSE. Note that the solver is not designed to be invoked recursively. The recursion logic existing among the IK chains is taken care of by the 3ds max IK (sub-)System. The data structure passed to the Solver is transient and thus will be discarded once the solution is copied back to the joints. If the return condition indicates failure, (i.e. > 0xff) the result will not be copied back to the joint nodes in the 3ds max scene database.

### **Parameters:**

**IKSys::LinkChain&**

A reference to the Link Chain.

### **Return Value:**

The ReturnCondition bit-set with one or more of the following flags;

**bLimitReached**

The limit is reached.

**bLimitClamped**

The limit is clamped.

**bMaxIterationReached**

The maximum number of iterations is reached.

**bGoalTooCloseToEE**

The goal is too close to the end effector.

**bInvalidArgument**

An invalid argument is passed.

## **bInvalidInitialValue**

An invalid initial value is passed.

## Class IIKChainControl

See Also: : [Class IKCmdOps](#), [Class IKChainActions](#)

### Description:

**This class is only available in release 5 or later.**

The interface class to TM controller that the IK chain node employs. Suppose node is an IK chain node, following code obtains the interface pointer

```
IIKChainControl* ikchain = (IIKChainControl*) node->GetTMController()->GetInterface(I_IKChainControl);
```

From this pointer, we can get all properties pertinent to an IK Chain.

### Methods:

#### Prototype:

```
virtual INode* StartJoint() const
```

#### Remarks:

The start joint, end joint of the chain, and the node to that this IK chain controller belongs.

#### Prototype:

```
virtual INode* EndJoint() const
```

#### Remarks:

Please refer to StartJoint().

#### Prototype:

```
virtual INode* GetNode() const
```

**Remarks:**

Please refer to StartJoint().

**Prototype:**

**virtual Point3 ChainNormal(TimeValue t, Interval& valid)**

**Remarks:**

Preferred angles are used to start the IK iteration. Hence, the pose when joint angles assume the preferred angles is also called the Initial Pose in the context of IK. It is animatable, meaning that at different time, IK may start iteration from different poses.

Let's call the plane that the joints of an IK chain the (IK) solver plane.

**InitPlane()** and **InitEEAxis()** returns the normal to the solver plane and the axis from the start joint to the end joint (end-effector axis) at the initial (preferred angle) pose, at time of input argument. They are represented in the parent space of the start joint. **ChainNormal()** returns the normal in the object space, as

**InitPlane() == ChainNormal() \* startIKCont->PrefRotation()**

where

**startIKCont = (IIKControl\*)StartJoint()->GetController()->GetInterface(I\_IKControl);**

Furthermore, **ChainNormal()** takes as an input argument a validity interval that will be intersected by the validity interval of the chain normal.

**Prototype:**

**virtual Point3 InitPlane(TimeValue)**

**Remarks:**

Please refer to ChainNormal().

**Prototype:**

**virtual Point3 InitEEAxis(TimeValue)**

**Remarks:**

Please refer to ChainNormal().

**Prototype:**

**virtual float InitChainLength(TimeValue)**

**Remarks:**

Please refer to ChainNormal().

**Prototype:**

**virtual float SwivelAngle(TimeValue t, Interval& valid)**

The swivel angle at time. The validity interval, valid, will be intersected.

**Prototype:**

**virtual const IKSys::ZeroPlaneMap\*  
DefaultZeroPlaneMap(TimeValue)**

The zero plane is the plane that, at each "start joint to end joint" axis, is used as a reference plane with regard to that the swivel angle is defined. The zero plane map maps an axis to a plane normal. The IK system offers as a possible default via **DefaultZeroPlaneMap()**.

**Prototype:**

**virtual SAParentSpace SwivelAngleParent() const**

Whether the zero plane is defined in the parent space of the start joint, **kSAInStartJoint**, or in the space of the IK goal, **kSAInGoal**.

**Prototype:**

**virtual Solver() const**

The solver that is assigned to this chain.

**Prototype:**

**virtual bool SolverEnabled(TimeValue, Interval& valid)**

Whether the solver is enabled (IK mode) or not (FK mode).

**Prototype:**

**virtual bool CanAutoEnable() const**

When the chain is in the FK mode, IK can still be invoked when the goal is moved interactively if **CanAutoEnabled()** is true. It is a PB2 parameter of index **kAutoEnable**. It is not animatable.

**Prototype:**

**virtual bool AutoEnableSet() const**

It is a transient state that is alive only at the time when joint angles are to be updated. Being true means that the Enabled state of the chain is off the IK is turned on by interactive manipulation.

**Prototype:**

**virtual bool Valid() const**

Whether this chain is a valid one. It is valid if it is assigned a proper IK solver and it has valid start joint and end joint.

**Prototype:**

**virtual Interface\_ID GoalInterfaceID()**

What IK goal interface this chain is prepared for. For now, there are two

interfaces: HI IK goal (**IHIKGoal**) and Spline IK goal (**ISplineIKGoal**). An IK chain will admit of plugin solvers that support this goal interface (**IKSolver::ExpectGoal()**).

**Prototype:**

**virtual BaseInterface\* AcquireGoal(TimeValue, Interval& valid, const Matrix3& parent\_of\_start\_joint)**

It returns an interface to the goal at the time. Validity interval is reconciled. The parent matrix of the start joint is also returned in the third argument.

## Class IIKControl

See Also: [Class INode](#), [Class Control](#), [Class Point2](#), [Class Point3](#), [Class Interval](#), [Inverse Kinematics](#)

class IIKControl

### Description:

This class is available in release 4.0 and later only.

This class represents the TM controller for IK Chains and is used for nodes that serve as IK joints. This class is closely knit together with the Class

**IIKChainControl** and form the IK system. An interface pointer to the **IKControl** class can be obtained by using

**Animatable::GetInterface(I\_IKCONTROL)**. See the Inverse Kinematics section for more detailed information.

### Methods:

public:

## Degrees of Freedom

### Prototype:

```
virtual bool DofActive(DofAxis) const = 0;
```

### Remarks:

This method allows you to determine which degrees of freedom are active and inactive for the translational and rotational axes.

### Parameters:

#### DofAxis

The DOF axis to check, which is one of the following **IKSys::TransX**, **IKSys::TransY**, **IKSys::TransZ**, **IKSys::RotX**, **IKSys::RotY**, **IKSys::RotZ**.

### Return Value:

TRUE if the specified DOF axis is active, otherwise FALSE.

### Prototype:

**virtual DofSet ActiveTrans() const = 0;**

**Remarks:**

This method allows you to determine which degrees of freedom are active and inactive for the translational axes. The returned DofSet can be tested for **IKSys::DofX**, **IKSys::DofY**, and **IKSys::DofZ**.

**Return Value:**

A DofSet structure containing the translational axes.

**Prototype:**

**virtual DofSet ActiveRot() const = 0;**

**Remarks:**

This method allows you to determine which degrees of freedom are active and inactive for the rotational axes. The returned DofSet can be tested for **IKSys::DofX**, **IKSys::DofY**, and **IKSys::DofZ**.

**Return Value:**

A DofSet structure containing the translational axes.

**Prototype:**

**virtual DofSet ActiveDofs() const = 0;**

**Remarks:**

This method allows you to determine which degrees of freedom are active and inactive for the translational and rotational axes. The returned DofSet can be tested for **IKSys::TransX**, **IKSys::TransY**, **IKSys::TransZ**, **IKSys::RotX**, **IKSys::RotY**, **IKSys::RotZ**.

**Return Value:**

A DofSet structure containing the translational and rotational axes.

## **IK Chains**

**Prototype:**

**virtual INodeTab IKChains(JointType) const = 0;**

**Remarks:**

This method returns a list of IK Chain nodes. Note that an IK chain starts at the rotational joint of the Start Joint and ends at the sliding joint of the End Joint.

**Parameters:**

**JointType**

The joint type, either **IKSys::SlidingJoint** or **IKSys::RotationalJoint**.

**Return Value:**

The node table of IK Chain nodes.

**Prototype:**

**virtual bool IKBound(TimeValue t, JointType jt) = 0;**

**Remarks:**

The parameter that decides whether an individual degree of freedom is active is not animatable. There is an animatable variable of IK chain that decides whether the goal defined in the IK chain actually affects the joints it covers at a specific time. This method allows you to query that.

**Parameters:**

**TimeValue t**

The time at which to test the joint type.

**JointType jt**

The joint type, either **IKSys::SlidingJoint** or **IKSys::RotationalJoint**.

**Return Value:**

TRUE if bound, otherwise FALSE.

## **Joint Limits**

**Prototype:**

**virtual bool DofLowerLimited(DofAxis) const = 0;**

**Remarks:**

This method allows you to check if a specific DOF axis has its lower bounds limited.

**Parameters:**

### **DofAxis**

The DOF axis to check, which is one of the following **IKSys::TransX**, **IKSys::TransY**, **IKSys::TransZ**, **IKSys::RotX**, **IKSys::RotY**, **IKSys::RotZ**.

#### **Return Value:**

TRUE if limited, otherwise FALSE.

#### **Prototype:**

```
virtual bool DofUpperLimited(DofAxis) const = 0;
```

#### **Remarks:**

This method allows you to check if a specific DOF axis has its upper bounds limited.

#### **Parameters:**

##### **DofAxis**

The DOF axis to check, which is one of the following **IKSys::TransX**, **IKSys::TransY**, **IKSys::TransZ**, **IKSys::RotX**, **IKSys::RotY**, **IKSys::RotZ**.

#### **Return Value:**

TRUE if limited, otherwise FALSE.

#### **Prototype:**

```
virtual Point2 DofLimits(DofAxis) const = 0;
```

#### **Remarks:**

This method allows you to check if a specific DOF axis has its upper and lower bounds limited.

#### **Parameters:**

##### **DofAxis**

The DOF axis to check, which is one of the following **IKSys::TransX**, **IKSys::TransY**, **IKSys::TransZ**, **IKSys::RotX**, **IKSys::RotY**, **IKSys::RotZ**.

#### **Return Value:**

A Point2 where X and Y are the lower and upper limits, respectively.

**Prototype:**

**virtual Point3 TransLowerLimits() const = 0;**

**Remarks:**

This method allows you to retrieve the translation lower limits.

**Return Value:**

A Point3 where X, Y, and Z represents the actual limits.

**Prototype:**

**virtual Point3 TransUpperLimits() const = 0;**

**Remarks:**

This method allows you to retrieve the translation upper limits.

**Return Value:**

A Point3 where X, Y, and Z represents the actual limits.

**Prototype:**

**virtual Point3 RotLowerLimits() const = 0;**

**Remarks:**

This method allows you to retrieve the rotational lower limits.

**Return Value:**

A Point3 where X, Y, and Z represents the actual limits.

**Prototype:**

**virtual Point3 RotUpperLimits() const = 0;**

**Remarks:**

This method allows you to retrieve the rotational upper limits.

**Return Value:**

A Point3 where X, Y, and Z represents the actual limits.

## **FK Sub-Controller Access**

**Prototype:**

**virtual Control\* FKSubController() const = 0;**

**Remarks:**

This method allows you to obtain a pointer to the Forward Kinematics sub-controller. Note that the IK controller is not designed to be instanced. It is expected to have a unique node.

**Prototype:**

**virtual INode\* GetNode() const = 0;**

**Remarks:**

This method allows you to obtain a pointer to the node that holds the Forward Kinematics TM controller.

## **Preferred Angles**

**Prototype:**

**virtual Point3 PrefPosition(TimeValue t, Interval& validityInterval) = 0;**

**Remarks:**

This method allows you to retrieve the preferred angle of translation. Note that the angles are constant with regard to animation time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the preferred angle.

**Interval& validityInterval**

The validity interval.

**Return Value:**

The X, Y, and Z, preferred angles.

**Prototype:**

**virtual Point3 PrefRotation(TimeValue t, Interval& validityInterval) = 0;**

**Remarks:**

This method allows you to retrieve the preferred angle of rotation. Note that the angles are constant with regard to animation time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the preferred angle.

**Interval & validityInterval**

The validity interval.

**Return Value:**

The X, Y, and Z, preferred angles.

**Prototype:**

**virtual void SetPrefTrans(const Point3& val, TimeValue t = 0) = 0;**

**Remarks:**

A solver may start off the solution process with joint angles being set to special values, preferred angles. This method allows you to set the preferred angles of the translational joints.

**Parameters:**

**const Point3& val**

The preferred angles you wish to set.

**TimeValue t = 0**

The time at which to set them.

**Prototype:**

**virtual void SetPrefRot(const Point3& val, TimeValue t = 0) = 0;**

**Remarks:**

A solver may start off the solution process with joint angles being set to special values, preferred angles. This method allows you to set the preferred angles of the rotational joints.

**Parameters:**

**const Point3& val**

The preferred angles you wish to set.

**TimeValue t = 0**

The time at which to set them.

**Prototype:**

**virtual void SetPrefTR(const Point3& trans, const Point3& rot, TimeValue t = 0) = 0;**

**Remarks:**

A solver may start off the solution process with joint angles being set to special values, preferred angles. This method allows you to set the preferred angles of both the translational and rotational joints.

**Parameters:**

**const Point3& trans**

The preferred translational angles you wish to set.

**const Point3& rot**

The preferred rotational angles you wish to set.

**TimeValue t = 0**

The time at which to set them.

## Joint Angles

**Prototype:**

**virtual Point3 TransValues(TimeValue, Interval\* = 0) = 0;**

**Remarks:**

This method returns the angles of sliding joints at a specific time.

**Parameters:**

**TimeValue**

The time at which to retrieve the values.

**Interval\* = 0**

The validity interval. If non-null the validity interval will be updated.

**Prototype:**

**virtual Point3 RotValues(TimeValue, Interval\* = 0) = 0;**

**Remarks:**

This method returns the angles of rotational joints at a specific time.

**Parameters:**

**TimeValue**

The time at which to retrieve the values.

**Interval\* = 0**

The validity interval. If non-null the validity interval will be updated.

**Prototype:**

**virtual void AssignTrans(const Point3&, const Interval&) = 0;**

**Remarks:**

This method allows you to set the angles of translational joints. Note that this method does not adjust the validity interval.

**Parameters:**

**const Point3&**

The joint angles.

**const Interval&**

The validity interval

**Prototype:**

**virtual void AssignRot(const Point3&, const Interval&) = 0;**

**Remarks:**

This method allows you to set the angles of rotational joints. Note that this method does not adjust the validity interval.

**Parameters:**

**const Point3&**

The joint angles.

**const Interval&**

The validity interval

**Prototype:**

**virtual void AssignActiveTrans(const Point3&, const Interval&) = 0;**

**Remarks:**

This method allows you to set the angles of translational joints. Note that this method will skip those degrees of freedom that are not active and that this method will not adjust the validity interval.

**Parameters:**

**const Point3&**

The joint angles.

**const Interval&**

The validity interval

**Prototype:**

**virtual void AssignActiveTrans(const DofSet&, const float[], const Interval&) = 0;**

**Remarks:**

This method allows you to set the angles of translational joints. The active DOF's are given as the first argument of type DofSet and the new values are supplied as a float array whose size should be the same as the DofSet [DofSet::Count()]. Note that this method does not adjust the validity interval.

**Parameters:**

**const DofSet&**

The degrees of freedom.

**const float[]**

The new angles you wish to set.

**const Interval&**

The validity interval

**Prototype:**

**virtual void AssignActiveRot(const Point3&, const Interval&) = 0;**

**Remarks:**

This method allows you to set the angles of rotational joints. Note that this method will skip those degrees of freedom that are not active and that this method does not adjust the validity interval.

**Parameters:**

**const Point3&**

The joint angles.

**const Interval&**

The validity interval

**Prototype:**

**virtual void AssignActiveRot(const DofSet&, const float[], const Interval&) = 0;**

**Remarks:**

This method allows you to set the angles of rotational joints. The active DOF's are given as the first argument of type DofSet and the new values are supplied as a float array whose size should be the same as the DofSet [DofSet::Count()]. Note that this method does not adjust the validity interval.

**Parameters:**

**const DofSet&**

The degrees of freedom.

**const float[]**

The new angles you wish to set.

**const Interval&**

The validity interval

**Prototype:**

**virtual void SetTransValid(const Interval& valid) = 0;**

**Remarks:**

This method allows you to set the validity interval for translational joint angles.

**Parameters:**

**const Interval& valid**

The validity interval.

**Prototype:**

**virtual void SetRotValid(const Interval& valid) = 0;**

**Remarks:**

This method allows you to set the validity interval for rotational joint angles.

**Parameters:**

**const Interval& valid**

The validity interval.

**Prototype:**

**virtual void SetTRValid(const Interval& valid) = 0;**

**Remarks:**

This method allows you to set the validity interval for both translational and rotational joint angles.

**Parameters:**

**const Interval& valid**

The validity interval.

## Class ZeroPlaneMap

See Also: [Class Point3](#), [Inverse Kinematics](#)

class ZeroPlaneMap

### Description:

This class is available in release 4.0 and later only.

This class is defined in **IKHierarchy.h** and provides the functionality that, given a unit axis, which is to be substituted by the end effector (EE) Axis, produces a unit vector, which will be interpreted as the normal to a plane. For more details see the section on Inverse Kinematics.

### Methods:

public:

### Prototype:

**virtual ~ZeroPlaneMap()**

### Remarks:

Destructor.

### Default Implementation:

{ }

### Prototype:

**virtual Point3 operator()(const Point3& EEAxis) const = 0;**

### Remarks:

The ZeroPlaneMap call operator.

### Parameters:

**const Point3& EEAxis**

The end effector axis.

### Return Value:

The unit vector described above.

## Class RootLink

See Also: [Class LinkChain](#), [Class Link](#), [Class Matrix3](#), [Class Point3](#), [Inverse Kinematics](#)

class RootLink

### Description:

This class is available in release 4.0 and later only.

A RootLink consists of a rotation plus a rigidExtend. It transforms like this:

**To\_Coordinate\_Frame = rigidExtend \* rotXYZ \*  
From\_Coordinate\_Frame**

where **rotXYZ = Rot\_x(rotXYZ[0]) \* Rot\_y(rotXYZ[1]) \*  
Rot\_z(rotXYZ[2])**.

Note that not all the x, y, and z, are degrees of freedom. Only Active() ones are. We put the whole rotation here so that some solver may choose to use it as a full rotation and then clamp the result to the permissible range.

### Data Members:

public:

**Point3 rotXYZ;**

The xyz rotation.

**Point3 initXYZ;**

Corresponds to the **PrefRotation()** of the Start Joint.

**Point3 llimits;**

The lower limits.

**Point3 ulimits;**

The upper limits.

**Matrix3 rigidExtend;**

The rigid extents.

private:

**unsigned flags;**

The root link flags.

### Methods:

public:

**Prototype:**

**RootLink():flags(7)**

**Remarks:**

Constructor.

**Default Implementation:**

```
{ }
```

**Prototype:**

**bool GetActive(unsigned i) const;**

**Remarks:**

This method informs the Solver whether a particular component is active.

**Parameters:**

**unsigned i**

The component, 0 through 3 for x, y, and z, respectively.

**Return Value:**

TRUE if the specified component is active, otherwise FALSE.

**Default Implementation:**

```
{ return flags&(1<<i)?true:false; }
```

**Prototype:**

**void SetActive(unsigned i, bool s);**

**Remarks:**

This method allows you to activate or deactivate a particular component.

**Parameters:**

**unsigned i**

The component, 0 through 3 for x, y, and z, respectively.

**bool s**

TRUE to activate, FALSE to deactivate.

**Prototype:**

**bool GetLLimited(unsigned i) const;**

**Remarks:**

This method informs the Solver whether a particular component has its lower limits active.

**Parameters:****unsigned i**

The component, 0 through 3 for x, y, and z, respectively.

**Return Value:**

TRUE if the specified component is active, otherwise FALSE.

**Default Implementation:**

```
{ return flags&(1<<(i+3))?true:false; }
```

**Prototype:**

```
bool SetLLimited(unsigned i, bool s) const;
```

**Remarks:**

This method allows you to activate or deactivate a particular component's lower limits.

**Parameters:****unsigned i**

The component, 0 through 3 for x, y, and z, respectively.

**bool s**

TRUE to activate, FALSE to deactivate.

**Prototype:**

```
bool GetULimited(unsigned i) const;
```

**Remarks:**

This method informs the Solver whether a particular component has its upper limits active.

**Parameters:****unsigned i**

The component, 0 through 3 for x, y, and z, respectively.

**Return Value:**

TRUE if the specified component is active, otherwise FALSE.

**Default Implementation:**

```
{ return flags&(1<<(i+6))?true:false; }
```

**Prototype:**

```
bool SetULimited(unsigned i, bool s) const;
```

**Remarks:**

This method allows you to activate or deactivate a particular component's upper limits.

**Parameters:**

**unsigned i**

The component, 0 through 3 for x, y, and z, respectively.

**bool s**

TRUE to activate, FALSE to deactivate.

**Prototype:**

```
Matrix3& RotateByAxis(Matrix3& mat, unsigned i) const;
```

**Remarks:**

This method will pre-apply the rotation about the x, y, or z axis. Therefore, starting with the identity matrix **mat**,

```
ApplyLinkMatrix(  
    RotateByAxis(  
        RotateByAxis(  
            RotateByAxis(mat, 2),  
            1),  
        0),  
    false)
```

should equal to **LinkMatrix(true)**.

**Parameters:**

**unsigned i**

The component, 0 through 3 for x, y, and z, respectively.

**Prototype:**

**Matrix3 LinkMatrix(bool include\_rot) const;**

**Remarks:**

This method returns the link matrix just defined if the argument is TRUE.

**Parameters:**

**bool include\_rot**

TRUE to return the link matrix, FALSE to return **RigidExtend**.

**Return Value:**

The link matrix, otherwise it simply returns **RigidExtend**.

**Prototype:**

**Matrix3& ApplyLinkMatrix(Matrix3& mat, bool include\_rot) const;**

**Remarks:**

This methods applies the LinkMatrix() to the input matrix, or **mat = mat \* LinkMatrix(include\_rot)**

**Parameters:**

**Matrix3& mat**

The input matrix.

**bool include\_rot**

When applying the DOF part, or rotation part, to a matrix, this will take place one at a time by calling **RootLink::RotateByAxis()**. If you want to apply the whole link, while already having applied the rotation part, you would need to set this flag to FALSE.

**Return Value:**

The reference to the input matrix, **mat**.

## Class Link

See Also: [Class LinkChain](#), [Class RootLink](#), [Class Matrix3](#), [Class Point3](#), [Inverse Kinematics](#)

class Link

### Description:

This class is available in release 4.0 and later only.

This class represents a single link in the link chain. A Link is a single degree of freedom rotation followed by a rigidExtend. The DOF axis is specified by dofAxis. It is always active.

### Data Members:

public:

#### **DofAxis dofAxis;**

The variable part of a Link is of one degree of freedom. It can be translational or rotational. One of the following; **TransX**, **TransY**, **TransZ**, **RotX**, **RotY**, or **RotZ**.

#### **float dofValue;**

The current value with regard to the degree of freedom.

#### **float initValue;**

The initial value.

#### **Point2 limits;**

The constrained lower and upper limits. [0] for the lower limit and [1] for the upper limit.

private:

#### **Matrix3 rigidExtend;**

The rigid extents.

#### **byte llimited : 1;**

Lower limit flag.

#### **byte ulimited : 1;**

Upper limit flag.

### Methods:

public:

**Prototype:**

**Link():rigidExtend(0),dofAxis(RotZ);**

**Remarks:**

Constructor.

**Default Implementation:**

**{ }**

**Prototype:**

**~Link();**

**Remarks:**

Destructor.

**Default Implementation:**

**{ if (rigidExtend) delete rigidExtend; rigidExtend = 0; }**

**Prototype:**

**bool NullLink() const;**

**Remarks:**

This method checks whether the link is a null-link. When TRUE, the rigid extend is logically an identity matrix.

**Default Implementation:**

**{ return rigidExtend?false:true; }**

**Prototype:**

**bool ZeroLength();**

**Remarks:**

This method checks whether the link has no length. When TRUE, it is a pure rotation matrix.

**Prototype:**

**bool LLimited() const;**

**Remarks:**

This method checks whether the degree of freedom is further constrained by lower limits. TRUE if constrained, otherwise FALSE.

**Default Implementation:**

```
{ return limited?true:false; }
```

**Prototype:**

```
bool ULimited() const;
```

**Remarks:**

This method checks whether the degree of freedom is further constrained by upper limits. TRUE if constrained, otherwise FALSE.

**Default Implementation:**

```
{ return ulimited?true:false; }
```

**Prototype:**

```
Matrix3 DofMatrix() const;
```

**Remarks:**

This method returns the matrix contribution by the degrees of freedom. Either it is a pure rotation or a pure translation, of one axis. The following identity holds;

```
LinkMatrix(true) ° LinkMatrix(false) * DofMatrix()
```

**Prototype:**

```
Matrix3& DofMatrix(Matrix3& mat) const;
```

**Remarks:**

This method allows you to apply a matrix, **mat**, by the **DofMatrix()** so that **mat = mat \* DofMatrix()**.

**Parameters:**

```
Matrix3& mat
```

The matrix to multiply by the DOF matrix.

**Return Value:**

A reference to the matrix argument.

**Prototype:**

**Matrix3 LinkMatrix(bool include\_dof = true) const;**

**Remarks:**

This method returns the link matrix just defined if the argument is TRUE.

**Parameters:**

**bool include\_dof = true**

TRUE to return the link matrix, FALSE to return **RigidExtend**.

**Return Value:**

The link matrix, otherwise it simply returns **RigidExtend**.

**Prototype:**

**Matrix3& ApplyLinkMatrix(Matrix3& mat, bool include\_dof = true) const;**

**Remarks:**

This methods applies the LinkMatrix() to the input matrix.

**Parameters:**

**Matrix3& mat**

The input matrix.

**bool include\_dof = true**

When applying the DOF part, or rotation part, to a matrix, this will take place one at a time by calling **RootLink::RotateByAxis()**. If you want to apply the whole link, while already having applied the rotation part, you would need to set this flag to FALSE.

**Return Value:**

The reference to the input matrix, **mat**.

**Prototype:**

**void SetLLimited(bool s) const;**

**Remarks:**

This method allows you to activate or deactivate the lower limits.

**Parameters:**

**bool s**

TRUE to activate, FALSE to deactivate.

**Prototype:**

**void SetULimited(bool s) const;**

**Remarks:**

This method allows you to activate or deactivate the upper limits.

**Parameters:**

**bool s**

TRUE to activate, FALSE to deactivate.

**Prototype:**

**void SetRigidExtend(const Matrix3& mat);**

**Remarks:**

This method allows you to set the RigidExtend matrix.

**Parameters:**

**const Matrix3& mat**

The rigid extend matrix you wish to set.

# Class LinkChain

## Description:

**This class is only available in release 5 or later.**

This class abstracts the data structure that the IK system pass to the plugin solver. With it, an IK problem becomes a pure mathematical one to the plugin solver.

A **LinkChain** starts with a **rootLink**, followed by a number of 1D link. The rootLink has a whole rotation (c.f. class **RootLink** in ikHierarch.h), as represented by Euler angles of order XYZ, rotXYZ. These angles may be marked as active or not. The **RootLink** exposes the whole rotation to allow the solver to handle the start joint specially.

The next 1D link is to be pivoted and aligned at the reference frame that is offset from the rotation of the **RootLink** by "**rigidExtend**".

## Methods:

### Prototype:

**Matrix3** parentMatrix

### Remarks:

This **LinkChain** is originally cut off from a transformation space quantified by this matrix. Specifically, it is the matrix that accumulates all transformations from the root of the world to the position component of the start joint.

### Prototype:

**unsigned LinkCount()** const

### Remarks:

The number of 1D links following the rootLink.

**Prototype:**

**const Link& LinkOf(unsigned i) const**

**Remarks:**

The i-th 1D link. The next link is pivoted and aligned at

**LinkOf(i).LinkMatrix(true) \* LinkOf(i-1).LinkMatrix(true) ... \*  
LinkOf(0).LinkMatrix(true) \* rootLink.LinkMatrix(true)**

This is the extremity after the i-th link. The position of the end effector, in particular, is the extremity of the last link.

**Prototype:**

**Link& LinkOf(unsigned i)**

**Remarks:**

Please refer to the above entry for full explanation.

**Prototype:**

**int PreBone(unsigned i) const**

**Remarks:**

A 1D link comprises a degree of freedom, which can be rotational or prismatic (sliding), and an offset transformation, "rigidExtend". If the length of "rigidExtend" is zero, the next 1D link comes from same joint of this link. PreBone(i) returns the first link that precedes the i-th link that has non-zero "rigidExtend." In other words, LinkOf(PreBone(i) + 1) starts a new joint that includes i-th link as one of its degrees of freedom. PreBone(i) is always less than i.

**Prototype:**

**unsigned Bone(unsigned i) const**

**Remarks:**

**Bone(i)** returns the first index,  $j$ , such that  $j \geq i$  and **LinkOf(j).ZeroLength()** false. This is the last link of the bone that includes  $i$ -th link as a degree of freedom.

**Prototype:**

**BaseInterface\* GetIKGoal()**

**Remarks:**

Returns an interface pointer to the goal. The actual type of goal can be queried from the interface. (They are documented separately.)

**Prototype:**

**void SetIKGoal(BaseInterface\*)**

**Remarks:**

They are used to set and release the goal. The plugin solver should not worried about it. The IK system will use them to the set and release the goal.

**Prototype:**

**void ReleaseIKGoal()**

**Remarks:**

Please refer to **SetIKGoal()** for full explanation.

## Class IKCmdOps

See Also: : [Class IKChainActions](#), [Class IKChainControl](#)

```
class IKCmdOps : public FPStaticInterface
```

### Description:

**This class is only available in release 5 or later.**

The program interface to functions of the IK system. To obtain a pointer to the interface, use

```
IKCmdOps* iksys = GetCoreInterface(IK_FP_INTERFACE_ID);
```

It should be non-null if the plugin of the IK system is loaded.

### Methods:

#### Prototype:

```
virtual INode* CreateIKChain(INode* start, INode* end, const  
TCHAR* solver);
```

#### Remarks:

This method assigns an IK solver from nodes start to end. To be successful, following conditions have to be met:

1. Start must be an ancestor of end in the node hierarchy;
2. All TM controllers along the chain must be replaceable;
3. The 3rd argument, solver, must be a name of a plugin solver.

Upon success, it will create an IK chain node that contains these pieces of information plus the IK goal and returns a pointer to it. Returning non-null

pointer indicates failure.

Example:

```
INode* ikchainNode = iksys->CreateIKChain(bone1, bone4,  
_T("HIIKSolver"));
```

**Prototype:**

```
virtual int SolverCount() const;
```

**Remarks:**

**Returns the number of IK solvers that have been loaded in.**

**Prototype:**

```
virtual TSTR SolverName(int i) const;
```

**Remarks:**

Returns the internal name of the i-th solver. This name is used in **CreateIKChain()** or the script.

**Prototype:**

```
virtual TSTR SolverUIName(int i) const;
```

**Remarks:**

Returns the UI name of the i-th solver. The UI names appear in the solver list on the Motion Panel, or the Animation menu. They are localized.

## Class ITCBFloatKey

See Also: [Class ITCBKey](#).

```
class ITCBFloatKey : public ITCBKey
```

### **Description:**

This class stores a Tension Continuity and Bias (TCB) floating point key.

### **Data Members:**

public:

**float val;**

The value of the key.

## Class ITCBPoint3Key

See Also: [Class ITCBKey](#), [Class Point3](#).

```
class ITCBPoint3Key : public ITCBKey
```

### **Description:**

This class stores a Tension Continuity and Bias (TCB) Point3 key.

### **Data Members:**

public:

**Point3 val;**

The value of the key.

## Class ITCBRotKey

See Also: [Class ITCBKey](#), [Class AngAxis](#).

```
class ITCBRotKey : public ITCBKey
```

### **Description:**

This class stores a Tension Continuity and Bias (TCB) rotation key.

### **Data Members:**

public:

**AngAxis val;**

The value of the key.

## Class ITCBScaleKey

See Also: [Class ITCBKey](#), [Class ScaleValue](#).

```
class ITCBScaleKey : public ITCBKey
```

### **Description:**

This class stores a Tension Continuity and Bias (TCB) Scale key.

### **Data Members:**

public:

**ScaleValue val;**

The value of the key.

## Class IBezFloatKey

See Also: [Class IKey](#).

```
class IBezFloatKey : public IKey
```

### Description:

This class stores a Bezier floating point key.

### Data Members:

public:

**float intan;**

The in tangent value.

**float outtan;**

The out tangent value.

**float val;**

The value of the key.

**float inLength;**

The in length of the horizontal handle.

**float outLength;**

The out length of the horizontal handle.

## Class IBezPoint3Key

See Also: [Class IKey](#), [Class Point3](#).

```
class IBezPoint3Key : public IKey
```

### Description:

This class stores a Bezier Point3 key.

The value of the intan and outtan values of IBezPoint3Key returned from GetKey is actually the tangent of the angle that is spanned between the horizontal axis and the tangent. In order to get the actual handle of the tangent it is important to know that the horizontal (time) distance from the handle to the key value is dependent on the previous (for intan) or next (for outtan) point. The horizontal distance is basically a third of the total distance between the current key and the previous (or next) key :

```
Control *c = node->GetTMController()->GetPositionController();  
dt = (c->GetKeyTime(PreviousIdx)-c->GetKeyTime(CurrentIdx)) /  
3.0f;
```

since the value in IBezPoint3Key pos\_key.intan.x is the tan(alpha), you can easily get the vertical location of the handle by calculating :

```
tan(alpha) = pos_key.intan.x = dy/dt -> dy = pos_key.intan.x * dt
```

dt is the horizontal coordinate of the tangent handle relative to the key value.

dy is the vertical coordinate of the tangent handle relative to the key value.

### Data Members:

public:

#### **Point3 intan;**

The incoming tangent vector. This can be thought of as the tangent handle of a 3D spline. The value is relative to **val**.

#### **Point3 outtan;**

The outgoing tangent vector. This can be thought of as the tangent handle of a 3D spline. The value is relative to **val**.

#### **Point3 val;**

The value of the key.

#### **Point3 inLength;**

The in length of the horizontal handle.

**Point3 outLength;**

The out length of the horizontal handle.

## Class IBezQuatKey

See Also: [Class IKey](#), [Class Quat](#).

```
class IBezQuatKey : public IKey
```

### **Description:**

This class stores a Bezier quaternion key.

### **Data Members:**

public:

#### **Quat val;**

The value of the key.

## Class IBezScaleKey

See Also: [Class IKey](#), [Class Point3](#), [Class ScaleValue](#).

```
class IBezScaleKey : public IKey
```

### Description:

This class stores a Bezier scale key.

### Data Members:

public:

**Point3 intan;**

The int tangent vector.

**Point3 outtan;**

The out tangent vector.

**ScaleValue val;**

The value of the key.

**Point3 inLength;**

The in length of the horizontal handle.

**Point3 outLength;**

The out length of the horizontal handle.

## Class ILinFloatKey

See Also: [Class IKey](#).

```
class ILinFloatKey : public IKey
```

### **Description:**

This class stores a Linear floating point key.

### **Data Members:**

public:

**float val;**

The value of the key.

## Class ILinPoint3Key

See Also: [Class IKey](#), [Class Point3](#).

```
class ILinPoint3Key : public IKey
```

### **Description:**

This class stores a Linear Point3 key.

### **Data Members:**

public:

#### **Point3 val;**

The value of the key.

## Class ILinRotKey

See Also: [Class IKey](#), [Class Quat](#).

```
class ILinRotKey : public IKey
```

### **Description:**

This class stores a Linear rotation key.

### **Data Members:**

public:

**Quat val;**

The value of the key.

## Class ILinScaleKey

See Also: [Class IKey](#), [Class ScaleValue](#).

```
class ILinScaleKey : public IKey
```

### **Description:**

This class stores a Linear scale key.

### **Data Members:**

public:

**ScaleValue val;**

The value of the key.

## Class AppSave

See Also: [Class AppLoad](#), [Class AppDataChunk](#), [Character Strings](#).

```
class AppSave
```

### Description:

This class is available in release 2.0 and later only.

This class is a general chunk-ifying code useful for writing hierarchical data structures to a linear stream, such as an AppData block.

All methods of this class are implemented by the system.

AppSave will write hierarchical chunks into a private buffer, enlarging it as needed. When completed, use the methods **BufferPtr()** and **NBytesWritten()** to get at this buffer. (**AppSave** will delete the buffer in its **DeleteThis()** method, so you need to copy the buffer to save the data.)

The chunk hierarchy should always have a single highest level chunk. Chunks can be nested to any depth.

A Chunk can contain either sub-chunks, or data, but not both.

### Sample Code:

```
AppSave *asave = NewAppSave(1000);
asave->BeginChunk(MAIN_CHUNK);
asave->BeginChunk(CHUNK1);
    // .. write data
asave->EndChunk();

asave->BeginChunk(CHUNK2);
    // .. write data
asave->EndChunk();

asave->BeginChunk(CHUNK3);
    // .. write data
asave->EndChunk();
asave->EndChunk(); // end MAIN_CHUNK
```

### Prototype:

**AppSave \*NewAppSave(int initBufSize);**

**Remarks:**

This global function create a new **AppSave** instance.

**Parameters:**

**int initBufSize**

Specifies the initial size the internal buffer is allocated to. It will be enlarged if necessary.

**Methods:**

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

This method deletes the **AppSave** instance.

**Prototype:**

**virtual BYTE \*BufferPtr()=0;**

**Remarks:**

This method is used after saving to get a pointer to the buffer created.

**Prototype:**

**virtual int NBytesWritten()=0;**

**Remarks:**

This method returns the number of bytes that were written in the buffer.

**Prototype:**

**virtual void BeginChunk(USHORT id)=0;**

**Remarks:**

This method is used to begin a chunk. The ID passed need only be unique within the plug-ins data itself.

**Parameters:**

**USHORT id**

The id for the chunk.

**Prototype:**

**virtual void EndChunk()=0;**

**Remarks:**

This method is used to end a chunk, and back-patch the length.

**Prototype:**

**virtual int CurChunkDepth()=0;**

**Remarks:**

This method is used internally for checking balanced BeginChunk/EndChunk.

**Prototype:**

**virtual IOResult Write(const void \*buf, ULONG nbytes, ULONG \*nwrit)=0;**

**Remarks:**

This method writes a block of bytes.

**Parameters:**

**const void \*buf**

The buffer to write.

**ULONG nbytes**

The number of bytes to write.

**ULONG \*nwrit**

The number of bytes actually written.

**Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**virtual IOResult WriteWString(const char \*str)=0;**

**Remarks:**

This method is used to write wide character strings.

**Parameters:**

**const char \*str**

The string to write.

**Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**virtual IOResult WriteWString(const wchar\_t \*str)=0;**

**Remarks:**

This method is used to write wide character strings.

**Parameters:**

**const wchar\_t \*str**

The string to write.

**Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**virtual IOResult WriteCString(const char \*str)=0;**

**Remarks:**

This method is used to write single byte character strings.

**Parameters:**

**const char \*str**

The string to write.

**Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**virtual IOResult WriteCString(const wchar\_t \*str)=0;**

**Remarks:**

This method is used to write single byte character strings.

**Parameters:**

**const wchar\_t \*str**

The string to write.

**Return Value:**

**IO\_OK** - The write was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

# Class AppLoad

See Also: [Class AppSave](#), [Class AppDataChunk](#).

class AppLoad

## Description:

This class takes a chunk-ified data stream (as written by Class AppSave), and provides routines for decoding it.

This class is available in release 2.0 and later only.

All methods of this class are implemented by the system.

## Prototype:

**AppLoad \*NewAppLoad(BYTE \*buf, int bufSize);**

## Remarks:

This global function creates a new AppLoad instance for reading chunks out of **buf**:

## Parameters:

**BYTE \*buf**

The buffer to read.

**int bufSize**

Specifies the number of bytes that are valid in **buf**.

## Methods:

### Prototype:

**virtual void DeleteThis()=0;**

### Remarks:

This method deletes the instance of AppLoad.

### Prototype:

**virtual IOResult OpenChunk()=0;**

### Remarks:

This method is used to open a chunk. If **OpenChunk()** returns **IO\_OK**, use the following 3 functions to get the information about the chunk. If it returns **IO\_END** this indicates there are no more chunks at this level.

**Return Value:**

**IO\_OK** - The result was acceptable - no errors.

**IO\_END** - This is returned when the end of the chunks at a certain level have been reached. It is used as a signal to terminate the processing of chunks at that level.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**virtual USHORT CurChunkID()=0;**

**Remarks:**

This method returns the ID of the most recently opened chunk.

**Prototype:**

**virtual ChunkType CurChunkType()=0;**

**Remarks:**

This method returns the type of the most recently opened chunk. This may be one of the following values:

**NEW\_CHUNK**

**CONTAINER\_CHUNK**

**DATA\_CHUNK**

**Prototype:**

**virtual ULONG CurChunkLength()=0;**

**Remarks:**

This method returns the chunk length **not** including the header.

**Prototype:**

**virtual int CurChunkDepth()=0;**

**Remarks:**

This method is used internally for checking for balanced OpenChunk/CloseChunk pairs.

**Prototype:**

**virtual IOResult CloseChunk()=0;**

**Remarks:**

This method is used to close the currently opened chunk, and position at the next chunk.

**Parameters:****Return Value:**

A return value of **IO\_ERROR** indicates there is no open chunk to close; otherwise **IO\_OK**.

**Prototype:**

**virtual USHORT PeekNextChunkID()=0;**

**Remarks:**

This method returns the ID of the next chunk without opening it. It returns 0 if there are no more chunks.

**Prototype:**

**virtual IOResult Read(void \*buf, ULONG nbytes, ULONG \*nread)=0;**

**Remarks:**

This method is used to read a block of bytes.

**Parameters:**

**void \*buf**

A pointer to the buffer to read.

**ULONG nbytes**

The number of bytes to read.

**ULONG \*nread**

The number of bytes that were read.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

**virtual IOResult ReadWStringChunk(char\*\* buf)=0;**

**Remarks:**

This method read a string that was stored as Wide characters. Note: This method reads a string from a string chunk. It is assumed the chunk is already open, it will NOT close the chunk.

**Parameters:**

**char\*\* buf**

A pointer to an array of characters.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

**virtual IOResult ReadWStringChunk(wchar\_t\*\* buf)=0;**

**Remarks:**

This method read a string that was stored as Wide characters. Note: This method reads a string from a string chunk. It is assumed the chunk is already open, it will NOT close the chunk.

**Parameters:**

**wchar\_t\*\* buf**

A pointer to an array of wide characters.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

**virtual IOResult ReadCStringChunk(char\*\* buf)=0;**

**Remarks:**

This method reads a string that was stored as single byte characters.

**Parameters:**

**char\*\* buf**

A pointer to an array of single byte characters.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

**Prototype:**

**virtual IOResult ReadCStringChunk(wchar\_t\*\* buf)=0;**

**Remarks:**

This method read a string that was stored as Wide chars. Note: This method reads a string from a string chunk. It is assumed the chunk is already open, it will NOT close the chunk.

**Parameters:**

**wchar\_t\*\* buf**

A pointer to an array of wide characters.

**Return Value:**

A return value of **IO\_ERROR** indicates an error occurred, otherwise **IO\_OK**.

## Class BigMatrix

See Also: [Class Matrix3](#).

class BigMatrix

### Description:

This class is available in release 2.0 and later only.

This class implements an  $m \times n$  matrix for situations & calculations where the usual 4x3 Matrix3 class is not adequate. BigMatrix implements several useful matrix operations, including matrix multiplication and inversion, but is not guaranteed to be especially efficient. All methods are implemented by the system.

### Data Members:

public:

**int m, n**

The dimensions of the matrix. There are **m** rows and **n** columns.

**float \*val**

The elements of the matrix.  $val[i*n+j]$  is the value in the  $i$ 'th row and the  $j$ 'th column.

### Methods:

#### Prototype:

**BigMatrix();**

#### Remarks:

Initializer. Sets m and n to zero, and val to NULL.

#### Prototype:

**BigMatrix(int mm, int nn);**

#### Remarks:

Initializer. Sets the dimensions of the matrix to mm by nn, and allocates space for the contents. The total size of the matrix,  $mm*nn$ , cannot exceed 10,000.

#### Prototype:

**BigMatrix(const BigMatrix & from);**

**Remarks:**

Initializer. Sets this BigMatrix equal to **from**.

**Prototype:**

**~BigMatrix();**

**Remarks:**

Destructor. Frees the internal arrays.

**Prototype:**

**void Clear();**

**Remarks:**

Frees the internal arrays and sets the matrix's size to 0x0.

**Prototype:**

**int SetSize(int mm, int nn);**

**Remarks:**

Sets the matrix's size to mm by nn, and allocates space for the contents.

**Return Value:**

Returns the total size of the matrix (mm \* nn) or -1 if there's an error.

**Prototype:**

**float \*operator[](int i) const;**

**Remarks:**

Returns a pointer to the i'th row in the matrix. Thus for a BigMatrix A, A[i][j] is the value in the i'th row and the j'th column.

**Prototype:**

**BigMatrix & operator=(const BigMatrix & from);**

**Remarks:**

Sets this BigMatrix equal to from.

**Prototype:**

**void SetTranspose(BigMatrix & trans) const;**

**Remarks:**

Sets trans to be the transpose of this BigMatrix.

**Prototype:**

**float Invert();**

**Remarks:**

Inverts this matrix. Note that this only works if this matrix is "square", i.e. if  $m = n$ . This algorithm is CUBIC in the number of rows, so be careful!

**Return Value:**

The determinant of the matrix (before inversion) is returned. If the return value is 0, the matrix could not be inverted.

**Prototype:**

**void Identity();**

**Remarks:**

If  $m$  and  $n$  are equal, this method sets this matrix to the identity. If  $m$  and  $n$  are not equal, it does nothing.

**Prototype:**

**void Randomize(float scale);**

**Remarks:**

This method is available in release 2.5 or later only.

Creates a random matrix for testing purposes. Reseeds the random number generator with the current system time, for a non-reproducible result. Values of the matrix are set to anything in the range  $(-scale, scale)$ .

**Prototype:**

**void MNDebugPrint();**

**Remarks:**

This method is available in release 2.5 or later only.

This method prints the contents of the BigMatrix to the IDE debugging window using DebugPrints.

# Class INodeTransformed

See Also: [Class INode](#), [Modifier Stack Branching](#)

## Description:

This class provides a layer that will add in a transformation to the node's objectTM. Most methods pass through to the INode, except for the objectTM methods which pre-multiply in the given matrix. The methods of this class are the same as INode. See [Class INode](#) for details. Specifically see the methods related to INodeTransformed in [INode - INodeTransformed methods](#). All methods of this class are implemented by the system.

## Data Members:

public:

**INode \*node;**

The original INode pointer.

**Matrix3 tm;**

The additional transformation.

**BOOL deleteMe;**

If set to FALSE this **INodeTransformed** will not get deleted. This may be used if the object is not allocated dynamically. If you create an

**INodeTransformed** on the stack you'll want to set **deleteMe** to FALSE.

## Methods:

### Prototype:

**void DisposeTemporary();**

### Remarks:

Deletes this **INodeTransformed**.

### Prototype:

**INode \*GetActualINode()**

### Remarks:

Returns the actual **INode** pointer of this **INodeTransformed**.

These functions are not part of this class but are available for use:  
INodeTransformed can be allocated on the stack, but if you need to create one dynamically, use these methods to create and delete them.

**Prototype:**

**INodeTransformed \*CreateINodeTransformed(INode \*n, Matrix3 tm, BOOL dm=TRUE);**

**Remarks:**

Creates an INodeTransformed on the heap.

**Parameters:**

**INode \*n**

The original INode pointer.

**Matrix3 tm**

The additional transformation matrix.

**BOOL dm=TRUE**

If TRUE this item will be deleted; otherwise it is left alone.

**Return Value:**

A pointer to the INodeTransformed created.

**Prototype:**

**void DeleteINodeTransformed(INodeTransformed \*n);**

**Remarks:**

Deletes the INodeTransformed passed.

**Parameters:**

**INodeTransformed \*n**

The INodeTransformed to delete.

## Class MaxNetManager

See Also: [Class MaxNet](#), [Class MaxNetCallBack](#), [Structure ManagerInfo](#), [Structure ClientInfo](#), [Structure JobsList](#), [Structure Job](#), [Class CJobText](#), [Structure HSERVER](#), [Structure JOBFRAMES](#), [Structure JobServer](#), [Structure ServerList](#), [Structure WeekSchedule](#), [Structure NetworkStatus](#)

```
class MaxNetManager: public MaxNet
```

### Description:

This class is available in release 4.0 and later only.

The MaxNetManager class provides all the methods to interact with the network rendering functions provided and acts as your primary interface. The API provided through the MaxNetManager allows clients to connect to the Network Rendering Manager and perform any and all functions available. It encapsulates all the networking details leaving the client code to concentrate on whatever it needs to do. The API handles all networking code and the intricacies of the communication protocols used by the lower layers. This class is derived from the MaxNet class which is solely used for exception handling as shown below.

### Sample Code:

```
try {  
    // the code being tried  
} catch (MaxNet* maxerr) {  
    // handle the error  
    // do NOT delete maxerr  
    // use maxerr->GetErrorText() to get the error description  
    // use maxerr->GetError() to get the error code  
}
```

### Methods Groups:

Construction and Destruction

Callback Methods

Session Methods

Queue Control

Client Related Methods  
Job Related Methods  
Server Methods  
Server Group Methods  
Network Archiving Functions

## **Construction and Destruction**

The following global functions are not part of class MaxNetManager but are available for use:

### **Function:**

**MaxNetManager\* CreateManager();**

### **Remarks:**

This method will create and return a new instance of the MaxNetManager class.

### **Function:**

**void DestroyManager(MaxNetManager\* mgr);**

### **Remarks:**

This method will destroy an instance of the MaxNetManager class.

### **Parameters:**

**MaxNetManager\* mgr**

Points to the MaxNetManager object to destroy.

### **Methods:**

## **Callback Methods**

### **Prototype:**

**void SetCallBack(MaxNetCallBack\* cb);**

### **Remarks:**

Sets a callback method to receive information updates about various

asynchronous events from the MaxNet API. It is not required to set up this callback.

**Parameters:**

**MaxNetCallBack\* mgr**

Points to a MaxNetCallBack object.

## Session Methods

**Prototype:**

```
bool FindManager(short port, char* manager, char* netmask = "255.255.255.0" );
```

**Remarks:**

This method will broadcast a message to the local area network in order to look for a Manager. If a Manager is found, the method returns its name in **manager**.

**Parameters:**

**short port**

Specifies which port will be used to access the Manager. Unless there is a specific reason to use a particular port, use the default **DF\_MGRPORT**.

**char\* manager**

A string representing the name of the Manager will be put into this variable. The variable itself should be **MAX\_PATH** in size.

**char\* netmask**

Specifies which local area network mask should be used for the scope of the broadcast. The default of 255.255.255.0 should work for most networks that are not divided into subnets.

**Return Value:**

TRUE if a Manager is found, otherwise FALSE.

**Prototype:**

```
void Connect(short port, char* manager, bool enable_callback = false );
```

**Remarks:**

This method allows you to connect to a Network Rendering Manager.

**Parameters:****short port**

Specifies which port will be used to access the Manager. Unless there is a specific reason to use a particular port, use the default **DF\_MGRPORT**.

**char\* manager**

A string representing the name or IP number of the Manager you want to connect to. The network name requires the network to have some form of name to address translation. It is recommended to use the name provided by **MaxNetManager::FindManager()**.

**bool enable\_callback = false**

This parameter enables or disables asynchronous messages from the Manager. If you connect to the Manager in order to collect information about the network queue then it is recommended to enable (set to true) this parameter in order to keep your lists updated. Otherwise you have to poll the Manager at various times to check for changes. When enabled calls will be received through the **MaxNetCallback** mechanism informing you whenever a new job has been completed, an error has occurred, a Server has changed, etc.

**Prototype:**

```
void Disconnect();
```

**Remarks:**

This method will cause a disconnect from the currently connected Manager.

**Prototype:**

```
void GetManagerInfo(ManagerInfo* info);
```

**Remarks:**

This method allows basic information about the Manager to be collected.

**Parameters:****ManagerInfo\* info**

A pointer to the Manager information.

**Prototype:****bool KillManager();****Remarks:**

This method will shut down the Manager and shut off the entire system.

**Return Value:**

TRUE if successful, otherwise FALSE. A reason for failure might be because this method is called without having the proper Manager rights such as operating in read only mode. Further explanation can be found in the **TakeManagerControl()** method.

**Prototype:****void EnableUpdate(bool enable);****Remarks:**

This method toggles updates from the Manager. For this method to function you will need to enable the **enable\_callback** in **MaxNetManager::Connect()**. The use of this method allows you to temporarily disable updates from the Manager which might be useful in a situation when you are submitting many jobs at once or executing any other large number of changes. Instead of receiving updates for all changes, you would temporarily disable the callbacks so you could execute your many changes and when done, re-enable the callbacks.

**Parameters:****bool enable**

TRUE or FALSE to enable or disable, respectively.

## Queue Control

**Prototype:****bool QueryManagerControl(bool wait);****Remarks:**

This method queries the Manager to check if you can take control of the queue. If no one has the queue control, it will immediately return true. If someone else has control, the Manager will ask the controlling client if it

wants to relinquish control. If relinquished, the method returns true, if control is not relinquished the method will return false.

**Parameters:**

**bool wait**

This parameter can be set to true in order to wait for an answer in case someone has control over the queue, causing this method not to return until it receives an answer from the controlling client. If there is no response from the controlling client, the method will time out in 10 seconds after which it will return true to allow a request for queue control.

**Return Value:**

TRUE if allowed to request queue control, otherwise FALSE.

**Prototype:**

**bool TakeManagerControl();**

**Remarks:**

This method allows the acquisition of control of the queue.

**Return Value:**

TRUE if control is granted, otherwise FALSE.

**Prototype:**

**void GrantManagerControl(bool grant);**

**Remarks:**

This method allows you to issue a grant or deny response to a MaxNetCallBack message. If you are the controlling client while another client wants control of the queue by calling the **QueryManagerControl()** method you will receive a message through the **MaxNetCallBack** mechanism. If you do not respond, control will be taken away from you automatically after 10 seconds and granted to the requesting client.

**Parameters:**

**bool grant**

This parameter allows you to respond TRUE to relinquish control of the queue or FALSE if you do not want to relinquish control.

**Prototype:****bool LockControl(bool lock);****Remarks:**

This method allows you to temporarily lock the queue control when performing a series of changes and don't want to get interrupted. While the queue control is locked, no queries are made. They all return false to prevent anyone from taking control of the queue. This method can only be called if you already have queue control. Please, do not forget to unlock the queue control after you are done performing your changes.

**Parameters:****bool lock**

Set this parameter to TRUE if you want to lock the queue or FALSE if you want to unlock the queue.

**Return Value:**

TRUE if queue control could be locked, otherwise FALSE.

**Client Related Methods****Prototype:****int GetClientCount();****Remarks:**

This method returns the number of clients currently connected to the Manager.

**Prototype:****int ListClients(int start, int end, ClientInfo\* clientList);****Remarks:**

This method allows you to list all the clients currently connected to the Manager.

**Parameters:****int start**

The first client in the list to return.

**int end**

The last client in the list to return. If you want the entire list of clients at once set the **start** and **end** to 0 and -1, respectively.

### **ClientInfo\* clientList**

The array to receive the list of clients. This array should be large enough to receive the number of clients requested.

#### **Return Value:**

The actual number of clients inserted in **clientList**. This could be either equal to the amount requested or less (if some client disconnected from the Manager).

## **Job Related Methods**

#### **Prototype:**

**int GetJobCount();**

#### **Remarks:**

This method returns the number of jobs in the queue.

#### **Prototype:**

**int ListJobs(int start, int end, JobList\* jobList);**

#### **Remarks:**

This method allows you to list all the jobs in the queue.

#### **Parameters:**

##### **int start**

The first job in the list to return.

##### **int end**

The last job in the list to return. If you want the entire list of jobs at once set the **start** and **end** to 0 and -1, respectively.

##### **JobList\* jobList**

The array to receive the list of jobs. This array should be large enough to receive the number of jobs requested.

#### **Return Value:**

The actual number of jobs inserted in **jobList**. This could be either equal to

the amount requested or less.

**Prototype:**

**void GetJob(HJOB hJob, Job\* job);**

**Remarks:**

This method allows you to get an individual job description structure.

**Parameters:**

**HJOB hJob**

The job handle.

**Job\* job**

A pointer to a Job structure for the received job.

**Prototype:**

**void GetJob(HJOB hJob, JobList\* jobList);**

**Remarks:**

This method allows you to get a single record for the jobList given the specified job handle.

**Parameters:**

**HJOB hJob**

The job handle.

**JobList\* jobList**

A pointer to a JobList structure for the received job.

**Prototype:**

**void GetJobText(HJOB hJob, CJobText& jobText, int count);**

**Remarks:**

Use this method to request the CJobText for a particular job. (See the CJobText class description for an explanation). Some job information are random both in number as they are in size. The job description structure (Job) will only show static elements. In order to get dynamic elements or elements with variable length, the CJobText class is used.

**Parameters:**

**HJOB hJob**

The job handle.

**CJobText& jobText**

A reference to a CJobText class to receive the information.

**int count**

The number of elements you are interested in receiving. The number of elements can be found in Job.jobtextcount.

**Prototype:**

```
void SetJob(HJOB hJob, Job* job, CJobText& jobText, bool  
reset);
```

**Remarks:**

This method submits changes to an existing job. Once you collect a job, you can change settings in both the Job structure as in the CJobText elements and send it back so the changes can be applied.

Note that you must use a **Getjob()** and **SetJob()** combination in order to make sure all the elements are correct. The Manager will automatically complete some of the structure members. It is not possible to create a new structure, fill in the data, and submit it. You can only do that when submitting a new job, which is then handled by a different set of methods.

**Parameters:****HJOB hJob**

The job handle.

**Job\* job**

A pointer to the job description.

**CJobText& jobText**

A reference to a CJobText class with the description of elements.

**bool reset**

This flag indicates whether or not the job is started from scratch. If set to FALSE it will continue from the current stage.

**Prototype:**

```
int GetJobPriority(HJOB hJob);
```

**Remarks:**

This method returns the job priority value for the specified Job.

**Parameters:****HJOB hJob**

The handle to the job for which to obtain its priority

**Prototype:**

**bool SetJobPriority(HJOB hJob, int priority);**

**Remarks:**

This method allows you to set the priority of a specified job.

**Parameters:****HJOB hJob**

The handle to the job for which to obtain its priority

**int priority**

The priority value you want to assign to the job.

**Return Value:**

TRUE if the priority as set successfully, otherwise FALSE

**Prototype:**

**void SetJobOrder(HJOB\* hJob, DWORD count);**

**Remarks:**

This method allows you to set the job order for a specific job.

**Parameters:****HJOB hJob**

The handle to the job for which to obtain its priority

**DWORD count**

The job order index.

**Prototype:**

**void DeleteJob(HJOB hJob);**

**Remarks:**

This method will delete a job from the queue permanently and remove all files

related to the job.

**Parameters:**

**HJOB hJob**

The job handle.

**Prototype:**

**void SuspendJob(HJOB hJob);**

**Remarks:**

This method will suspend a specific job. This method is the opposite of **ActivateJob()**.

**Parameters:**

**HJOB hJob**

The job handle

**Prototype:**

**void ActivateJob(HJOB hJob);**

**Remarks:**

This method will activate a specific job. This method is the opposite of **SuspendJob()**.

**Parameters:**

**HJOB hJob**

The job handle

**Prototype:**

**int GetJobServersCount(HJOB hJob);**

**Remarks:**

This method will return the number of Servers assigned to a given job.

**Parameters:**

**HJOB hJob**

The job handle

**Prototype:**

```
int GetJobServers(int start, int end, HJOB hJob, JobServer* servers);
```

**Remarks:**

This method will return a list of the Servers assigned to a given job.

**Parameters:****int start**

The first Server in the list to return.

**int end**

The last Server in the list to return. If you want the entire list of servers at once set the **start** and **end** to 0 and -1, respectively.

**HJOB hJob**

The job handle.

**JobServer\* servers**

The array to receive the list of Servers. This array should be large enough to receive the number of Servers requested.

**Return Value:**

The actual number of Servers inserted in **servers**. This could be either equal to the amount requested or less.

**Prototype:**

```
void GetJobServerStatus(HJOB hJob, HSERVER hServer, TCHAR* status_text);
```

**Remarks:**

This method allows you to obtain the textual status of a specific Server for a given job. The JobServer structure return by **GetJobServers()** will only describe the status of a Server using a flag. If the flag shows the Server status is "Error", you can use this method to receive a more descriptive text message, like "Could not write to d:/path/file.tga".

**Parameters:****HJOB hJob**

The job handle.

**HSERVER hServer**

The Server handle.

**TCHAR\* status\_text**

A pointer to a string to receive the message. This string should be **MAX\_PATH** long.

**Prototype:**

**void SuspendJobServer(HJOB hJob, HSERVER hServer);**

**Remarks:**

This method will suspend a specific Server for a given job. The Server will stop working with the given job and start working on another one, provided another job exists. Use the **AssignJobServer()** to reactive it.

**Parameters:**

**HJOB hJob**

The job handle.

**HSERVER hServer**

The Server handle.

**Prototype:**

**void AssignJobServer(HJOB hJob, HSERVER hServer);**

**Remarks:**

This method can be used to assign a Server to a given job.

**Parameters:**

**HJOB hJob**

The job handle.

**HSERVER hServer**

The Server handle.

**Prototype:**

**int GetJobFramesCount(HJOB hJob);**

**Remarks:**

This method will return the number of frames for a given job.

**Parameters:**

## **HJOB hJob**

The job handle.

### **Prototype:**

```
int GetJobFrames(int start, int end, HJOB hJob, JOBFAMES* frames);
```

### **Remarks:**

This method will return a list of frames for a given job.

### **Parameters:**

#### **int start**

The first frame in the list to return.

#### **int end**

The last frame in the list to return. If you want the entire list of frames at once set the **start** and **end** to 0 and -1, respectively.

#### **HJOB hJob**

The job handle.

#### **JOBFAMES\* frames**

The array to receive the list of frames with information for each individual frame. This array should be large enough to receive the number of frames requested.

### **Return Value:**

The actual number of frames inserted in **frames**. This could be either equal to the amount requested or less.

### **Prototype:**

```
int GetJobLog(int start, int count, HJOB hJob, TCHAR** buffer);
```

### **Remarks:**

This method will return the log file for a given job.

### **Parameters:**

#### **int start**

The first log file line to return (base zero, 0 is the first line).

**int count**

The number of lines (rows) to return. If you want the whole file at once, set **start** to 0 and **count** to -1. Alternatively, if you want any lines added since the last time you call, set **start** to the last line you collected + 1 and **count** to -1.

**HJOB hJob**

The job handle.

**TCHAR\*\* buffer**

A pointer to a TCHAR array to receive the lines of the log file. The buffer will be allocated so it will accommodate the incoming data. This array should be freed using **LocalFree()** when you are finished.

**Return Value:**

The size of the allocated buffer. This is the size of the entire buffer including the last NULL terminating byte. If **GetJobLog()** returns 0, it means there are no new log file lines available.

**Prototype:**

**bool CheckOutputVisibility(TCHAR\* output, TCHAR\* err);**

**Remarks:**

When submitting a job, you can use this method to find out if the Manager can write a given output image file. This is usually the case when you have the output image file set to a local drive. The other participants in the network rendering may not be able to "see" this path and they will eventually fail. This test is not guaranteed as the Servers may have a different set of rights than the Manager, in which case the Manager might fail the write test while the Servers would have no problem otherwise.

**Parameters:****TCHAR\* output**

The output image file name you want to check. This is the full path and filename (i.e. d:/badpath/file.tga).

**TCHAR\* err**

A TCHAR string to receive the error message if one exists. This will explain why the test failed (such as path not found, access denied, etc.)

**Return Value:**

TRUE if the Manager could write to the given path, otherwise FALSE.

**Prototype:**

```
void AssignJob(Job* job, TCHAR* archive, HSERVER* servers,  
CJobText& jobtext, DWORD blocksize = 0);
```

**Remarks:**

This method allows you to assign a new job to the network queue.

**Parameters:**

**Job\* job**

The job structure containing the information about the job. See the Job structure description for an explanation.

**TCHAR\* archive**

The full path and filename of the archive containing the job files. This is the "\*.maz" file created by the **Maz()** function in the API.

**HSERVER\* servers**

An array containing the Servers assigned to this job. If the job flag is set to "use all Servers" and **job.servercount** is zero, this argument can be NULL (ignored). Otherwise it should be an array **job.servercount \* sizeof(HSERVERS)** long with the list of Servers to assign to this job.

**CJobText& jobtext**

A reference to a CJobText class with the proper elements.

**DWORD blocksize**

An optional alternate block size to use for network transfers. If **blocksize** is set to zero, the API will use the default **DF\_READCHUNK**. You may want to set this to something smaller if you are running over slow connections such as a modem connection. You may want to make it larger if you have a high performance network. This number will determine how large of a block of data to send at once to the Manager.

## **Server Methods**

**Prototype:**

```
int GetServerCount();
```

**Remarks:**

This method will return the number of Servers registered with the Manager.

**Prototype:**

```
int ListServers(int start, int end, ServerList* serverList);
```

**Remarks:**

This method allow you to list the Servers registered with the Manager.

**Parameters:****int start**

The first Server to return.

**int end**

The last Server to return. If you want the whole list at once, set **start** to 0 and **end** to -1.

**ServerList\* serverList**

The array to receive the list of Servers. This array should be large enough to receive the number of Servers requested.

**Return Value:**

The actual number of Servers inserted in **serverList**. This could be either equal to the amount requested or less.

**Prototype:**

```
void GetServer(HSERVER hServer, ServerList* serverList);
```

**Remarks:**

This method allows you to get a Server and retrieve a single record in the serverList given the Server handle.

**Parameters:****HSERVER hServer**

The handle to the Server.

**ServerList\* serverList**

A pointer to the ServerList in which to retrieve the Server.

**Prototype:****bool DeleteServer(HSERVER hServer);****Remarks:**

This method allows you to delete a Server from the Manager's Server list. You can not delete an active Server (i.e. a Server which is currently working on a job).

**Parameters:****HSERVER\* hServer**

The Server handle.

**Return Value:**

TRUE if the server is successfully deleted, otherwise FALSE.

**Prototype:****bool ResetServerIndex(HSERVER hServer);****Remarks:**

The Manager keeps a performance index for each Server. This index is computed based on the Server performance while rendering frames. All factors are taken into consideration such as the time it takes to load a job, the time it takes to process requests, the time it takes to render a frame, the memory and CPU load, etc. This index is in turn used internally to determine the best distribution of workload. You can use this method to reset a Server's performance index.

**Parameters:****HSERVER\* hServer**

The Server handle.

**Return Value:**

TRUE if resetting the Server's performance index was successful, otherwise FALSE.

**Prototype:****void GetWeekSchedule(HSERVER hServer, WeekSchedule\* schedule);****Remarks:**

This method allows you to obtain a given Server's weekly schedule structure.

**Parameters:**

**HSERVER\* hServer**

The Server handle.

**WeekSchedule\* schedule**

A pointer to a WeekSchedule structure to receive the schedule.

**Prototype:**

```
void SetWeekSchedule(HSERVER hServer, WeekSchedule*  
schedule);
```

**Remarks:**

This method allows you to set a given Server's weekly schedule structure.

**Parameters:**

**HSERVER\* hServer**

The Server handle.

**WeekSchedule\* schedule**

A pointer to a WeekSchedule structure with the new weekly schedule.

**Prototype:**

```
void GetServerNetStat(HSERVER hServer, NetworkStatus*  
net_stat);
```

**Remarks:**

This method allows you to obtain the network status for a given server. This method mostly serves as means to check network diagnostics.

**Parameters:**

**HSERVER\* hServer**

The Server handle.

**NetworkStatus\* net\_stat**

A pointer to a NetworkStatus structure to receive the status data.

## **Server Group Methods**

**Prototype:**

**int GetServerGroupCount();**

**Remarks:**

This method will return the number of Server groups.

**Prototype:**

**int GetServerGroupXCount(int group);**

**Remarks:**

This method will return the number of Servers for a given Server group.

**Parameters:**

**int group**

The zero based index into the Server group list.

**Prototype:**

**int GetServerGroup(int group, int count, HSERVER\* grplist,  
TCHAR\* name);**

**Remarks:**

This method allows you to obtain a Server group.

**Parameters:**

**int group**

The zero based index into the Server group list.

**int count**

The number of Servers to send, in order to define the size of the **grplist**.

**HSERVER\* grplist**

The array in which the list of Servers will be returned. This array should be large enough to accommodate **count** Servers.

**TCHAR\* name**

The name of the Server group. This string must be at least **MAX\_PATH** long.

**Return Value:**

The number of Servers collected.

**Prototype:**

```
void NewServerGroup(int count, HSERVER* grplist, TCHAR* name);
```

**Remarks:**

This method allows you to submit (create) a new Server group.

**Parameters:**

**int count**

The number of Servers in the list.

**HSERVER\* grplist**

The array containing **count** Servers.

**TCHAR\* name**

The name of the server group.

**Prototype:**

```
void DeleteServerGroup(int group);
```

**Remarks:**

This method allows you to delete a given Server group.

**Parameters:**

**int group**

The zero based index of the Server group to delete.

## **Network Archiving Functions**

The following global functions are not part of class MaxNetManager but are available for use:

**Function:**

```
bool Maz(TCHAR* archivename, TCHAR* file_list, DWORD* filesize = 0);
```

**Remarks:**

This function creates a Network Rendering archive. This is the archive sent to the manager when submitting a new job. Note that even though you can use whatever name you feel like, the Manager and Servers will look for a

"jobname.maz" file.

**Parameters:**

**TCHAR\* archivename**

The full path and filename of the archive you want to create.

**TCHAR\* file\_list**

A list of NULL terminated filenames to include in the archive. You should provide a full path and filename. However, only the file names will be saved in the archive and all files are going to be un-archived in the same directory.

An example: `file_list[] = {"c:\\path\\file.maz\0c:\\anotherpath\\maps.tga\0\0"};`

**DWORD\* filesize**

Optional pointer to a variable to receive the accumulated size of all files included in the archive. This is the "uncompressed" size. You can use this to compute the disk space necessary to uncompress the archive.

**Return Value:**

TRUE if the archive was successfully created, otherwise FALSE.

**Function:**

**bool UnMaz(TCHAR\* archivename, TCHAR\* output\_path);**

**Remarks:**

This function is the opposite of **Maz()** and will un-archive the given archive into the specified directory.

**Parameters:**

**TCHAR\* archivename**

The full path and filename of the archive you want to un-archive.

**TCHAR\* output\_path**

The path you want the files extracted to.

**Return Value:**

TRUE if the archive was successfully extracted, otherwise FALSE.

**Function:**

**void jobSetJobDefaults(Job\* job);**

**Remarks:**

This function will set the default values for the given Job structure. You can use this function to prevent frequent resetting of the structure and its default fields such as size, version, etc.

**Parameters:**

**Job \*job**

The job structure containing the information about the job. See the Job structure description for an explanation.

**Function:**

**bool jobReadMAXProperties(char\* filename, Job\* job, CJobText& jobText);**

**Remarks:**

This function will initialize a job structure using the data read from a specified 3ds max scene file. If you would want to submit a job based on a 3ds max file alone, you would call this function passing it to the file specified. This function takes care of filling all the fields so you can turn around and just send the job to the queue. An example of this can be found in the SDK under **\MAXSDK\SAMPLES\NETRENDER\JOBASSIGN**.

**Parameters:**

**char\* filename**

The filename of the 3ds max scene file (\*.max).

**Job\* job**

A pointer to an empty job structure which will be initialized by the method. Because the function will initialize the structure any values present prior to calling this method will be reset.

**CJobText& jobText**

A reference to an empty CJobText class which will be initialized by the function. Just like the job structure, any values present prior to calling this function will be reset.

**Return Value:**

TRUE if reading the properties was successful, otherwise FALSE.

## Class MaxNet

See Also: [Class MaxNetManager](#), [List of MaxNet Errors](#)

class MaxNet

### Description:

This class is available in release 4.0 and later only.

The MaxNet class serves as the base class for MaxNetManager and should be treated as the exception handler when using the Network Rendering API.

### Methods:

#### Prototype:

**MaxNet();**

#### Remarks:

Constructor

#### Prototype:

**maxnet\_error\_t GetError();**

#### Remarks:

This method returns the MaxNet error. See the list of MaxNet error codes for details.

#### Prototype:

**int GetGError();**

#### Remarks:

This method returns the MaxNet error. See the list of MaxNet error codes for details.

#### Prototype:

**const TCHAR\* GetErrorText();**

#### Remarks:

This method returns the MaxNet error description string.

## List of Parameter Block IDs

See Also: Class [Base Object](#) (method **GetParamBlockIndex()**).

Interfaces into some of the standard plug-ins that ship with 3ds max. These are from the file `\MAXSDK\INCLUDE\ISTDPLUG.H`.

The following are parameter block IDs for procedural objects:

### **Obj/Mod ID Data Type**

Arc **ARC\_RADIUS** float

**ARC\_FROM** float

**ARC\_TO** float

**ARC\_PIE** int

**ARC\_REVERSE** int

Box **BOXOBJ\_LENGTH** float

**BOXOBJ\_WIDTH** float

**BOXOBJ\_HEIGHT** float

**BOXOBJ\_WSEGS** int

**BOXOBJ\_LSEGS** int

**BOXOBJ\_HSEGS** int

**BOXOBJ\_GENUVS** int

Circle **CIRCLE\_RADIUS** float

Cone **CONE\_RADIUS1** float

**CONE\_RADIUS2** float

**CONE\_HEIGHT** float

**CONE\_SEGMENTS** int

**CONE\_CAPSEGMENTS** int

**CONE\_SIDES** int

**CONE\_SMOOTH** int

**CONE\_SLICEON** int

**CONE\_PIESLICE1** float

**CONE\_PIESLICE2** float

**CONE\_GENUVS** int

Cylinder **CYLINDER\_RADIUS** float

**CYLINDER\_HEIGHT** float

**CYLINDER\_SEGMENTS** int

**CYLINDER\_CAPSEGMENTS** int  
**CYLINDER\_SIDES** int  
**CYLINDER\_SMOOTH** int  
**CYLINDER\_SLICEON** int  
**CYLINDER\_PIESLICE1** float  
**CYLINDER\_PIESLICE2** float  
**CYLINDER\_GENUVS** int  
Donut **DONUT\_RADIUS1** float  
**DONUT\_RADIUS2** float  
Ellipse **ELLIPSE\_LENGTH** float  
**ELLIPSE\_WIDTH** float  
Hedra **HEDRA\_RADIUS** float  
**HEDRA\_FAMILY** int  
**HEDRA\_P** float  
**HEDRA\_Q** float  
**HEDRA\_SCALEP** float  
**HEDRA\_SCALEQ** float  
**HEDRA\_SCALER** float  
**HEDRA\_VERTS** int  
**HEDRA\_GENUVS** int  
Helix **HELIX\_RADIUS1** float  
**HELIX\_RADIUS2** float  
**HELIX\_HEIGHT** float  
**HELIX\_TURNS** float  
**HELIX\_BIAS** float  
**HELIX\_DIRECTION** int  
Ngon **NGON\_RADIUS** float  
**NGON\_SIDES** int  
**NGON\_CIRCULAR** int  
PatchGrid **PATCHGRID\_LENGTH** float  
**PATCHGRID\_WIDTH** float  
**PATCHGRID\_WSEGS** int  
**PATCHGRID\_LSEGS** int  
**PATCHGRID\_TEXTURE** int

Rain/Snow **RSPART\_VPTPARTICLES** int

**RSPART\_RNDPARTICLES** int

**RSPART\_DROPSIZE** float

**RSPART\_SPEED** float

**RSPART\_DISPTYPE** int

**RSPART\_VARIATION** float

**RSPART\_STARTTIME** int

**RSPART\_LIFETIME** int

**RSPART\_EMITTERWIDTH** float

**RSPART\_EMITTERHEIGHT** float

**RSPART\_HIDEEMITTER** int

**RSPART\_BIRTHRATE** float

**RSPART\_CONSTANT** int

**RSPART\_RENDER** int

**RSPART\_TUMBLE** int

**RSPART\_SCALE** float

Rectangle **RECTANGLE\_LENGTH** float

**RECTANGLE\_WIDTH** float

**RECTANGLE\_FILLET** float

Sphere **SPHERE\_RADIUS** float

**SPHERE\_SEGS** int

**SPHERE\_SMOOTH** int

**SPHERE\_HEMI** float

**SPHERE\_SQUASH** int

**SPHERE\_RECENTER** int

**SPHERE\_GENUVS** int

Star **START\_RADIUS1** float

**START\_RADIUS2** float

**START\_POINTS** int

**START\_DISTORT** float

**START\_FILLET1** float

**START\_FILLET2** float

Teapot **TEAPOT\_RADIUS** float

**TEAPOT\_SEGS** int

**TEAPOT\_SMOOTH** int  
**TEAPOT\_TEAPART** int  
**TEAPOT\_BODY** int  
**TEAPOT\_HANDLE** int  
**TEAPOT\_SPOUT** int  
**TEAPOT\_LID** int  
**TEAPOT\_GENUVS** int  
Text **TEXT\_SIZE** float  
**TEXT\_KERNING** float  
**TEXT\_LEADING** float  
Torus **TORUS\_RADIUS** float  
**TORUS\_RADIUS2** float  
**TORUS\_ROTATION** float  
**TORUS\_TWIST** float  
**TORUS\_SEGMENTS** int  
**TORUS\_SIDES** int  
**TORUS\_SMOOTH** int  
**TORUS\_SLICEON** int  
**TORUS\_PIESLICE1** float  
**TORUS\_PIESLICE2** float  
**TORUS\_GENUVS** int  
Tube **TUBE\_RADIUS** float  
**TUBE\_RADIUS2** float  
**TUBE\_HEIGHT** float  
**TUBE\_SEGMENTS** int  
**TUBE\_CAPSEGMENTS** int  
**TUBE\_SIDES** int  
**TUBE\_SMOOTH** int  
**TUBE\_SLICEON** int  
**TUBE\_PIESLICE1** float  
**TUBE\_PIESLICE2** float  
**TUBE\_GENUVS** int  
Grid **GRIDHELP\_LENGTH** float  
**GRIDHELP\_WIDTH** float

**GRIDHELP\_GRID** float

The following are parameter block IDs for modifiers:

Bend **BEND\_ANGLE** float

modifier **BEND\_DIR** float

**BEND\_AXIS** int

**BEND\_DOREGION** int

**BEND\_FROM** float

**BEND\_TO** float

Bomb **BOMB\_STRENGTH** float

space warp **BOMB\_GRAVITY** float

**BOMB\_CHAOS** float

**BOMB\_DETONATION** int

Deflector **DEFLECTOR\_BOUNCE** float

space warp **DEFLECTOR\_WIDTH** float

**DEFLECTOR\_HEIGHT** float

Displace **DISPLACE\_MATYPE** int

(modifier and **DISPLACE\_UTILE** float

space warp **DISPLACE\_VTILE** float

object) **DISPLACE\_WTILE** float

**DISPLACE\_BLUR** float

**DISPLACE\_USEMAP** int

**DISPLACE\_APPLYMAP** int

**DISPLACE\_STRENGTH** float

**DISPLACE\_DECAY** float

**DISPLACE\_CENTERLUM** int

**DISPLACE\_UFLIP** int

**DISPLACE\_VFLIP** int

**DISPLACE\_WFLIP** int

**DISPLACE\_CENTERLUM** float

**DISPLACE\_CAP** int

**DISPLACE\_LENGTH** float

**DISPLACE\_WIDTH** float

**DISPLACE\_HEIGHT** float

**DISPLACE\_AXIS** int

Extrude **EXTRUDE\_AMOUNT** float  
modifier **EXTRUDE\_SEGS** int  
**EXTRUDE\_CAPSTART** int  
**EXTRUDE\_CAPEND** int  
**EXTRUDE\_CAPTYPE** int  
**EXTRUDE\_OUTPUT** int  
**EXTRUDE\_MAPPING** int  
Gravity **GRAVITY\_STRENGTH** float  
space warp **GRAVITY\_DECAY** float  
**GRAVITY\_TYPE** int  
**GRAVITY\_DISPLENGTH** float  
Wind **WIND\_STRENGTH** float  
space warp **WIND\_DECAY** float  
**WIND\_TYPE** int  
**WIND\_DISPLENGTH** float  
**WIND\_TURBULENCE** float  
**WIND\_FREQUENCY** float  
**WIND\_SCALE** float  
UVW map **UVWMAP\_MAPTYPE** int  
modifier **UVWMAP\_UTILE** float  
**UVWMAP\_VTILE** float  
**UVWMAP\_WTILE** float  
**UVWMAP\_UFLIP** int  
**UVWMAP\_VFLIP** int  
**UVWMAP\_WFLIP** int  
**UVWMAP\_CAP** int  
**UVWMAP\_CHANNEL** int  
**UVWMAP\_LENGTH** float  
**UVWMAP\_WIDTH** float  
**UVWMAP\_HEIGHT** float  
**UVWMAP\_AXIS** int  
Noise **NOISEMOD\_SEED** int  
modifier **NOISEMOD\_SCALE** float  
**NOISEMOD\_FRACTAL** int

**NOISEMOD\_ROUGH** float  
**NOISEMOD\_ITERATIONS** float  
**NOISEMOD\_ANIMATE** int  
**NOISEMOD\_FREQ** float  
**NOISEMOD\_PHASE** int  
**NOISEMOD\_STRENGTH** Point3

Optimize **OPTMOD\_RENDER** int  
modifier **OPTMOD\_VIEWS** int  
**OPTMOD\_FACETHRESH1** float  
**OPTMOD\_EDGETHRESH1** float  
**OPTMOD\_BIAS1** float  
**OPTMOD\_PRESERVEMAT1** int  
**OPTMOD\_PRESERVESMOOTH1** int  
**OPTMOD\_FACETHRESH2** float  
**OPTMOD\_EDGETHRESH2** float  
**OPTMOD\_BIAS2** float  
**OPTMOD\_PRESERVEMAT2** int  
**OPTMOD\_PRESERVESMOOTH** int  
**OPTMOD\_MAXEDGE2** float  
**OPTMOD\_AUTOEDGE** int  
**OPTMOD\_MANUPDATE** int

Volume **VOLSEL\_LEVEL** int  
selection **VOLSEL\_METHOD** int  
modifier **VOLSEL\_TYPE** int  
**VOLSEL\_VOLUME** int  
**VOLSEL\_INVERT** int

Ripple/Wave **RWAVE\_AMPLITUDE** float  
space warp **RWAVE\_AMPLITUDE2** float  
object and **RWAVE\_WAVELEN** float  
object space **RWAVE\_PHASE** float  
modifier **RWAVE\_DECAY** float

(Note: These next three are only valid for space warp objects).

**RWAVE\_CIRCLES** int  
**RWAVE\_SEGMENTS** int

**RWAVE\_DIVISIONS** int  
Ripple/Wave **RWAVE\_FLEX** float  
binding (modifier)  
Skew **SKEW\_AMOUNT** float  
modifier **SKEW\_DIR** float  
**SKEW\_AXIS** int  
**SKEW\_DOREGION** int  
**SKEW\_FROM** float  
**SKEW\_TO** float  
Material **MATMOD\_MATID** int  
modifier  
Smoothing **SMOOTHMOD\_AUTOSMOOTH** int  
group **SMOOTHMOD\_THRESHOLD** float  
modifier **SMOOTHMOD\_SMOOTHBITS** int  
Normal **NORMMOD\_UNIFY** int  
modifier **NORMMOD\_FLIP** int  
SurfRev **SURFREV\_DEGREES** float  
modifier **SURFREV\_SEGS** int  
**SURFREV\_CAPSTART** int  
**SURFREV\_CAPEND** int  
**SURFREV\_CAPTYPE** int  
**SURFREV\_WELDCORE** int  
**SURFREV\_OUTPUT** int  
**SURFREV\_MAPPING** int  
Taper **TAPER\_AMT** float  
modifier **TAPER\_CRV** float  
**TAPER\_AXIS** int  
**TAPER\_EFFECTAXIS** int  
**TAPER\_SYMMETRY** int  
**TAPER\_DOREGION** int  
**TAPER\_FROM** float  
**TAPER\_TO** float  
Twist **TWIST\_ANGLE** float  
modifier **TWIST\_BIAS** float

**TWIST\_AXIS** int

**TWIST\_DOREGION** int

**TWIST\_FROM** float

**TWIST\_TO** float

Material **MATMOD\_MATID** int

modifier

Smooth **SMOOTH\_AUTOSMOOTH** int

modifier **SMOOTH\_THRESHOLD** float

**SMOOTH\_SMOOTHBITS** int

Normal **NORMALMOD\_UNIFY** int

modifier **NORMALMOD\_FLIP** int

Tessellation **TESSMOD\_TYPE** int

modifier **TESSMOD\_TENSION** float

**TESSMOD\_ITERATIONS** int

**TESSMOD\_FACE\_TYPE** int

UVW XForm **UVWXFORM\_UTILE** float

modifier **UVWXFORM\_VTILE** float

**UVWXFORM\_WTILE** float

**UVWXFORM\_UOFFSET** float

**UVWXFORM\_VOFFSET** float

**UVWXFORM\_WOFFSET** float

**UVWXFORM\_UFLIP** int

**UVWXFORM\_VFLIP** int

**UVWXFORM\_WFLIP** int

**UVWXFORM\_CHANNEL** int

# Class Deformer

See Also: [Class Object](#), [Class Point3](#).

class Deformer

## Description:

This is the callback object used by modifiers to deform "Deformable" objects.

## Methods:

## Prototype:

**virtual Point3 Map(int i, Point3 p) = 0;**

## Remarks:

Implemented by the Plug-In.

This is the method that is called to deform or alter a single point. Note that this method needs to be thread safe. A problem may occur when a non-local variable is modified inside of Map(). Since two versions of Map() could be executing at the same time, they could both end up modifying the same variable simultaneously which usually causes problems. See the Advanced Topics section [Thread Safe Plug-Ins](#) for more details.

## Parameters:

### int i

The index of the point to be altered. Note: An index of -1 may be passed. This indicates that the deformer is not being applied to a regular object but instead points that are generated on the fly for display purposes.

### Point3 p

The point to be altered.

## Return Value:

The altered point.

## Class GPort

See Also: Advanced Topics on [Palettes](#), [Class Color](#), [Class GammaMgr](#), [COLORREF](#).

class GPort

### Description:

A useful utility class for managing user interface colors. This class has several purposes:

- Maintain the default 3ds max palette for doing 256 color graphics.

- Provides a mechanism for allocating "animated color slots" in the default palette for use in the user interface.

- Provide various functions for doing dithered graphics using the default 3ds max palette.

All methods of this class are implemented by the system.

The following global function is used to get a pointer that may be used to call the methods of this class:

### Prototype:

```
GPort* GetGPort();
```

### Remarks:

There is only a single global instance of this class, and this method returns a pointer to it. A developer may use this pointer to call the methods of **GPort**.

### Sample Code:

```
int animColSlot = GetGPort()->GetAnimPalSlot();
```

### Methods:

#### Prototype:

```
virtual int GetAnimPalSlot()=0;
```

### Remarks:

Returns a slot number if available, -1 if not. Typically this is called in **WM\_INITDIALOG** processing code for as many slots as you need (the total number available is 8).

### Return Value:

A slot number if available; otherwise -1.

**Prototype:**

**virtual int AnimPalIndex(int i)=0;**

**Remarks:**

Retrieves the palette index associated with the 'i-th' slot.

**Parameters:**

**int i**

Specifies the slot.

**Prototype:**

**virtual void ReleaseAnimPalSlot(int i)=0;**

**Remarks:**

Releases the specified animated palette slot. Typically this is called in the **WM\_DESTROY** code for each slot obtained with **GetAnimPalSlot()**.

**Parameters:**

**int i**

The palette slot to release.

**Prototype:**

**virtual void SetAnimPalEntry(int i, COLORREF cr)=0;**

**Remarks:**

Sets the color associated with the 'i-th' animated slot.

**Parameters:**

**int i**

The slot index.

**COLORREF cr**

The color to set.

**Prototype:**

**virtual void AnimPalette(HDC hdc)=0;**

**Remarks:**

After several calls to **SetAnimPalEntry()**, call this to affect the HDC's palette. This puts the palette into the **HDC** so it will take effect.

**Parameters:****HDC hdc**

The handle of the device context.

**Prototype:****virtual HPALETTE PlugPalette(HDC hdc)=0;****Remarks:**

This method puts the standard 3ds max palette into the palette for the HDC, handing back a handle to the old palette.

**Parameters:****HDC hdc**

The device context.

**Return Value:**

The handle of the old palette.

**Prototype:****virtual void RestorePalette(HDC hDC,HPALETTE hOldPal)=0;****Remarks:**

This method puts the previous 3ds max palette (returned from **PlugPalette()**) into the palette for the HDC.

**Parameters:****HDC hDC**

The handle of the device context.

**HPALETTE hOldPal**

The palette to restore.

**Prototype:****virtual HBRUSH MakeAnimBrush(int slotNum, COLORREF col)=0;**

**Remarks:**

This method creates a brush for drawing with the specified animated palette slot color.

**Parameters:****int slotNum**

The animated palette slot.

**COLORREF col**

The color to use.

**Return Value:**

The handle of the brush created.

**Prototype:**

```
virtual int UpdateColors(HDC hdc)=0;
```

**Remarks:**

This method calls the Windows API **UpdateColors()** on the specified device context handle **hdc**. Call this when you get a **WM\_PALETTECHANGED** message.

**Parameters:****HDC hdc**

The handle of the device context.

**Return Value:**

Nonzero if it changed screen pixel values; otherwise zero.

**Prototype:**

```
virtual void MapPixels(UBYTE* inp, UBYTE *outp, int x, int y,  
int width)=0;
```

**Remarks:**

This method maps a single row of pixels in 24 bit color to indices into the current GPort palette, applying a dither pattern. This routine does NOT do gamma correction.

Note that **x** and **y** are necessary to establish the dither pattern alignment. The dither pattern covers the entire map, and if you want to just dither part of it, you need to tell this method where you are within the pattern.

**Parameters:****UBYTE\* inp**

Points to an array of width BGR triples. This is a sequence of bytes structured as B,G,R,B,G R, etc. The first pixel is B,G,R then the next pixel is B,G,R, etc.

**UBYTE \*outp**

The result - the color indices into the GPort palette. This array is **width** bytes in length.

**int x**

The x alignment position.

**int y**

The y alignment position.

**int width**

The number of items in the arrays above.

**Prototype:**

```
virtual void DisplayMap(HDC hdc, Rect& drect, int xsrc, int ysrc,  
UBYTE *map, int bytesPerRow)=0;
```

**Remarks:**

Display an array of 24 bit colors in the HDC. If the current display is 8 bit it will display it (with dither) using in the GPort palette, otherwise it will just blit to the screen. This method does NOT do gamma correction.

**Parameters:****HDC hdc**

The handle of the device context.

**Rect& drect**

The destination rectangle in the hdc.

**int xsrc**

The X position within this source raster of the upper left corner of the rectangle to be copied.

**int ysrc**

The Y position within this source raster of the upper left corner of the rectangle to be copied.

**UBYTE \*map**

Points to an array of BGR triples.

**int bytesPerRow**

The number of bytes per row on each scanline of **map**.

**Prototype:**

```
virtual void DisplayMap(HDC hdc, Rect& dest, Rect& src,  
    UBYTE *map, int bytesPerRow)=0;
```

**Remarks:**

This version of **DisplayMap()** stretches the image if the source rectangle is not the same size as the destination rectangle. **src** should be the size of the image.

**Parameters:**

**HDC hdc**

The handle of the device context.

**Rect& dest**

The destination rectangle in the **hdc**.

**Rect& src**

The source rectangle in **map**.

**UBYTE \*map**

Points to an array of RGB triples.

**int bytesPerRow**

The number of bytes per row on each scanline of **map**.

**Prototype:**

```
virtual void DitherColorSwatch(HDC hdc, Rect& r, Color c)=0;
```

**Remarks:**

This method first gamma corrects **Color c** using the current display gamma. In paletted modes, it will fill rectangle **r** with a dithered pattern approximating **Color c**. In 24 bit modes it just fills the rectangle with **Color c**.

**Parameters:**

**HDC hdc**

The handle of the device context.

**Rect& r**

The rectangle to fill.

**Color c**

The color to approximate (8 bit) or fill with (24 bit).

**Prototype:**

**virtual void PaintAnimPalSwatch(HDC hdc, DWORD col, int slot, int left, int top, int right, int bottom)=0;**

**Remarks:**

This method attempts to use the animated color slot indicated by "slot" to paint a rectangular color swatch. If slot is -1, it will uses **DitherColorSwatch()**. This method does handle gamma correction.

**Parameters:****HDC hdc**

The handle of the device context.

**DWORD col**

The color to paint. The format is the same as [COLORREF](#).

**int slot**

Specifies the slot to use. If -1 then **DitherColorSwatch()** is used.

**int left**

The left coordinate of the rectangular area to paint.

**int top**

The top coordinate of the rectangular area to paint.

**int right**

The right coordinate of the rectangular area to paint.

**int bottom**

The bottom coordinate of the rectangular area to paint.

**Prototype:**

**virtual HPALETTE GetPalette()=0;**

**Remarks:**

Returns the current GPort palette.

**The following functions are not part of class GPort but are available for use**

**Prototype:**

**static inline void gammaCorrect(COLORREF& c)**

**Remarks:**

Gamma corrects the specified color using the display gamma setting.

**Parameters:**

**COLORREF& c**

The color to gamma correct.

**Prototype:**

**static inline void deGammaCorrect(COLORREF& c)**

**Remarks:**

Reverses the effect of gamma correction on the specified color.

**Parameters:**

**COLORREF& c**

The color to de-gamma.

# Class ParamDimension

See Also: [Class ParamDimensionBase](#), [List of Dimension Types](#).

## Description:

Any parameter that can be controlled by a controller has a dimension. This dimension can be considered a unit of measure. It describes its type and its order of magnitude. When a controller needs to display the parameter values (for example in the function curve editor) it converts the value using its parameter dimension Convert() function. It can also convert back using the Unconvert() function.

Some parameters are stored over one range of values and displayed in another. For example parameter that use **stdAngleDim** store their parameters in radians but display them in degrees. By using **stdAngleDim** the value is converted to the proper format for display. Some parameter dimensions do not perform any conversion, for example **stdWorldDim**.

There are several default parameter dims implemented. Listed with each one is the type, convert functions and range of values usually stored (these are not enforced in any way).

If the type of parameter for your plug-in does not fit within any of the dimensions listed here you may simply use **defaultDim**. This performs no conversions and has no range.

### **ParamDimension \*defaultDim;**

Convert() is original value.

UnConvert() is original value.

Range: None.

### **ParamDimension \*stdWorldDim;**

The DimType is **DIM\_WORLD**

Convert() return original value.

UnConvert() returns original value.

### **ParamDimension \*stdAngleDim;**

The DimType is **DIM\_ANGLE**.

Convert() is RadToDeg()

UnConvert() is DegToRad()

### **ParamDimension \*stdColorDim;**

The DimType is **DIM\_COLOR**  
Convert() return original value.  
UnConvert() returns original value.  
Range: 0-1

**ParamDimension \*stdColor255Dim;**

The DimType is **DIM\_COLOR255**  
Convert() is value \* 255.0f  
UnConvert() is value / 255.0f  
Range: 0-255

**ParamDimension \*stdPercentDim;**

The DimType is **DIM\_PERCENT**  
Convert() is value \* 100.0f  
UnConvert() is value / 100.0f  
Range: 0-100

**ParamDimension \*stdNormalizedDim;**

The DimType is **DIM\_NORMALIZED**  
Convert() is original value.  
UnConvert() is original value.  
Range: 0-1

**ParamDimension \*stdSegmentsDim;**

The DimType is **DIM\_SEGMENTS**  
Convert() is original value.  
UnConvert() is original value.

**ParamDimension \*stdTimeDim;**

The DimType is **DIM\_TIME**  
Convert() is value/GetTicksPerFrame()  
UnConvert() is value\*GetTicksPerFrame() (see the section [Time](#) for details).

**Methods:**

If the DimType is custom than the methods below must be implemented. Note: This class is derived from ParamDimensionBase which provides methods **DimensionType()**, **Convert()** and **Unconvert()**.

**Prototype:**

**virtual float GetDimScale();**

**Remarks:**

Implemented by the Plug-In.

Returns the dimension scale.

**Prototype:**

**virtual void SetDimScale();**

**Remarks:**

Implemented by the Plug-In.

Sets the dimension scale.

**Prototype:**

**virtual TCHAR \*DimensionName();**

**Remarks:**

Implemented by the Plug-In.

Returns the name of the dimension.

## Class GetParamName

See Also: [ParamDimension](#).

### Description:

This class is used to hold a parameter name. When a client of a parameter block receives the **REFMSG\_GET\_PARAM\_NAME** message, the partID field is set to point at one of these structures. The client should fill in the parameter name.

### Data Members:

**TSTR name;**

Assign the parameter name to this variable.

**int index;**

Index of the parameter in the parameter block.

### Methods:

### Prototype:

**GetParamName(TSTR n, int i)**

### Remarks:

Constructor.

## Class GetParamDim

See Also: [ParamDimension](#).

### Description:

This class is used to store a parameter dimension. When a client of a parameter block receives the **REFMSG\_GET\_PARAM\_DIM** message, the **partID** field is set to point at one of these structures. The client should set **dim** to point at its dim descriptor.

### Data Members:

**ParamDimension \*dim;**

Assign the dimension to this variable.

**int index;**

Index of the parameter in the parameter block.

### Methods:

### Prototype:

**GetParamDim(int i)**

### Remarks:

Constructor.

## Class ParamBlockDescID

See Also: [Class ParamBlockDesc](#).

```
class ParamBlockDescID
```

### Description:

The parameter block descriptor describes each parameter in a parameter block. This version has an ID used to identify each parameter.

```
class ParamBlockDescID {  
    public:  
        ParamType type;  
        UserType *user;  
        BOOL animatable;  
        DWORD id;  
};
```

### Data Members:

#### **ParamType type**

The parameter type. See [List of Parameter Types](#).

#### **UserType \*user**

This value is not currently used -- it must always be passed as NULL.

#### **BOOL animatable**

This is a flag indicating if the parameter may be animated or not. Pass TRUE if the value may be animated and FALSE if it is constant.

#### **DWORD id**

This is an ID used to identify this parameter. This provides a solution to the problem of backwards compatibility. If you alter the parameter structure of your plug-in in the future (by adding or deleting parameters for example) previously saved 3ds max files will be incompatible. You can however use a mechanism which uses these IDs to convert older versions to the current version. See the Advanced Topics section on [Parameter Maps](#) for more detail on how this is done.

## Class ParamUIDesc

See Also: [Class IParamMap](#), [Parameter Maps](#).

### Description:

The ParamUIDesc class is used in conjunction with the parameter maps mechanism. It is used for creating descriptors that define the properties of a user interface control such as its type (spinner, radio button, check box, etc.), which resource ID it refers to, and which index into the virtual array of parameters it uses. See the Advanced Topics section on [Parameter Maps](#) for an overview of how these descriptors are used.

### Methods

### Prototype:

```
ParamUIDesc(int index,  
             EditSpinnerType spinType,int idEdit,int idSpin,  
             float lowLim,float highLim,float scale,  
             ParamDimension *dim=defaultDim);
```

### Remarks:

Constructor. This constructor is used for a float or int controlled by a single spinner control:

### Parameters:

#### **int index**

This is the index into the IParamArray virtual array of this UI control.

#### **EditSpinnerType spinType**

This parameter specifies the type of value which may be entered. The valid types (listed in **CUSTCONT.H**) are:

- **EDITTYPE\_INT** - Any integer value.
- **EDITTYPE\_FLOAT** - Any floating point value.
- **EDITTYPE\_UNIVERSE** - This is a value in world space units. It respects the systems unit settings (for example feet and inches).
- **EDITTYPE\_POS\_INT** - Any integer  $\geq 0$
- **EDITTYPE\_POS\_FLOAT** - Any floating point value  $\geq 0.0$
- **EDITTYPE\_POS\_UNIVERSE** - This is a positive value in world space units. It respects the systems unit settings (for example feet and inches) .

· **EDITTYPE\_TIME** - This is a time value. It respects the system time settings (SMPTE for example).

**int idEdit, int idSpin**

These are the resource IDs the edit control and the spinner control.

**float lowLim**

This is the minimum value the spinner can take on.

**float highLim**

This is the maximum value the spinner can take on.

**float scale**

This is the increment or decrement value used when the user uses the up or down arrow buttons of the spinner control. You may also pass the value **SPIN\_AUTOSCALE**. This causes 3ds max to automatically adjust the value used to increment or decrement based on the current value being edited. This allows the spinner to cover a wider range of values with less mouse movement or button clicking from the user.

**Prototype:**

```
ParamUIDesc(int index,ControlType type,int *ctrlIDs,  
int count,int *vals=NULL);
```

**Remarks:**

Constructor. This constructor is used for an **int** controlled by n radio buttons where

**vals[i]** represents the value if **ctrlIDs[i]** is checked. If **vals=NULL** then **ctrlIDs[i]** represents a value of i.

Or it may be used for:

An **int** controlled by multiple check boxes where each check boxes controls a single bit.

**vals[i]** specifies which bit **ctrlIDs[i]** controls. If **vals=NULL** then **ctrlIDs[i]** controls the i-th bit.

**Parameters:**

**int index**

This is the index into the IParamArray virtual array of this UI control.

**ControlType type**

This specifies the type of control to use. The available control are:

- **TYPE\_RADIO** - Radio Buttons.
- **TYPE\_MULTICHECKBOX** - Multiple Check Boxes. Note: This option is not currently supported.

**int \*ctrlIDs**

An array of control IDs. See the Remarks above.

**int count**

This is the number of control IDs in the array above.

**int \*vals=NULL**

An array of values. See the Remarks above.

**Prototype:**

**ParamUIDesc(int index,ControlType type,int id);**

**Remarks:**

Constructor. This version is used for an int controlled by a single check box (BOOL) or a Point3 controlled by a color swatch.

**Parameters:**

**int index**

This is the index into the IParamArray virtual array of this UI control.

**ControlType type**

This specifies the type of control to use. The available control types (defined in IPARAM.H) are:

- **TYPE\_SPINNER** - Spinner Control.
- **TYPE\_RADIO** - Radio Button.
- **TYPE\_SINGLECHECKBOX** - Single Check Box.
- **TYPE\_MULTICHECKBOX** - Multiple Check Boxes. Note: This option is not currently supported.
- **TYPE\_COLORSWATCH** - Color Swatch.

**int id**

This is the resource ID of the control.

**Prototype:**

**ParamUIDesc(int index,EditSpinnerType spinType,int idEdit1,**

```
int idSpin1,int idEdit2,int idSpin2,int idEdit3,  
int idSpin3,float lowLim,float highLim,float scale,  
ParamDimension *dim=defaultDim);
```

**Remarks:**

Implemented by the System.

This version is for a Point3 controlled by 3 spinners

**Parameters:**

**int index**

This is the index into the IParamArray virtual array of this UI control.

**EditSpinnerType spinType**

This parameter specifies the type of value which may be entered. The valid types (listed in **CUSTCONT.H**) are:

- **EDITTYPE\_INT** - Any integer value.
- **EDITTYPE\_FLOAT** - Any floating point value.
- **EDITTYPE\_UNIVERSE** - This is a value in world space units. It respects the systems unit settings (for example feet and inches).
- **EDITTYPE\_POS\_INT** - Any integer  $\geq 0$
- **EDITTYPE\_POS\_FLOAT** - Any floating point value  $\geq 0.0$
- **EDITTYPE\_POS\_UNIVERSE** - This is a positive value in world space units. It respects the systems unit settings (for example feet and inches) .
- **EDITTYPE\_TIME** - This is a time value. It respects the system time settings (SMPTE for example).

**int idEdit1,int idSpin1**

These are the resource IDs of the first edit and spinner controls.

**int idEdit2,int idSpin2**

These are the resource IDs of the second edit and spinner controls.

**int idEdit3,int idSpin3**

These are the resource IDs of the third edit and spinner controls.

**float lowLim**

This is the minimum value the spinner can take on.

**float highLim**

This is the maximum value the spinner can take on.

**float scale**

This is the increment or decrement value used when the user uses the up or down arrow buttons of the spinner control. You may also pass the value **SPIN\_AUTOSCALE**. This causes 3ds max to automatically adjust the value used to increment or decrement based on the current value being edited. This allows the spinner to cover a wider range of values with less mouse movement or button clicking from the user.

**ParamDimension \*dim=defaultDim**

This parameter represents the type and magnitude of the parameter. See [ParamDimension](#).

## Class ParamMapUserDlgProc

See Also: [IParamMap](#).

### Description:

This class is used with parameter maps. If there is some custom handling required by a particular control, the client can derive a class from **ParamMapUserDlgProc** and set it as the parameter map's user callback (usually using **SetUserDialogProc()**).

### Methods:

### Prototype:

```
virtual BOOL DlgProc(TimeValue t,IParamMap *map,HWND  
hWnd,UINT msg,WPARAM wParam,LPARAM lParam)=0;
```

### Remarks:

Implemented by the Plug-In.

This is the dialog proc that will be called to process the control messages. This proc will be called after the default processing is complete.

### Parameters:

#### **TimeValue t**

This is the current time.

#### **IParamMap \*map**

This is a pointer to the parameter map.

#### **HWND hWnd**

This is the handle to the dialog.

#### **UINT msg**

This is the message that should be processed by the dialog proc.

#### **WPARAM wParam**

This is a parameter that holds message specific information.

#### **LPARAM lParam**

This is a parameter that holds message specific information.

### Return Value:

This is essentially the equivalent of a normal Windows dialog proc, so it should return whatever value a normal dialog proc returns for the message. An

exception is that the value **REDRAW\_VIEWS** may be returned to cause the viewports to be redrawn.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Implemented by the Plug-In.

If the DlgProc is non-NULL when the ParamMap is deleted the DeleteThis() method will be called. This method is usually implemented as follows: **void DeleteThis() {delete this;}**

**Prototype:**

**virtual void Update(TimeValue t);**

**Remarks:**

This method is available in release 2.0 and later only.

This method is called whenever the dialog is being updated. If the parameter map is invalidated, 3ds max will update the user interface. When it does, this method is called so a developer may do anything they need to on each update.

**Parameters:**

**TimeValue t**

The time at which the update is taking place.

**Default Implementation:**

**{}**

See Also: For more information on Dialog Procs see the Advanced Topics section on [Custom Controls](#).

## Class ParamBlockPLCB

See Also: [Class ParamVersionDesc](#).

### Description:

This is a handy post load call back for fixing up parameter blocks. This will look up the version of the loaded callback and fix it up so it matches the current version. NOTE: this deletes itself when its done. See [Parameter Maps](#) for more details.

### Methods:

### Prototype:

```
ParamBlockPLCB(ParamVersionDesc *v,int  
cnt,ParamVersionDesc *c,ReferenceTarget *t,int refNum);
```

### Remarks:

Constructor.

### Parameters:

**ParamVersionDesc \*v**

This is an array of ParamVersionDescs.

**int cnt**

This is the number of elements in the array specified above.

**ParamVersionDesc \*c**

This is a pointer to the current version of the ParamVersionDesc.

**ReferenceTarget \*t**

This is a pointer to a reference target. This is usually the **this** pointer of the object.

**int refNum**

This is the reference index of the parameter block.

See Also: Advanced Topics section under [Parameter Maps](#) for an explanation of how this is used.

## Class RefEnumProc

See Also: Advanced Topics section on [References](#).

class RefEnumProc

### Description:

This class is available in release 2.0 and later only.

This is the callback object for the global function

**EnumRefHierarchy(ReferenceMaker \*rm, RefEnumProc &proc);**

This classes **proc()** method is called for each element in the reference hierarchy.

### Methods:

#### Prototype:

**virtual void proc(ReferenceMaker \*rm)=0;**

### Remarks:

Implemented by the Plug-In.

This method is called once for each element in the reference hierarchy.

### Parameters:

**ReferenceMaker \*rm**

A pointer to the reference maker to this item.

## List of Animatable Flags

The following flags are bits of the aflag data member of class Animatable. See methods **Animatable::ClearAFlag()**, **SetAFlag()** and **TestAFlag()** to work with these flags.

### **A\_EVALUATING**

This is used internally.

### **A\_NOTIFYDEP**

This is used internally.

### **A\_CHILD\_TREE\_OPEN**

This indicates the item is a node and its children are opened up in the track view.

### **A\_SUBANIM\_TREE\_OPEN**

This indicates the sub-anim of the item are opened up in the track view.

### **A\_OBJECT\_REDUCED**

This is used internally.

**The following flags depends on the various sub-classes:**

**Atmospheric flags**

**A\_ATMOS\_DISABLED**

The atmosphere effect is disabled.

**Object flags**

**A\_OBJ\_CREATING**

The object is being created. It doesn't want to snap to itself.

**Modifier flags**

**A\_MOD\_DISABLED**

The modifier is disabled.

**A\_MOD\_BEING\_EDITED**

The modifier is being edited.

**A\_MOD\_USE\_SEL**

This is not used any longer.

**ModApp flags**

**A\_MODAPP\_DISABLED**

This is used internally.

**A\_MODAPP\_BEING\_EDITED**

This is used internally.

**A\_MODAPP\_SELECTED**

This is used internally.

**A\_MODAPP\_DISPLAY\_ACTIVE**

This is used internally.

**A\_MODAPP\_DYNAMIC\_BOX**

This is used internally.

**Derived Object Flags**

**A\_DERIVEDOBJ\_DONTDELETE**

This is used internally.

**Control flags**

**A\_ORT\_MASK**

This is used internally.

### **A\_ORT\_BEFORESHIFT**

Bits 5, 6 and 7.

### **A\_ORT\_AFTERSHIFT**

Bits 8, 9, 10.

### **A\_CTRL\_DISABLED**

This is used internally.

### **A\_ORT\_DISABLED**

Indicates that the out of range type is disabled.

### **Inode flags**

#### **A\_INODE\_IK\_TERMINATOR**

Terminates the top of an IK chain

#### **A\_INODE\_IK\_POS\_PINNED**

The position is pinned.

#### **A\_INODE\_IK\_ROT\_PINNED**

The rotation is pinned.

### **ToneOperator flags**

#### **A\_TONEOP\_DISABLED**

The exposure control is disabled.

#### **A\_TONEOP\_PROCESS\_BG**

The exposure control processes the background.

**Flags for Hold and Restore logic, for "lazy holding", to avoid multiple holding.**

**A\_HELD**

Typically a plug-in would not hold unless this flag was not set. Then set it once it has held something, then clear it once EndHold() is called on the RestoreObj. This will keep it from putting multiple restore objects in one cycle. See [Undo/Redo](#) for more details.

**A\_SET**

This is similar to above except this is used by controllers.

**A\_IS\_DELETED**

This is used internally.

**A\_BEING\_AUTO\_DELETED**

This is used internally.

**Reserved for superclass use**

**A\_SUPERCLASS1**

This is used internally.

**A\_SUPERCLASS2**

This is used internally.

**These are reserved for use by plug-ins for any purpose they need. No other plug-in will set these flags.**

**A\_PLUGIN1**

**A\_PLUGIN2**

**A\_PLUGIN3**

**A\_PLUGIN4**

**A\_DEPENDENCY\_TEST**

This is used internally.

**A\_LOCK\_TARGET**

Setting this flag will keep an item from being deleted when you delete a reference to it. For example, if you need to swap references for two items. For instance, say you have two nodes and two objects and you want to swap the object reference of the nodes. If you simply call ReplaceReference() on one node with the other node's object, the old object will get deleted because nothing else is referencing it anymore. By setting this flag temporarily you can keep it from being deleted and perform the swap.

The following work flags may be used by plug-ins but it is not guaranteed that they will remain unaltered by someone else. Therefore they are only used for temporary storage.

**A\_WORK1**

**A\_WORK2**

**A\_WORK3**

**A\_WORK4**

## Class IRenderElementMgr

See Also: [Class IRenderElement](#), [Class FPMixinInterface](#), [Class ReferenceTarget](#), [Render Elements](#)

```
class IRenderElementMgr : public FPMixinInterface
```

### Description:

This class is available in release 4.0 and later only.

This class represents the interface for the Render Element Manager. A sample plugin of a Render Element can be found in the SDK samples;  
`\MAXSDK\SAMPLES\RENDER\RENDERELEMENTS.`

### Methods:

public:

### Prototype:

```
virtual BOOL AppendMergedRenderElement(IRenderElement  
*pRenderElement)=0;
```

### Remarks:

This method gets called by the system to add a Render Element when it is merged from another file.

### Parameters:

**IRenderElement \*pRenderElement**  
A pointer to the Render Element to add.

### Return Value:

TRUE if the merging was successful, FALSE if it was not.

### Prototype:

```
virtual BOOL AppendMergedRenderElement(ReferenceTarget  
*pRenderElement)=0;
```

### Remarks:

This method gets called by the system to add a Render Element when it is merged from another file and ensures that the Reference Target is a Render Element.

**Parameters:**

**ReferenceTarget \*pRenderElement**

A pointer to the Render Element to add.

**Return Value:**

TRUE if the merging was successful, FALSE if it was not.

**Prototype:**

**virtual BOOL AddRenderElement(IRenderElement  
\*pRenderElement)=0;**

**Remarks:**

This method adds an IRenderElement instance to the manager's list.

**Parameters:**

**IRenderElement \*pRenderElement**

A pointer to the Render Element to add.

**Return Value:**

TRUE if the addition was successful, FALSE if it was not.

**Prototype:**

**virtual BOOL AddRenderElement(ReferenceTarget  
\*pRenderElement)=0;**

**Remarks:**

This method adds an IRenderElement instance to the manager's list and ensures that the Reference Target is a Render Element.

**Parameters:**

**ReferenceTarget \*pRenderElement**

A pointer to the Render Element to add.

**Return Value:**

TRUE if the addition was successful, FALSE if it was not.

**Prototype:**

**virtual BOOL RemoveRenderElement(ReferenceTarget  
\*pRenderElement)=0;**

**Remarks:**

This method removes an IRenderElement instance from the manager's list and ensures that the Reference Target is a Render Element.

**Parameters:**

**ReferenceTarget \*pRenderElement**

A pointer to the Render Element to remove.

**Return Value:**

TRUE if the addition was successful, FALSE if it was not.

**Prototype:**

**virtual void RemoveAllRenderElements()=0;**

**Remarks:**

This method will remove all the IRenderElement instances from the manager's list.

**Prototype:**

**virtual int NumRenderElements()=0;**

**Remarks:**

This method returns the number of Render Elements contained in the manager's list.

**Prototype:**

**virtual IRenderElement \*GetRenderElement(int index)=0;**

**Remarks:**

This method returns a pointer to a specific Render Element in manager's list.

**Parameters:**

**int index**

The index of the Render Element in the manager's list.

**Return Value:**

A pointer to the specific Render Element or NULL if the index is invalid.

**Prototype:**

**virtual void SetElementsActive(BOOL elementsActive)=0;**

**Remarks:**

This method sets whether the Render Elements List should be active during a render.

**Parameters:**

**BOOL elementsActive**

TRUE to activate the Render Elements List, FALSE to deactivate.

**Prototype:**

**virtual BOOL GetElementsActive() const = 0;**

**Remarks:**

This method checks if the Render Elements List is active during a render and will return TRUE if it is or FALSE if it is not.

**Prototype:**

**virtual void SetDisplayElements(BOOL displayElements)=0;**

**Remarks:**

This method sets whether the Render Elements should be displayed in their own (VFB) viewer window.

**Parameters:**

**BOOL displayElements**

TRUE to display in their own viewer, FALSE if you do not want to.

**Prototype:**

**virtual BOOL GetDisplayElements() const = 0;**

**Remarks:**

This method checks whether the Render Elements are displayed in their own (VFB) viewer window and will return TRUE if they are or FALSE if they are not.

**Prototype:**

**virtual void SetCombustionOutputEnabled(BOOL**

**combustionOutEnabled)=0;**

**Remarks:**

This method sets whether the Render Element List should be exported to a Combustion format file.

**Parameters:**

**BOOL combustionOutEnabled**

TRUE to enable Combustion format file output.

**Prototype:**

**virtual BOOL GetCombustionOutputEnabled() const = 0;**

**Remarks:**

This method checks whether the Render Element List will be exported to a Combustion format file and will return TRUE if they are or FALSE if they are not.

**Prototype:**

**virtual void SetCombustionOutputPath(const TCHAR  
\*combustionOutputPath)=0;**

**Remarks:**

This method allows you to set the output path for a Combustion format output file.

**Parameters:**

**const TCHAR \*combustionOutputPath**

The path string.

**Prototype:**

**virtual void SetCombustionOutputPath(const TSTR&  
combustionOutputPath)=0;**

**Remarks:**

This method allows you to set the output path for a Combustion format output file.

**Parameters:**

## **const TSTR& combustionOutputPath**

The path string.

### **Prototype:**

```
virtual const TSTR& GetCombustionOutputPath() const = 0;
```

### **Remarks:**

This method returns the output path for a Combustion format file.

### **Prototype:**

```
virtual const TCHAR* GetCombustionOutputPathPtr() const = 0;
```

### **Remarks:**

This method returns the output path for a Combustion format file.

## Class IRenderElement

See Also: [Class SpecialFX](#), [Class PBitmap](#), [Class ISave](#), [Class ILoad](#), [List of Reference Messages](#), [Render Elements](#)

```
class IRenderElement : public SpecialFX
```

### Description:

This class is available in release 4.0 and later only.

This is the interface that must be supported by all render elements whether they support the 3ds max renderer or some other renderer. The UI in the render dialog uses this interface exclusively to control the element.

The methods are almost all state-setting methods, with ones that are settable by the UI provided by both sets and gets. A sample plugin of a Render Element can be found in the SDK samples;

**\MAXSDK\SAMPLES\RENDER\RENDERELEMENTS.**

### Methods:

public:

### Prototype:

```
virtual void SetEnabled(BOOL enabled)=0;
```

### Remarks:

This method enables or disables the Render Element.

### Parameters:

**BOOL enabled**

Set to TRUE in order to enable the Render Element. FALSE to disable it.

### Prototype:

```
virtual BOOL IsEnabled() const = 0;
```

### Remarks:

This method returns TRUE if the Render Element is enabled, otherwise FALSE.

### Prototype:

**virtual void SetFilterEnabled(BOOL filterEnabled)=0;**

**Remarks:**

Each active render element has the option of either using the current AA filter or simple blending within the pixel. This method will set the internal filter enable to the value of parameter **filterEnabled**.

**Parameters:**

**BOOL filterEnabled**

Set to TRUE in order to enable. FALSE to disable it.

**Prototype:**

**virtual BOOL IsFilterEnabled() const = 0;**

**Remarks:**

This method returns the current state of whether the AA filter is enabled.

**Parameters:**

This method returns TRUE if filters for the Render Element are enabled, otherwise FALSE.

**Prototype:**

**virtual BOOL BlendOnMultipass() const =0;**

**Remarks:**

This method returns whether this element type should be blended during multipass effects.

When multipass camera effects such as depth of field are used in a rendering, each of the separate elements may be blended into a final bitmap like the composite color, or they may be "frozen" after the first pass. Blending is not appropriate for some elements, like z-depth. This is typically a query only, it is unlikely that this will need to be turned on & off.

**Return Value:**

TRUE if blending during multipass effects, otherwise FALSE.

**Prototype:**

**virtual void SetApplyAtmosphere(BOOL applyAtmosphere)=0;**

**Remarks:**

This method enables or disables the Apply Atmosphere flag for the Render Element.

**Parameters:****BOOL applyAtmosphere**

Set to TRUE in order to enable the atmospheric effects for the Render Element. FALSE to disable it.

**Prototype:**

**virtual BOOL AtmosphereApplied() const = 0;**

**Remarks:**

This method returns TRUE if atmospheric effects for the Render Element are enabled, otherwise FALSE.

**Prototype:**

**virtual void SetApplyShadows(BOOL applyShadows)=0;**

**Remarks:**

This method enables or disables the Apply Shadows flag for the Render Element.

**Parameters:****BOOL applyShadows**

Set to TRUE in order to enable the shadows for the Render Element. FALSE to disable it.

**Prototype:**

**virtual BOOL ShadowsApplied() const = 0;**

**Remarks:**

This method returns TRUE if shadows for the Render Element are enabled, otherwise FALSE.

**Prototype:**

**virtual void SetElementName(TCHAR\* newName)=0;**

**Remarks:**

This method sets the Render Element's name as it appears in the render dialog.

**Parameters:**

**TCHAR\* newName;**

The name for the Render Element.

**Prototype:**

**virtual const TCHAR\* ElementName() const = 0;**

**Remarks:**

This method returns a string representing the Render Element's name as it appears in the render dialog.

**Prototype:**

**virtual void SetPBBitmap(PBBitmap\* &pPBBitmap) const = 0;**

**Remarks:**

This method allows you to set the bitmapinfo/bitmap to use for the Render Element.

Each render element has an output bitmap. We use the pb2 style bitmap as it contains the pathname as well as the bitmap & bitmapInfo structures needed by windows. The bitmap is created by the render element manager, then held by the element until it's not needed. These are the calls that set & get the elements bitmap.

**Parameters:**

**PBBitmap\* &pPBBitmap**

The pointer to the PBBitmap reference.

**Prototype:**

**virtual void GetPBBitmap(PBBitmap\* &pPBBitmap) const = 0;**

**Remarks:**

This method allows you to get the bitmapinfo/bitmap that is used for the Render Element.

**Parameters:**

## **PBBitmap\* &pPBBitmap**

The pointer to the PBBitmap reference which was retrieved.

### **Prototype:**

```
virtual IRenderElementParamDlg  
*CreateParamDialog(IRendParams *ip);
```

### **Remarks:**

Each render element may define a rollup that will be displayed in the render dialog when that element is selected. Most current render elements do not provide their own rollups, but some like z-depth and blend do. This method creates the elements parameter rollup and return it's pointer to the system. If no rollup is supported, NULL should be returned.

### **Parameters:**

**IRendParams \*ip**

A pointer to the IRendParams data.

### **Default Implementation:**

```
{ return NULL; }
```

### **Prototype:**

```
virtual BOOL SetDlgThing(IRenderElementParamDlg* dlg);
```

### **Remarks:**

Implement this method if you are using the ParamMap2 [AUTO\\_UI](#) system and the IRenderElement has secondary dialogs that don't have the IRenderElement as their 'thing'. Called once for each secondary dialog, for you to install the correct 'thing'. This method will set the "thing" of a secondary dialog.

### **Parameters:**

**IRenderElementParamDlg\* dlg**

The pointer to the parameter dialog.

### **Return Value:**

TRUE if you process the dialog, otherwise FALSE.

### **Default Implementation:**

```
{ return FALSE; };
```

**Prototype:**

```
IOResult Save(ISave *iSave)
```

**Remarks:**

This method handles saving the plugin data. it is critical for merging that this code is called at the start of a plug-in's save and load methods. SpecialFX's base implementation saves/loads SpecialFX::name, which is used to populate the 'Merge Render Elements' dialog box. if a plugin re-implements this function, it should first call **RenderElement::Save(iSave)** or **IRenderElement::Load(iLoad)**

**Parameters:**

```
ISave *iSave
```

You may use this pointer to call methods of ISave to write data.

**Return Value:**

One of the following values: **IO\_OK**, **IO\_ERROR**.

**Default Implementation:**

```
{  
  name = ElementName();  
  return SpecialFX::Save(iSave);  
}
```

**Prototype:**

```
IOResult Load(ILoad *iLoad)
```

**Remarks:**

This method handles loading the plugin data. It is critical for merging that this code is called at the start of a plug-in's save and load methods. SpecialFX's base implementation saves/loads SpecialFX::name, which is used to populate the 'Merge Render Elements' dialog box. if a plugin re-implements this function, it should first call **RenderElement::Save(iSave)** or **IRenderElement::Load(iLoad)**

**Parameters:**

## **ILoad \*iLoad**

You may use this pointer to call methods of ILoad to read data.

### **Return Value:**

One of the following values: **IO\_OK**, **IO\_ERROR**.

### **Default Implementation:**

```
{ return SpecialFX::Load(iLoad); }
```

### **Prototype:**

```
virtual RefResult NotifyRefChanged(Interval changeInt,  
RefTargetHandle hTarget, PartID& partID, RefMessage  
message);
```

### **Remarks:**

A plug-in which makes references must implement this method to receive and respond to messages broadcast by its dependents.

### **Parameters:**

#### **Interval changeInt**

This is the interval of time over which the message is active. Currently, all plug-ins will receive **FOREVER** for this interval.

#### **RefTargetHandle hTarget**

This is the handle of the reference target the message was sent by. The reference maker uses this handle to know specifically which reference target sent the message.

#### **PartID& partID**

This contains information specific to the message passed in. Some messages don't use the **partID** at all. See the section [List of Reference Messages](#) for more information about the meaning of the **partID** for some common messages.

#### **RefMessage message**

The message parameters passed into this method is the specific message which needs to be handled. See [List of Reference Messages](#).

### **Return Value:**

The return value from this method is of type **RefResult**. This is usually **REF\_SUCCEEDED** indicating the message was processed. Sometimes, the

return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

**Default Implementation:**

```
{ return REF_SUCCEED; }
```

**Prototype:**

```
SClass_ID SuperClassID();
```

**Remarks:**

This method returns the plugin's SuperClass ID.

**Default Implementation:**

```
{ return RENDER_ELEMENT_CLASS_ID; }
```

**Prototype:**

```
virtual void* GetInterface(ULONG id)=0;
```

**Remarks:**

This method is being used as a general extension method in 3dw max. The Render Elements use it to get the interface for a specific renderer's element interface. The renderer will call this method to see if an IRenderElement is compatible with it. By asking for a specific renderer interface, the element can either provide the interface, or return **NULL**. If **NULL** is returned, then this element will not be available for this renderer. Note that this strategy allows a single render element to support more than one renderer. The max default renderer iid is in **\MAXSDK\INCLUDE\renderElements.h** and is defined as **0xeffeefe**, accessed through **MaxRenderElement::IID**.

**Parameters:**

**ULONG id**

Currently this is not used and is reserved for future use.

**Prototype:**

```
virtual void ReleaseInterface(ULONG id, void *i)=0;
```

**Remarks:**

This method is not currently used. It is reserved for future use. Its purpose is

for releasing an interface created with **GetInterface()**.

## Class IRenderElementCompatible

See Also: [Class IRenderElement](#) , [Render Elements](#)

class IRenderElementCompatible

### Description:

This class is available in release 4.0 and later only.

The system will ask a Renderer for this interface by calling

**Renderer::GetInterface (IRenderElementCompatible::IID)**. If the

Renderer returns a pointer to this interface, it is implied that the Renderer supports render elements. The system will then call

**IRenderElementCompatible::IsCompatible()** to determine if an IRenderElement instance is compatible with it. To determine compatibility, the

Renderer can call **IRenderElement::GetInterface()** (inherited from class Animatable), passing in an interface ID that a compatible IRenderElement would understand. If the Renderer receives a valid interface pointer, it can return TRUE to **IRenderElementCompatible::IsCompatible()**.

### Methods:

public:

### Prototype:

```
virtual BOOL IsCompatible(IRenderElement  
*pIRenderElement)=0;
```

### Remarks:

This method determines if the specified IRenderElement instance is compatible with the renderer.

### Parameters:

**IRenderElement \*pIRenderElement**

A pointer to the IRenderElement to check for compatibility.

### Return Value:

TRUE if the RenderElement is compatible, FALSE if it is not.

## Class MaxRenderElement

See Also: [Class IRenderElement](#), [Class ShadeContext](#), [Class IllumParams](#) , [Render Elements](#)

```
class MaxRenderElement : public IRenderElement
```

### Description:

This class is available in release 4.0 and later only.

This is the RenderElement base class for 3ds max' default scanline renderer. RenderElement plugins that utilize ShadeContext and IllumParams should subclass from here. . All render elements that support the max renderer should derive from this class. The class implements a handler for the ShadeOutputIndex of the elements value in the ShadeOutput array. Access to the elements value is accomplished within PostIllum or PostAtmosphere by:

```
AColor myColor;  
// ... do some computation  
sc.out.elementVals[ mShadeOutputIndex ] = myColor;  
// ... or  
sc.out.elementVals[ ShadeOutputIndex() ] = myColor;
```

A sample plugin of a Render Element can be found in the SDK samples;  
**\MAXSDK\SAMPLES\RENDER\RENDERELEMENTS.**

### Data Members:

protected:

```
int mShadeOutputIndex;
```

The index into element value array in shadeOutput class.

### Methods:

public:

### Prototype:

```
void SetShadeOutputIndex(int shadeOutputIndex);
```

### Remarks:

This method sets the index into the element array in the ShadeOutput class. This method is implemented by this base class & need not be re-implemented by derivative classes. Derivative classes access the index either through the interface or directly as mShadeOutputIndex. Set is used by the system to assign an index to each element. This index persists throughout a single rendering.

**Parameters:**

**int shadeOutputIndex**

The shadeOutput index.

**Prototype:**

**int ShadeOutputIndex();**

**Remarks:**

This method gets the index into the element array in the ShadeOutput class. This method is implemented by this base class & need not be re-implemented by derivative classes. Derivative classes access the index either through the interface or directly as mShadeOutputIndex. Set is used by the system to assign an index to each element. This index persists throughout a single rendering.

**Prototype:**

**virtual void Update(TimeValue timeValue)**

**Remarks:**

This method will update the element to time **timeValue**. Note that most elements don't care.

Update is called to communicate the time to the element. It will be called on each element before the call to **PostIllum()** is executed. No current render element uses this, but it could prove useful someday. This class provides a default stub implementation, so this need not be implemented unless needed.

**Parameters:**

**TimeValue timeValue**

The timevalue at which the update gets called.

**Prototype:**

```
virtual void PostIllum(ShadeContext& sc, IllumParams& ip)=0;
```

**Remarks:**

This method is used to compute the element and store the result in the ShadeContext's ShadeOutput.

This is the first of the actual computation methods for the render element.

**PostIllum()** is called by the material just after an illumination is computed for a fragment. Some materials, like the multi-materials don't compute illumination themselves but mix illuminations from other leaf materials. Such materials do not call **PostIllum()**, but need to consider the elements when blending the shadeoutputs from the leaf materials. The ShadeContext is the same shadecontext passed into the material's shade method, & contains the member variable 'out', which is the current output from the full pixel shading and the storage place for renderElement values.

The other param, **IllumParams**, contains detailed information about the shading process. Specifically, the component-wise output from the shading process is available in the **IllumParams**. Also the illumParams passed in are dependent on whether the element has requested shadows or not. The standard material keeps two sets of these and supplies the one requested. This method must be implemented, tho some elements merely stub it out or clear the output element value. It should put it's output is in

**sc.out.elementVals[ShadeOutputIndex() ]**. Even if you do not need this function, it is a good idea to clear the element val.

**Parameters:**

**ShadeContext& sc**

A reference to the ShadeContext.

**IllumParams& ip**

A reference to the IllumParams.

**Prototype:**

```
virtual void PostAtmosphere(ShadeContext& sc, float z, float prevZ)=0;
```

**Remarks:**

This is the second computation method and is only called if the elements has

atmospheres applied. If it is applied, then the element value is retrieved from the shadeContext, the atmosphere is applied to it & and the output from the atmosphere is left in **sc.out.c** and **sc.out.t**. Last, PostAtmosphere is called with the shadeContext, and the 2 z values used by the atmosphere to compute it. If the point is directly visible to the camera, then **prevZ** will be 0.0f, otherwise it is the z of the next closest obscuring transparent fragment in front of the fragment being shaded. It is up to the render element to process the output in **sc.out.c** and save the result in **sc.out.elementVals[ShadeOutputIndex() ]**. Note that when PostAtmosphere is called the original value set by PostIllum is saved in the element val. This can be overwritten or used in some computation. For example, to separate atmosphere from the composite color, **sc.out.c** and **sc.out.elementVals[ShadeOutputIndex() ]** can be differenced.

**Parameters:**

**ShadeContext& sc**

A reference to the ShadeContext.

**float z**

The first depth value.

**float prevZ**

The previous depth value.

**Prototype:**

**virtual void\* GetInterface(ULONG id)=0;**

**Remarks:**

The renderer will call this method to see if IRenderElement is compatible with it This is used for future expansion in interface classes. When the 3ds max development team needs to add additional functionality to an interface class this method provides a way to do that without breaking the API. If the 3ds max developers would add methods to an existing class it would invalidate the plug-ins that used the class. Using this method allows additional functionality to be added to the interface class without breaking the API.

**Parameters:**

**ULONG id**

Currently this is not used and is reserved for future use.

**Prototype:**

**virtual void ReleaseInterface(ULONG id, void \*i)=0;**

**Remarks:**

This method is not currently used. It is reserved for future use. Its purpose is for releasing an interface created with **GetInterface()**.

## Class IOsnapManager

See Also: [Class OsnapHit](#), [Class INode](#), [Class ViewExp](#), [Class Matrix3](#), The Advanced Topics section on [Snapping](#).

```
class IOsnapManager
```

### Description:

This class is available in release 2.0 and later only.

This class provides an interface to the OsnapManager. Developers who implement osnaps need to record hits with the osnap manager. Developers implementing command modes are responsible for getting the snap preview done and may be responsible for initializing and closing point sequences. See the Advanced Topics section on [Snapping](#) for more details.

### Methods:

#### Prototype:

```
virtual BOOL getactive() const=0;
```

#### Remarks:

This method is used internally but may be called to determine if the **OsnapManager** is currently on.

#### Prototype:

```
virtual BOOL getAxisConstraint()=0;
```

#### Remarks:

This method is used internally but may be called to determine if the **OsnapManager** will use the system level axis constraints when performing translation.

#### Prototype:

```
virtual void RecordHit(OsnapHit* somehit)=0;
```

#### Remarks:

When a plugin osnap finds a hit, it should call this method to record it with the manager. This will enter the hit in a stack which is sorted based on proximity to the cursor position. When multiple hits are recorded with the manager, the

user has the ability to cycle through the stack.

**Parameters:**

**OsnapHit\* somehit**

A pointer to the hit to record. The Osnap plugin should instantiate the hits and the manager is responsible for freeing them.

**Prototype:**

**virtual BOOL OKForRelativeSnap()=0;**

**Remarks:**

Some snaps only make sense relative to a reference point. Consider, for example, the tangent snap which looks for points of tangency on splines. You can only find a tangent point relative to a second point. As objects are created and manipulated, the OsnapManager maintains a list of the Points that have been input to the current command mode. This method tells you if the first point has been recorded.

**Prototype:**

**virtual BOOL RefPointWasSnapped()=0;**

**Remarks:**

This method tells you if the point on the top of the reference stack was actually snapped.

**Prototype:**

**virtual Point3 GetRefPoint(BOOL top = TRUE)=0;**

**Remarks:**

This method retrieves the Point on the top of the reference stack. The point returned is in world space. Note that calling this method when the stack is empty will crash the program. Remember to call **OKForRelativeSnap()** first. Here are the first few lines of the tangent snap's main method.

```
void ShapeSnap::Snap(Object* pObj, IPoint2 *p, TimeValue t) {  
    if(!theman->OKForRelativeSnap())  
        return;  
    //Get the reference point
```

**Point3 relpoint(theman->GetRefPoint());//the last point the user clicked.**

**//transform the reference point into the node's coordinate system**

**Matrix3 tm = theman-> GetObjectTM();**

**relpoint = Inverse(tm) \* relpoint;**

**...**

**Parameters:**

**BOOL top = TRUE**

The default is to return the top of the stack, i.e. the last point which was input to the command mode. If you pass FALSE, the bottom of the stack will be returned.

**Prototype:**

**virtual BOOL IsHolding()=0;**

**Remarks:**

This method is used internally. This method tells if the any hits were recorded with the OsnapManager during the last scene traversal.

**Prototype:**

**virtual OsnapHit &GetHit()=0;**

**Remarks:**

This method is used internally. It returns the current hit i.e. the hit which the manager is currently displaying.

**Prototype:**

**virtual ViewExp \*GetVpt()=0;**

**Remarks:**

Returns a pointer to a **ViewExp**. This is only valid for the duration of the current scene traversal. It is guaranteed to be valid while in a plugin's **Snap()** method.

**Prototype:**

**virtual INode \*GetNode()=0;**

**Remarks:**

Returns a pointer to the node which is currently being snapped.

**Prototype:**

**virtual int GetSnapStrength()=0;**

**Remarks:**

Returns the current snap strength. This is the radius of a circular area about the cursor. It is the plugin's responsibility to determine if any "interesting" parts of the object lie within this area. For testing points, see the method

**OsnapManager::CheckPotentialHit()**.

**Prototype:**

**virtual Matrix3 GetObjectTM()=0;**

**Remarks:**

Returns the object transformation matrix of the node which is currently being snapped.

**Prototype:**

**virtual TimeValue GetTime()=0;**

**Remarks:**

Returns the animation time of the current scene traversal.

**Prototype:**

**virtual void wTranspoint(Point3 \*inpt, IPoint3 \*outpt)=0;**

**Remarks:**

This method is not currently used. The method transforms a point in the current node's object space into screen space.

**Parameters:**

**Point3 \*inpt**

A pointer to the object space point.

**IPoint3 \*outpt**

A pointer to storage for the screen space point.

**Prototype:**

**virtual void Reset()=0;**

**Remarks:**

This method may be called to clear out the OsnapManager's reference point stack. Typically this is handled internally. However, objects which implement their own creation processes may need to call this method upon completion of a creation cycle.

# Class HitMesh

See Also: [Class Osnap](#), The Advanced Topics section on [Snapping](#)..

```
class HitMesh
```

## Description:

This class is available in release 2.0 and later only.

This is a class to hold a list of object space points for highlighting the geometry associated with a hit. One oddity of this class is that you need to allocate one point more than the number of points you actually need. This additional storage is used for clipping against the viewport at display time. For example the "endpoint" snap would allocate a three-point hitmesh to highlight the edge which was hit (See the example below) .

The class has data members for the number of points and a pointer to the actual **Point3** list.

In practice, developers need only use the first two methods shown below.

## Methods:

### Prototype:

```
HitMesh(int n);
```

### Remarks:

Constructor. The number of vertices is set to the value **n** passed and the points array is allocated with that size. This is the constructor that plugin developers will typically use in conjunction with the **setVert()** method described below.

### Parameters:

```
int n
```

The number of points to allocate.

### Prototype:

```
void setVert(int i, const Point3 &xyz);
```

### Remarks:

Sets the 'i-th' vertex to the specified value.

### Parameters:

```
int i
```

The vertex to set.

```
const Point3 &xyz
```

The value to set.

**In practice, developers need only use the two previous methods. The following code segment from the mesh snap exemplifies their use.**

```
//add the hit points based on the active subsnaps
```

```
if(GetActive(EDGE_SUB)
```

```
)// The edge snapping is active and we have a hit on the edge  
defined by from and to.
```

```
{
```

```
    HitMesh *hitmesh = new HitMesh(3); //Allocate one more than we  
need.
```

```
    hitmesh->setVert(0, from);
```

```
    hitmesh->setVert(1, to);
```

```
    float dap = Length(cursor - sf2);
```

```
    assert(Length(st2 - sf2)>=0.0f);
```

```
    float pct = (float)sqrt(fabs(dap*dap - distance*distance)) /  
Length(st2 - sf2);
```

```
    Point3 cand;
```

```
    float pctout = gw->interpWorld(&xyz[0],&xyz[1],pct,&cand);
```

```
    theman->RecordHit(new EdgeHit(cand, this, EDGE_SUB,  
hitmesh, ifrom, ito, pct));
```

```
}
```

**Prototype:**

```
    HitMesh();
```

**Remarks:**

Constructor. The number of points is set to zero and the point list is set to NULL.

**Prototype:**

**HitMesh(const HitMesh& h);**

**Remarks:**

Constructor. The number of points and the points are initialized from the **HitMesh** passed.

**Parameters:**

**const HitMesh& h**

The **HitMesh** to init from.

**Prototype:**

**~HitMesh();**

**Remarks:**

Destructor. If point list is allocated it is freed.

**Prototype:**

**int getNumVerts();**

**Remarks:**

Returns the number of points in this **HitMesh**.

**Prototype:**

**void setNumVerts(int n);**

**Remarks:**

Sets the number of vertices. This frees any existing verts and allocates new ones.

**Parameters:**

**int n**

The number of points to allocate.

**Prototype:**

**Point3& getVert(int i);**

**Remarks:**

Returns the 'i-th' vertex.

**Prototype:**

**Point3\* getVertPtr();**

**Remarks:**

Returns a pointer to the array of points.

**Operators:**

**Prototype:**

**Point3 operator[](int i);**

**Remarks:**

Access operator. Return the 'i-th' point.

**Parameters:**

**int i**

Specifies the point to return.

## Class OsnapHit

See Also: [Class IOsnapManager](#), [Class Osnap](#), [Class HitMesh](#), [Class Point3](#), [Class IPoint3](#), The Advanced Topics section on [Snapping](#).

```
class OsnapHit : public BaseInterfaceServer
```

### Description:

This class is available in release 2.0 and later only.

This class encapsulates the data required to record a snapped point. Typically a plug-in creates instances of this class and records them with the **OsnapManager**. The manager is responsible for freeing the memory associated with recorded hits. All the methods of this class are implemented by the system. If a snap plugin needs to record additional data for its hits, it should derive from this class and provide a clone method which copies this additional data and calls the base classes clone method.

### Friend Classes:

```
friend class OsnapManager;
```

```
friend class Osnap;
```

### Methods:

### Prototype:

```
OsnapHit(Point3 p3, Osnap* s, int sub, HitMesh *m);
```

### Remarks:

Constructor. The data members are initialized to the values passed.

### Parameters:

**Point3 p3**

The point that was hit in object space.

**Osnap\* s**

Points to the Osnap instance which made this hit.

**int sub**

The sub-snap index which made this hit.

**HitMesh \*m**

Points to the mesh used to hilite the topology that was hit.

**Prototype:**

**OsnapHit(const OsnapHit& h);**

**Remarks:**

Constructor. The data members are initialized from the **OsnapHit** passed.

**Parameters:**

**const OsnapHit& h**

The data members are copied from this **OsnapHit**.

**Prototype:**

**virtual ~OsnapHit();**

**Remarks:**

Destructor. If a **HitMesh** was allocated it is deleted.

**Prototype:**

**virtual OsnapHit\* clone();**

**Remarks:**

Returns a pointer to a new instance of the **OsnapHit** class and initializes it with this instance. Developers deriving from this class should override this method.

**Prototype:**

**void setscreendata(IPoint3 screen3, int len);**

**Remarks:**

This method is used internally. Sets the data members associated with screen hit data.

**Parameters:**

**IPoint3 screen3**

The hit location in screen space.

**int len**

The distance from the cursor.

**Prototype:**

**boolean display(ViewExp \*vpt, TimeValue t, Point3 color, int markersize, boolean markers = TRUE, boolean hilite = TRUE);**

**Remarks:**

This display method is implemented by the system and used internally.

**Parameters:**

**ViewExp \*vpt**

The viewport to display in.

**TimeValue t**

The current time.

**Point3 color**

The color to draw it in.

**int markersize**

The relative size of the icon.

**boolean markers = TRUE**

Controls whether or not the hit icon is drawn.

**boolean hilite = TRUE**

Controls whether or not the mesh part of the hit is drawn.

**Prototype:**

**void erase(ViewExp \*vpt, TimeValue t) const;**

**Remarks:**

This method is not currently used.

**Prototype:**

**void GetViewportRect(TimeValue t, ViewExp \*vpt, Rect \*rect, int marksize) const;**

**Remarks:**

Implemented by the system.

This method determines the damage rectangle for this hit.

**Parameters:**

**TimeValue t**

The time at which to compute the rectangle.

**ViewExp \*vpt**

The viewport in which to compute the rectangle.

**Rect \*rect**

Points to storage for the computed result.

**int marksize**

The size of the icon.

**Prototype:****Point3 GetHitpoint();****Remarks:**

Returns the hit location in object space.

**Prototype:****Point3 GetWorldHitpoint();****Remarks:**

Returns the hit location in world space.

**Prototype:****IPoint3 GetHitscreen();****Remarks:**

Returns the hit location in screen space. **IPoint3.z** is the depth in screen space.

**Prototype:****int GetSubsnap();****Remarks:**

Returns the sub-snap index which made this hit.

**Prototype:**

**INode \*GetNode();**

**Remarks:**

Returns a pointer to the node which got hit.

**Prototype:**

**void Update(TimeValue t);**

**Remarks:**

This method is not currently implemented.

**Prototype:**

**Point3 ReEvaluate(TimeValue t);**

**Remarks:**

Implemented by the System.

This method updates the internal data to reflect a change in time. For example, if a hit is recorded on the endpoint of a particular edge of a mesh, the node moving would invalidate the hit data and a call to this method would be required before using its data.

**Parameters:**

**TimeValue t**

The time at which to reevaluate it.

**Return Value:**

The updated point in world space.

**Prototype:**

**void Dump()const;**

**Remarks:**

This method is used internally for debugging and is not implemented for plug-in use.

**Operators:**

**Prototype:**

**virtual OsnapHit& operator=(const OsnapHit& h);**

**Remarks:**

Assignment operator.

**Parameters:**

**const OsnapHit& h**

The **OsnapHit** to assign.

**Prototype:**

**BOOL operator<(OsnapHit& hit);**

**Remarks:**

This comparison operator is used in sorting.

**Parameters:**

**OsnapHit& hit**

The **OsnapHit** to compare.

**Return Value:**

TRUE if the distance from the cursor is less than **hit**; otherwise checks if the depth in Z space is less than **hit**. If so TRUE; otherwise FALSE.

**Prototype:**

**BOOL operator>(OsnapHit& hit);**

**Remarks:**

This comparison operator is used in sorting.

**Parameters:**

**OsnapHit& hit**

The **OsnapHit** to compare.

**Return Value:**

TRUE if the distance from the cursor is less than **hit**; otherwise checks if the depth in Z space is less than **hit**. If so TRUE; otherwise FALSE.

## Class OsnapMarker

See Also: [Class Osnap](#), [Class IOsnapManager](#), [Class GraphicsWindow](#), [Class IPoint3](#).

```
class OsnapMarker
```

### Description:

This class is available in release 2.0 and later only.

This class is used for drawing Osnap markers in the viewports. The marker is drawn as a polyline. The class maintains a cache of the points for the polyline. There are constructors used to initialize the cache and a **display()** method to draw the marker in the specified viewport.

The Osnap class must implement the **GetMarkers()** method which typically returns pointers to these static instances.

### Methods:

#### Prototype:

```
OsnapMarker();
```

#### Remarks:

Constructor. The cache of marker points is set to NULL and the number of points is set to 0.

#### Prototype:

```
OsnapMarker(int n, IPoint3 *ppt, int *pes);
```

#### Remarks:

Constructor. This initializes the cache with the points and edge flags passed.

#### Parameters:

**int n**

The number of points in the marker polyline.

**IPoint3 \*ppt**

The array of points for the polyline.

**int \*pes**

The edge state array. This is an array that indicates if the 'n-th' edge is one of three state:

### **GW\_EDGE\_SKIP**

Nonexistent - totally invisible.

### **GW\_EDGE\_VIS**

Exists and is solid.

### **GW\_EDGE\_INVIS**

Exists and is hidden - shown as a dotted line.

You may pass NULL for this array and the method will assume that the edges are all solid.

#### **Prototype:**

**OsnapMarker(const OsnapMarker& om);**

#### **Remarks:**

Constructor. The marker data is initialized from the **OsnapMarker** passed.

#### **Parameters:**

**const OsnapMarker& om**

The settings are copied from this **OsnapMarker**.

#### **Prototype:**

**~OsnapMarker();**

#### **Remarks:**

Destructor. If any marker points have been allocated for the cache they are freed.

#### **Prototype:**

**void display(IPoint3 xyz, int markersize, GraphicsWindow \*gw);**

#### **Remarks:**

This is method is used internally to display the marker cache at the specified size in the specified viewport. Plugin developers need not call this method.

#### **Operators:**

#### **Prototype:**

**OsnapMarker& operator=(const OsnapMarker& om);**

**Remarks:**

Assignment operator.

**Parameters:**

**const OsnapMarker& om**

The OsnapMarker to assign.

## Structure SnapInfo

See Also: [List of Snap Flags](#), [Class ViewExp](#).

This structure describes the snap settings used for an operation.

**typedef struct {**

**int snapType;**

One of the following values:

**SNAP\_2D**

2-D Snap.

**SNAP\_25D**

2 1/2-D Snap.

**SNAP\_3D**

3-D Snap.

**int strength;**

Maximum snap distance.

**int vertPriority;**

Geometry vertex priority.

**int edgePriority;**

Geometry edge priority.

**int gIntPriority;**

Grid intersection priority.

**int gLinePriority;**

Grid line priority.

**DWORD flags;**

See [List of Snap Flags](#).

**Matrix3 plane;**

Plane to use for snap computations.

**Point3 bestWorld;**

Best snap point in world space.

**Point2 bestScreen;**

Best snap point in screen space.

**int bestDist;**

Best snap point distance.

```
int priority;  
    Best point's priority.  
} SnapInfo;
```

## Class XFormModes

See Also: [List of Standard Sub-Object Modes](#).

class XFormModes

### Description:

This class contains a set of six command mode pointers that make up the XForm modes. Plug-in developers can specify these for their sub-object types. See the method **ActivateSubobjSel** in [class BaseObject](#) or [class Control](#) for more details.

### Data Members:

public:

**CommandMode \*move;**

Standard command mode to process Move.

**CommandMode \*rotate;**

Standard command mode to process Rotate.

**CommandMode \*scale;**

Standard command mode to process Non-Uniform Scale.

**CommandMode \*uscale;**

Standard command mode to process Uniform Scale.

**CommandMode \*squash;**

Standard command mode to process Squash.

**CommandMode \*select;**

Standard command mode to process Selection.

### Methods:

### Prototype:

**XFormModes();**

### Remarks:

Constructor. All the data members are set to NULL.

### Prototype:

**XFormModes(CommandMode \*move,CommandMode**

**\*rotate,CommandMode \*scale,CommandMode  
\*uscale,CommandMode \*squash,CommandMode \*select)**

**Remarks:**

Constructor. The data members are set to the command modes passed.

## List of Standard Sub-Object Modes

Below is a list of the standard sub-object selection command modes. These classes are ready to be instantiated - no derived classes need to be defined. The constructors are shown along with their intended use.

The following modes are appropriate for any object or modifier in the pipeline that needs to implement its sub-object selection modes. While these modes are named **...ModBox...** they are used by objects as well (for example the boolean object and the loft object) - so despite their name they are not just for use by modifiers.

```
MoveModBoxCMode( BaseObject *o, IObjParam *i )  
RotateModBoxCMode( BaseObject *o, IObjParam *i )  
UScaleModBoxCMode( BaseObject *o, IObjParam *i )  
NUScaleModBoxCMode( BaseObject *o, IObjParam *i )  
SquashModBoxCMode( BaseObject *o, IObjParam *i )  
SelectModBoxCMode( BaseObject *o, IObjParam *i )
```

When controllers implement their sub-object selection the following modes may be used:

```
MoveCtrlApparatusCMode( Control *c, IObjParam *i )  
RotateCtrlApparatusCMode( Control *c, IObjParam *i )  
NUScaleCtrlApparatusCMode( Control *c, IObjParam *i )  
SquashCtrlApparatusCMode( Control *c, IObjParam *i )  
SquashCtrlApparatusCMode( Control *c, IObjParam *i )  
SelectCtrlApparatusCMode( Control *c, IObjParam *i )
```

For working with modifier and sub-object selection, two classes are defined that the developer may use. These are **GenModSelectionProcessor** and **SubModSelectionProcessor**.

See the Advanced Topics section on [Sub-Object Selection](#) for more information on the use of these modes.

## Class SubObjAxisCallback

See Also: [Class BaseObject](#), [Class Control](#).

class SubObjAxisCallback

### Description:

The callback is used by the two methods **GetSubObjectCenters()** and **GetSubObjectTMs()** found in the classes **BaseObject** and **Control**.

### Methods:

#### Prototype:

```
virtual void Center(Point3 c,int id)=0;
```

#### Remarks:

Implemented by the System.

This method is called to specify the individual coordinate system center for the axes whose **id** is passed.

#### Parameters:

**Point3 c**

The center point.

**int id**

The id of the axis.

#### Prototype:

```
virtual void TM(Matrix3 tm,int id)=0;
```

#### Remarks:

Implemented by the System.

This method is called to specify the individual coordinate system transformation for the axes whose **id** is passed.

#### Parameters:

**Matrix3 tm**

The transformation matrix.

**int id**

The id of the axis.

**Prototype:**

```
virtual int Type()=0;
```

**Remarks:**

Implemented by the System.

The user has three options for center of the coordinate system, center of the selection set, or pivot. For center of the coordinate system the system does not need to call **GetSubObjCenters()**. The plug-in may call this method to determine which center option it is returning the Centers and TMs for.

**Return Value:**

One of the following values:

**SO\_CENTER\_SELECTION**

**SO\_CENTER\_PIVOT**

## Time Function Reference

See Also: Advanced Topic [Intervals](#), [Class Interval](#).

This section documents the API functions related to time. For an overview of these functions see the Advanced Topics section [Time](#).

**Function:**

**int GetTicksPerFrame();**

**Remarks:**

Returns the number of ticks per frame.

**Function:**

**void SetTicksPerFrame(int ticks);**

**Remarks:**

Sets the number of ticks per frame.

**Parameters:**

**int ticks**

The number of ticks per frame.

**Function:**

**int GetFrameRate();**

**Remarks:**

Retrieves the current frame rate used by 3ds max. This is the Frames Per Second (FPS) setting in the Time Configuration dialog.

**Return Value:**

The current frame rate in frames per second.

**Function:**

**void setFrameRate(int rate);**

**Remarks:**

Sets the current frame rate used by 3ds max. This is the Frames Per Second (FPS) setting in the Time Configuration dialog. Note: This call is simply another way to adjust the ticks per frame setting.

**Parameters:****int rate**

The rate in frames per second.

**Function:****TimeDisp GetTimeDisplayMode();****Remarks:**

Returns the display mode in use for time values.

**Return Value:**

One of the following values:

**DISPTIME\_FRAMES**

Frame display format.

**DISPTIME\_SMPTE**

SMPTE time code format.

**DISPTIME\_FRAMETICKS**

Frame:Ticks format.

**DISPTIME\_TIMETICKS**

MM:SS:Ticks format.

**Function:****void SetTimeDisplayMode(TimeDisp m);****Remarks:**

Sets the display mode for time values. When this setting is changed a notification is sent to automatically update any UI controls containing time values (including plug-in custom controls).

**Parameters:****TimeDisp m**

The time display mode. This may be one of the following values:

**DISPTIME\_FRAMES**

Frame display format.

**DISPTIME\_SMPTE**

SMPTE time code format.

## **DISPTIME\_FRAMETICKS**

Frame:Ticks format.

## **DISPTIME\_TIMETICKS**

MM:SS:Ticks format.

### **Function:**

```
void TimeToString(TimeValue t,TSTR &string);
```

### **Remarks:**

Formats a time value into a string based on the current frame rate, ticks per frame and display mode.

### **Parameters:**

#### **TimeValue t**

The time to format.

#### **TSTR &string**

The string to store the result.

### **Function:**

```
void StringToTime(TSTR string,TimeValue &t);
```

### **Remarks:**

Parses the specified time string using the current time settings (frame rate, ticks per frame, and time format) and converts it to a TimeValue.

### **Parameters:**

#### **TSTR string**

The string to convert.

#### **TimeValue &t**

The TimeValue to store the result.

**The following #defines are useful when working with time (from \MAXSDK\INCLUDE\MAXTYPES.H).**

```
#define TIME_TICKSPERSEC 4800
```

```
#define TicksToSec( ticks ) ((float)
```

```
(ticks)/(float)TIME_TICKSPERSEC)
```

```
#define SecToTicks( secs ) ((TimeValue)  
(secs*TIME_TICKSPERSEC))  
#define TicksSecToTime( ticks, secs ) ( (TimeValue)  
(ticks)+SecToTicks(secs) )  
#define TimeToTicksSec( time, ticks, secs ) { (ticks) =  
(time)%TIME_TICKSPERSEC; (secs) =  
(time)/TIME_TICKSPERSEC ; }  
#define TIME_PosInfinity 0x7fffffff  
#define TIME_NegInfinity 0x80000000  
typedef int TimeValue;
```

## List of Screen-Time-Value Macros

The following macros are available for use when working with Track View. These are used to scale into and out of screen space.

**#define TimeToScreen(t,scale,scroll)**

**(int(floor((t)\*(scale)+0.5)) - (scroll))**

Given a TimeValue t, if you have the time scale and time scroll factors, this macro returns the screen space X for that time.

**#define ScreenToTime(s,scale,scroll)**

**((int)floor((s)/(scale) + (scroll)/(scale)+0.5))**

Given a screen coordinate s, if you have the scale and scroll factors, this macro will return a TimeValue for that screen position.

**#define ValueToScreen(v,h,scale,scroll)**

**(h-int(floor((v)\*(scale)+0.5)) - (scroll))**

Given a Value v, if you have the value scale and value scroll factors, this macro returns the screen space Y for that value.

**#define ScreenToValue(s,h,scale,scroll)**

**((float(h)-(float(s)+float(scroll)))/(scale))**

Given a screen coordinate s, if you have the scale and scroll factors, this macro will return a value for that screen position.

## Class ActionCallback

See Also: [Class BaseInterfaceServer](#), [Class ActionTable](#), [Class ActionItem](#), [Class ActionContext](#), [Class IActionManager](#), [Class DynamicMenu](#), [Class DynamicMenuCallback](#), [Class Interface](#).

```
class ActionCallback : public BaseInterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

An important part of implementing an ActionTable is creating a sub-class of this class. This is an abstract class with a virtual method called ExecuteAction(int id). Developers need to sub-class this class and pass an instance of it to the system when they activate an ActionTable. Then when the system wants to execute an action the ExecuteAction() method is called.

All methods of this class are implemented by the plug-in.

### Data Members:

private:

```
ActionTable *mpTable;
```

Points to the table that uses this ActionCallback.

### Methods:

public:

### Prototype:

```
virtual BOOL ExecuteAction(int id) = 0;
```

### Remarks:

This method is called to actually execute the action.

### Parameters:

```
int id
```

The ID of the item to execute.

### Return Value:

This returns a BOOL that indicates if the action was actually executed. If the item is disabled, or if the table that owns it is not activated, then it won't execute, and returns FALSE. If it does execute then TRUE is returned.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual IMenu* GetDynamicMenu(int id, HWND hwnd, IPoint2&  
m);
```

**Remarks:**

This method is called when an action item says it is a dynamic menu. This returns a pointer to the menu itself.

**Parameters:****int id**

The item ID which is passed to the

**DynamicMenuCallback::MenuItemSelected()****HWND hwnd**

If the menu is requested by a right-click quad menu, then hwnd is the window where the click occurred. If the item is used from a menu bar, then hwnd will be NULL.

**IPoint2& m**

If the menu is requested by a right-click quad menu, then this will be the point in the window where the user clicked.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
ActionTable* GetTable();
```

**Remarks:**

Returns a pointer to the ActionTable the callback uses.

**Default Implementation:**

```
{ return mpTable; }
```

**Prototype:**

```
void SetTable(ActionTable* pTable);
```

**Remarks:**

Sets the ActionTable the callback uses.

**Parameters:**

**ActionTable\* pTable**

Points to the ActionTable the callback uses.

**Default Implementation:**

```
{ mpTable = pTable; }
```

## Class ActionContext

See Also: [Class ActionTable](#), [Class ActionItem](#), [Class ActionCallback](#), [Class IActionManager](#), [Class DynamicMenu](#), [Class DynamicMenuCallback](#), [Class Interface](#).

class ActionContext

### Description:

This class is available in release 4.0 and later only.

Every ActionTable also has an ActionContextId associated with it. This ActionContextId can be shared with other tables.

When assigning keyboard shortcuts to items in an ActionTable, tables that share a unique context id are forced to have unique shortcuts. Tables with different context ids can have overlapping keyboard shortcut assignments.

An ActionContext is an identifier of a group of keyboard shortcuts. Examples are the Main 3ds max UI, Track View, and the Editable Mesh. They are registered using **IActionManager::RegisterActionContext()**.

Note: **typedef DWORD ActionContextId;**

### Methods:

public:

### Prototype:

**ActionContext(ActionContextId contextId, TCHAR \*pName);**

### Remarks:

Constructor. The context ID and the name are initialized from the data passed.

### Parameters:

**ActionContextId contextId**

The ID for the ActionContext.

**TCHAR \*pName**

The name for the ActionContext.

### Prototype:

**TCHAR\* GetName();**

### Remarks:

Returns the name of this ActionContext.

**Prototype:**

**ActionContextId GetContextId();**

**Remarks:**

Returns the ID of this ActionContext.

**Prototype:**

**bool IsActive();**

**Remarks:**

Returns true if this ActionContext is active; otherwise false. An active ActionContext means that it uses its keyboard accelerators. This corresponds to the "Active" checkbox in the keyboard customization UI.

**Prototype:**

**void SetActive(bool active);**

**Remarks:**

Sets the active state of this ActionContext.

**Parameters:**

**bool active**

Pass true for active; false for inactive.

## Class IActionManager

See Also: [Class ActionTable](#), [Class ClassDesc](#), [Class ActionItem](#), [Class ActionCallback](#), [Class ActionContext](#), [Class DynamicMenu](#), [Class DynamicMenuCallback](#), [Class Interface](#).

class IActionManager

### Description:

This class is available in release 4.0 and later only.

The ActionManager manages a set of ActionTables, callbacks and ActionContexts. The manager handles the keyboard accelerator tables for each ActionTable as well. You get a pointer to this class using

**Interface::GetActionManager()**.

### Methods:

public:

### Prototype:

**virtual void RegisterActionTable(ActionTable\* pTable) = 0;**

### Remarks:

Register an action table with the manager. Note that most plug-ins will not need this method. Instead, plug-ins export action table with the methods in ClassDesc. See [ClassDesc Action Table Methods](#).

### Parameters:

**ActionTable\* pTable**

Points to the Action Table to register.

### Prototype:

**virtual int NumActionTables() = 0;**

### Remarks:

Returns the number of ActionTables.

### Prototype:

**virtual ActionTable\* GetTable(int i) = 0;**

### Remarks:

Returns a pointer to the 'i-th' ActionTable.

**Parameters:**

**int i**

The zero based index of the table.

**Prototype:**

**virtual int ActivateActionTable(ActionCallback\* pCallback,  
ActionTableId id) = 0;**

**Remarks:**

This method is called to activate the action table. Some plug-ins (for instance Modifiers or Geometric Objects) may only want to activate the table when they are being edited in the command panel (between BeginEditParams() and EndEditParams()). Others, for instance Global Utility Plug-ins, may wish to do so when they are initially loaded so the actions are always available.

Note that if this method is called multiple times, only the callback from the last call will be used.

**Parameters:**

**ActionCallback\* pCallback**

Points to the callback object which is responsible for executing the action.

**ActionTableId id**

This is the ID of the table to activate.

**Return Value:**

TRUE if the action table was activated. FALSE if the table is already active or doesn't exist.

**Prototype:**

**virtual int DeactivateActionTable(ActionCallback\* pCallback,  
ActionTableId id) = 0;**

**Remarks:**

This method is called to deactivate the action table. After the table is deactivated (for example in EndEditParams()) the callback object can be deleted. Tables are initially active, please do not call this method without a preceding call to **ActivateActionTable()**.

**Parameters:****ActionCallback\* pCallback**

Points to the callback object responsible for executing the action. Pass the same callback that was originally passed to **ActivateActionTable()** and do not set this to NULL.

**ActionTableId id**

The ID of the table to deactivate.

**Return Value:**

TRUE if the action table was deactivated. FALSE if the table was already deactivated or doesn't exist.

**Prototype:**

```
virtual ActionTable* FindTable(ActionTableId id) = 0;
```

**Remarks:**

This method returns a pointer to the action table as specified by its ID.

**Parameters:****ActionTableId id**

The ID of the table to find.

**Prototype:**

```
virtual BOOL GetShortcutString(ActionTableId tableId, int  
commandId, TCHAR* buf) = 0;
```

**Remarks:**

Retrieves the string that describes the keyboard shortcut for the operation.

**Parameters:****ActionTableId tableId**

The ID of the action table.

**int commandId**

The ID of the command for the action.

**TCHAR\* buf**

Points to storage for the string.

**Return Value:**

TRUE if found; FALSE if not found.

**Prototype:**

```
virtual BOOL GetActionDescription(ActionTableId tableId, int  
commandId, TCHAR* buf) = 0;
```

**Remarks:**

Retrieves a string that describes the specified operation from the action table whose ID is passed.

**Parameters:**

**ActionTableId tableId**

The ID of the action table.

**int commandId**

The ID of the command.

**TCHAR\* buf**

Points to storage for the description string.

**Return Value:**

TRUE if the string was returned; FALSE if not.

**Prototype:**

```
virtual BOOL RegisterActionContext(ActionContextId contextId,  
TCHAR* pName) = 0;
```

**Remarks:**

Register the specified action context with the system. This is called when you create the action table that uses this context.

**Parameters:**

**ActionContextId contextId**

The context ID.

**TCHAR\* pName**

The name for the action context.

**Return Value:**

If the specified action context is already registered FALSE is returned; otherwise TRUE is returned.

**Prototype:**

**virtual int NumActionContexts() = 0;**

**Remarks:**

Returns the number of ActionContexts.

**Prototype:**

**virtual ActionContext\* GetActionContext(int i) = 0;**

**Remarks:**

Returns a pointer to the 'i-th' ActionContext.

**Parameters:**

**int i**

The zero based index of the ActionContext.

**Prototype:**

**virtual ActionContext\* FindContext(ActionContextId contextId) = 0;**

**Remarks:**

Returns a pointer to the ActionContext based on it's ID. If not found NULL is returned.

**Parameters:**

**ActionContextId contextId**

The ID whose context is found.

**Prototype:**

**virtual BOOL IsContextActive(ActionContextId contextId) = 0;**

**Remarks:**

Returns TRUE if the specified context is active; otherwise FALSE.

**Parameters:**

**ActionContextId contextId**

Specifies the context to check.

## Class DynamicMenu

See Also: [Class DynamicMenuCallback](#), [Class IMenu](#), [Class ActionTable](#), [Class ActionItem](#), [Class ActionCallback](#), [Class ActionContext](#), [Class IActionManager](#), [Class Interface](#).

class DynamicMenu

### Description:

This class is available in release 4.0 and later only.

This class provides a simple way for plug-ins to produce the menu needed in the **ActionItem::GetDynamicMenu()** method. The constructor of this class is used to create the menu and the GetMenu() method returns the appropriate IMenu pointer.

### Methods:

public:

### Prototype:

**DynamicMenu(DynamicMenuCallback\* pCallback);**

### Remarks:

Constructor.

### Parameters:

**DynamicMenuCallback\* pCallback**

Points to the instance of the DynamicMenuCallback class that handles the menu selection.

### Prototype:

**IMenu\* GetMenu();**

### Remarks:

Returns a pointer to the IMenu. This method may be called after menu creation to get a pointer to the IMenu created. This is the required value to return from **ActionItem::GetDynamicMenu()**.

### Prototype:

**void AddItem(DWORD flags, UINT itemId, TCHAR\* pItemTitle);**

**Remarks:**

This method adds an item to the dynamic menu.

**Parameters:**

**DWORD flags**

One or more of the following values:

**kDisabled**

Item is disabled (can't be selected)

**kChecked**

Item has a check mark beside it.

**kSeparator**

Item is a separator between the previous menu item and the next one.

**UINT itemId**

The ID for the menu item.

**TCHAR\* pItemTitle**

The name to appear for the menu item.

**Prototype:**

**void BeginSubMenu(TCHAR\* pTitle);**

**Remarks:**

This begins a new sub menu. Items added after this call will appear as sub choices of this one until **EndSubMenu()** is called.

**Parameters:**

**TCHAR\* pTitle**

The name to appear for the sub menu item.

**Prototype:**

**void EndSubMenu();**

**Remarks:**

This ends a sub menu. Items added after this call will appear as they did prior to calling **BeginSubMenu()**.

## Structure ActionDescription

See Also: [Class ActionTable](#), [Class ActionItem](#), [Class ActionCallback](#).

### Description:

This structure is available in release 4.0 and later only.

This is a helper structure used for building ActionTables. A static array of these descriptors is passed to the ActionTable constructor.

### **struct ActionDescription {**

#### **int mCmdID;**

A unique identifier for the command (must be unique per table). When an action is executed this is the command ID passed to

**ActionCallback::ExecuteAction()**.

#### **int mDescriptionResourceID;**

A string resource id that describes the command.

#### **int mShortNameResourceID;**

A string resource ID for a short name for the action. This name appears in the list of Actions in the Customize User Interface dialog.

#### **int mCategoryResourceID;**

A string resource ID for the category of an operation. This name appears in the Category drop down list in the Customize User Interface dialog.

**};**

## Class DynamicMenuCallback

See Also: [Class DynamicMenu](#), [Class IMenu](#), [Class ActionTable](#), [Class ActionItem](#), [Class ActionCallback](#), [Class ActionContext](#), [Class IActionManager](#), [Class Interface](#).

```
class DynamicMenuCallback
```

### Description:

This class is available in release 4.0 and later only.

This is the callback object for a dynamic menu. When a user makes a selection from a dynamic menu the **MenuItemSelected()** method of this class is called to process that selection.

### Methods:

public:

### Prototype:

```
virtual void MenuItemSelected(int itemId) = 0;
```

### Remarks:

This method is called to process the user's menu selection.

### Parameters:

**int itemId**

The ID of the item selected.

## Class ActionItem

See Also: [Class ActionTable](#), [Class ActionCallback](#), [Structure ActionDescription](#), [Class ActionContext](#), [Class IActionManager](#), [Class DynamicMenu](#), [Class DynamicMenuCallback](#), [Class MAXIcon](#), [Class Interface](#).

```
class ActionItem : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

The class ActionItem is used to represent the operation that live in ActionTables. ActionItem is an abstract class with operations to support various UI operations. The system provides a default implementation of this class that works when the table is build with the **ActionTable::BuildActionTable()** method. However, developers may want to specialize this class for more special-purpose applications. For example, MAXScript does this to export macroScripts to an ActionTable. Methods that are marked as internal should not be used.

### Data Members:

protected:

```
ActionTable* mpTable;
```

Points to the table that owns the action.

### Methods:

public:

### Prototype:

```
virtual int GetId() = 0;
```

### Remarks:

This method retrieves the unique identifier for the action. This action must be unique in the table, but not does not have to be unique between tables.

### Prototype:

```
virtual BOOL ExecuteAction() = 0;
```

### Remarks:

Calling ExecuteAction causes the item to be run. This returns a BOOL that

indicates if the action was actually executed. If the item is disabled, or if the table that owns it is not activated, then it won't execute, and returns FALSE.

**Return Value:**

TRUE if the action is executed; otherwise FALSE.

**Prototype:**

**virtual void GetButtonText(TSTR& buttonText) = 0;**

**Remarks:**

This method retrieves the text that will be used when the ActionItem is on a text button. The text is stored into the buttonText parameter.

**Parameters:**

**TSTR& buttonText**

Storage for the retrieved text.

**Prototype:**

**virtual void GetMenuText(TSTR& menuText) = 0;**

**Remarks:**

This method retrieves the text to use when the item is on a menu (either Quad menu or main menu bar). This can be different from the button text. This method is called just before the menu is displayed, so it can update the text at that time. For example, the "Undo" menu item in 3ds max's main menu adds the name of the command that will be undone.

**Parameters:**

**TSTR& menuText**

Storage for the retrieved text.

**Prototype:**

**virtual void GetDescriptionText(TSTR& descText) = 0;**

**Remarks:**

This method gets the text that will be used for tool tips and menu help. This is also the string that is displayed for the operation in all the lists in the customization dialogs.

**Parameters:**

**TSTR& descText**

Storage for the retrieved text.

**Prototype:**

**virtual void GetCategoryText(TSTR& catText) = 0;**

**Remarks:**

This method retrieves the text for the category of the operation. This is used in the customization dialog to fill the "category" drop-down list.

**Parameters:**

**TSTR& catText**

Storage for the retrieved text.

**Prototype:**

**virtual BOOL IsChecked() = 0;**

**Remarks:**

This method determines if the action is in a "Checked" state. For menus, this means that a check mark will appear next to the item, if this returns TRUE. If the item is on a button, this is used to determine if the button is in the "Pressed" state. Note that button states are automatically updated on selection change and command mode changes. If your plug-in performs an operation that requires the CUI buttons to be redrawn, you need to call the method **CUIFrameMgr::SetMacroButtonStates(TRUE)**.

**Prototype:**

**virtual BOOL IsItemVisible() = 0;**

**Remarks:**

This method determines if an item is visible on a menu. If it returns FALSE, then the item is not included in the menu. This can be used to create items that are context sensitive. For example, you may want an item to appear on a menu only when the selected object is of a particular type. To do this, you have this method check the class id of the current selection.

**Prototype:**

```
virtual BOOL IsEnabled() = 0;
```

**Remarks:**

This method determines if the operation is currently available. If it is on a menu, or button, the item is grayed out if this method returns FALSE. If it assigned to a keyboard shortcut, then it will not execute the operation if invoked. . If your plug-in performs an operation that requires the CUI buttons to be redrawn, you need to call the method

```
CUIFrameMgr::SetMacroButtonStates(TRUE).
```

**Return Value:**

TRUE for enabled; FALSE for disabled.

**Prototype:**

```
virtual MaxIcon* GetIcon() = 0;
```

**Remarks:**

If you've provided an icon for this operation, you return it with this method. If no icon is available, this returns NULL. The icon is used on CUI buttons, and in the list of operations in the customization dialogs.

**Prototype:**

```
virtual void DeleteThis() = 0;
```

**Remarks:**

Called to delete the ActionItem. This normally happens when the table that owns it is deleted.

**Prototype:**

```
ActionTable* GetTable();
```

**Remarks:**

This returns a pointer to the table that owns the ActionItem. An item can only be owned by a single table.

**Default Implementation:**

```
{ return mpTable; }
```

**Prototype:**

```
void SetTable(ActionTable* pTable);
```

**Remarks:**

Sets the table that owns the item. Used internally. May be used if you implement a custom sub-class of ActionItem.

**Parameters:**

```
ActionTable* pTable
```

Points to the table to set.

**Default Implementation:**

```
{ mpTable = pTable; }
```

**Prototype:**

```
TCHAR* GetShortcutString();
```

**Remarks:**

Returns the string that describes all the keyboard shortcuts associated with the item. This will look something like "Alt+A" or "C, Shift+Alt+Q". This returns NULL if no keyboard shortcut is associated with the item.

**Prototype:**

```
virtual MacroEntry* GetMacroScript();
```

**Remarks:**

Returns the representation of the macroScript for the item, if it's implemented by a macroScript, it returns NULL otherwise.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual BOOL IsDynamicMenu();
```

**Remarks:**

Determines if a menu is created or if an action takes place. If this method returns TRUE, then the ActionItem creates a menu. If it returns FALSE then

an action is performed.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void SetIsDynamicMenu();
```

**Remarks:**

This method may be called after an action item is created to tell the system that it is a dynamic menu action. Note: Dynamic menus may be added to the quad menus programmatically (via the IMenuManager API) or 'manually'.

**Prototype:**

```
virtual IMenu* GetDynamicMenu(HWND hwnd, IPoint2& m);
```

**Remarks:**

If the ActionItem does produce a menu, this method is called to return a pointer to the menu. See [Class DynamicMenu](#) for an easy way to produce these menus.

**Parameters:**

**HWND hwnd**

If the menu is requested by a right-click quad menu, then this hwnd is the handle of the window where the click occurred. If the item is used from a menu bar, this hwnd will be NULL.

**IPoint2& m**

If the menu is requested by a right-click quad menu, then this parameter is the point in the window where the user clicked.

**Return Value:**

A pointer to the menu.

**Default Implementation:**

```
{ return NULL; }
```

## Units of Measurement Reference

This section documents the API structures and functions related to units of measurement. For overview information see the Advanced Topics section [Units of Measurement](#).

### Structures:

See [Structure DispInfo](#).

### Function:

**double GetMasterScale(int type);**

### Remarks:

Retrieves the master scale in terms of the specified unit type. For example: **GetMasterScale(UNITS\_INCHES)** returns the number of inches per unit. Returns -1.0 if an invalid unit type is supplied.

### Parameters:

**int type**

Supported unit types:

**UNITS\_INCHES**

**UNITS\_FEET**

**UNITS\_MILES**

**UNITS\_MILLIMETERS**

**UNITS\_CENTIMETERS**

**UNITS\_METERS**

**UNITS\_KILOMETERS**

### Return Value:

The master scale in terms of the specified unit type.

### Function:

**void GetMasterUnitInfo(int \*type, float \*scale);**

### Remarks:

Retrieves the master unit settings in effect. These are the unit type (Inches, Feet, Meters, etc.) and the master scale setting.

### Parameters:

**int \*type**

Retrieves the unit type in effect. This may be one of the following values:

Supported unit types:

**UNITS\_INCHES**  
**UNITS\_FEET**  
**UNITS\_MILES**  
**UNITS\_MILLIMETERS**  
**UNITS\_CENTIMETERS**  
**UNITS\_METERS**  
**UNITS\_KILOMETERS**

**float \*scale**

The master scale setting. This is the value the user entered in the  
1 Unit = XXX field of the General Page in the Preference Settings dialog box.

**Function:**

**int SetMasterUnitInfo(int type,float scale);**

**Remarks:**

Set the master unit settings. These are the unit type (Inches, Feet, Meters, etc.)  
and the master scale setting.

**Parameters:****int \*type**

Specifies the unit type to use. This may be one of the following values:

Supported unit types:

**UNITS\_INCHES**  
**UNITS\_FEET**  
**UNITS\_MILES**  
**UNITS\_MILLIMETERS**  
**UNITS\_CENTIMETERS**  
**UNITS\_METERS**  
**UNITS\_KILOMETERS**

**float \*scale**

The master scale setting.

**Return Value:**

Nonzero if the information was set; 0 if invalid values were specified and nothing was stored.

**Function:**

**void GetUnitDisplayInfo(DispInfo \*info);**

**Remarks:**

Retrieves the current unit display information.

**Parameters:**

**DispInfo \*info**

Points to storage for the display information. See [Structure DispInfo](#).

**Function:**

**int SetUnitDisplayInfo(DispInfo \*info);**

**Remarks:**

Sets the unit display information used by 3ds max.

**Parameters:**

**DispInfo \*info**

Points to display information. See [Structure DispInfo](#).

**Return Value:**

Nonzero if the information was set; 0 if invalid values were specified and nothing was stored.

**Function:**

**int GetUnitDisplayType();**

**Remarks:**

Returns the current unit display type.

**Return Value:**

One of the following values:

**UNITDISP\_GENERIC**

**UNITDISP\_METRIC**

**UNITDISP\_US**

**UNITDISP\_CUSTOM**

**Function:****int SetUnitDisplayType(int type);****Remarks:**

Sets the current unit display type.

**Parameters:****int type**

One of the following values:

**UNITDISP\_GENERIC****UNITDISP\_METRIC****UNITDISP\_US****UNITDISP\_CUSTOM****Return Value:**

Nonzero if the value was set. Zero if an improper value was specified.

**Function:****TCHAR \*FormatUniverseValue(float value);****Remarks:**

This function converts the specified value to an ASCII representation according to the current unit scale. Note that this can cause a string overflow, especially when the units are set to miles or kilometers. If an overflow occurs the function returns a null string (i.e. `_T("")`). Thus developers can check for this condition using something like `if (buf[0] == '\0')` to see if an overflow occurred.

**Parameters:****float value**

The input value to convert.

**Function:****float DecodeUniverseValue(TCHAR \*string, BOOL \*valid = NULL);****Remarks:**

This method parses the specified string using the current unit settings and

converts it to a floating point value. If an error occurs in the parsing then valid is set to FALSE.

**Parameters:**

**TCHAR \*string**

The input string to convert.

**BOOL \*valid = NULL**

TRUE if the string was converted correctly; FALSE on error.

## Class RestoreObj

See Also: [Class Hold](#), [Undo and Redo](#).

```
class RestoreObj : public InterfaceServer
```

### Description:

This class is the restore object used in the undo / redo system of 3ds max.

### Methods:

#### Prototype:

```
virtual void Restore(int isUndo)=0;
```

#### Remarks:

Implemented by the Plug-In.

The developer implements this method to restore the state of the database to as it was when **theHold.Put()** was called with this restore object.

#### Parameters:

**int isUndo**

Nonzero if **Restore()** is being called in response to the Undo command; otherwise zero. If **isUndo** is nonzero, the developer needs to save whatever data they need to allow the user to redo the operation.

#### Prototype:

```
virtual void Redo()=0;
```

#### Remarks:

Implemented by the Plug-In.

This method is called when the user selects the Redo command. The developer should restore the database to the state prior to the last Undo command.

#### Prototype:

```
virtual int Size()
```

#### Remarks:

Implemented by the Plug-In.

Returns the size of the restore object in bytes. This size does not need to be

exact but should be close. This is used to make sure all the accumulated restore objects do not grow beyond a manageable size.

**Default Implementation:**

```
{ return 1; }
```

**Return Value:**

The size of the restore object in bytes.

**Prototype:**

```
virtual void EndHold()
```

**Remarks:**

Implemented by the Plug-In.

This method is called when **theHold.Accept()** or **theHold.Cancel()** is called. This means the restore object is no longer held, it was either tossed out or sent to the undo system. The developer may then call **ClearAFlag(A\_HELD)** to indicate the restore object is no longer being held.

**Prototype:**

```
virtual TSTR Description()
```

**Remarks:**

Implemented by the Plug-In.

This method is used internally to 3ds max in debugging only. It is used to display a symbolic name for the restore object.

**Return Value:**

The name of the restore object.

**Default Implementation:**

```
{ return TSTR(_T("---")); }
```

**Prototype:**

```
virtual INT_PTR Execute(int cmd, ULONG_PTR arg1=0,  
    ULONG_PTR arg2=0, ULONG_PTR arg3=0);
```

**Remarks:**

This method is available in release 4.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

**Default Implementation:**

```
{return -1;}
```

## Class Hold

See Also: [Class RestoreObj](#), [Undo and Redo](#).

```
class Hold : public BaseInterfaceServer
```

### Description:

The undo / redo system of 3ds max uses a global instance of this class called **theHold**. Developers call methods of **theHold** object to participate in the undo / redo system.

### Methods:

#### Prototype:

```
void Begin();
```

#### Remarks:

Implemented by the System.

When a developer is about to modify the database they should check to see if **theHold** is 'holding'. This indicates that the **Begin()** method has been called. This signifies the beginning of a potential undo/redo operation. If **theHold** is not holding, they should call **Begin()**. After **Begin()** has been called the system is ready to accept restore objects.

In certain cases the system may already be 'holding' when the plug-in is about to begin its modification to the database. For example controllers would normally not call **Begin()** because it usually has been called already. A procedural object will normally call **Begin()** because there is no other part of the system that may alter a procedural object so **Begin()** would not have been called.

#### Example:

```
theHold.Begin();
```

#### Prototype:

```
int Holding();
```

#### Remarks:

Implemented by the System.

This indicates if **theHold.Begin()** has been called. Any operation that

modifies the database checks to see if **theHold** is currently in a holding state. If the undo system is 'holding' it is ready to accept restore objects. For more details see the Advanced Topics section on [Undo / Redo](#).

**Return Value:**

Nonzero if **theHold** is holding; otherwise 0.

**Example:**

```
if ( theHold.Holding() ) {  
    ...  
}
```

**Prototype:**

```
int Restoring(int& isUndo);
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Returns nonzero if the system is restoring and zero if not.

**Parameters:**

**int& isUndo**

This parameter is updated to indicate if the restore is coming from an undo. It's assigned nonzero if it is; zero if not.

**Prototype:**

```
int Redoing();
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Returns nonzero if the system is redoing and zero if not.

**Prototype:**

```
int RestoreOrRedoing();
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Returns nonzero if the system is restoring or redoing and zero if not.

**Prototype:**

**void Accept(int nameID);**

**Remarks:**

Implemented by the System.

Leaves the database in its modified state and registers an undo object with the undo system. This will allow the user to undo the operation.

**Parameters:**

**int nameID**

The resource ID of the string to appear in the Edit menu next to Undo or Redo.

**Example:**

**theHold.Accept(IDS\_MOVE);**

**Prototype:**

**void Accept(TCHAR \*name);**

**Remarks:**

Implemented by the System.

Leaves the database in its modified state and registers an undo object with the undo system. This will allow the user to undo the operation.

**Parameters:**

**TCHAR \*name**

The string to appear in the Edit menu next to Undo or Redo.

**Prototype:**

**void Cancel();**

**Remarks:**

Implemented by the System.

Restores the database to its previous state and throws out the restore object. This cancels the operation.

**Prototype:**

```
void End();
```

**Remarks:**

Implemented by the System.

This method is used internally to 3ds max and should not be called by a plug-in developer. It leaves the database in its modified state but throws out the restore object.

**Prototype:**

```
void Put(RestoreObj *rob);
```

**Remarks:**

Implemented by the System.

The developer calls this method to register a new restore object with the system. Once this object is registered the developer should set the **A\_HELD** flag of **Animatable**, i.e. call **SetAFlag(A\_HELD)**. This indicates that a restore object is registered with the system.

**Parameters:**

```
RestoreObj *rob
```

The restore object to register.

**Example:**

Example code from **EDMREST.CPP**

```
if ( theHold.Holding() ) {  
    theHold.Put(new MeshSelectRestore(meshData,this));  
}
```

**Prototype:**

```
void Suspend();
```

**Remarks:**

Implemented by the System.

This is used internally. It temporarily suspends holding.

**Prototype:**

**void Resume();**

**Remarks:**

Implemented by the System.

This is used internally. It resumes holding if it was suspended.

**Prototype:**

**void EnableUndo();**

**Remarks:**

Implemented by the System.

This is used internally. Plug-In developers should not call this method. Allows Undo when **Accept()** is called.

**Prototype:**

**void DisableUndo();**

**Remarks:**

Implemented by the System.

This is used internally. Plug-In developers should not call this method. Prevents Undo when **Accept()** is called.

**Prototype:**

**void Restore();**

**Remarks:**

Implemented by the System.

This method will call Restore() on all the restore objects registered since the last **Begin()**. This restores the database to the state it was in when **Begin()** was called. The restore objects are NOT deleted.

**Prototype:**

**void Release();**

**Remarks:**

Implemented by the System.

This tosses out the restore objects since the last **Begin()** but still continues

holding.

Group several Begin-End lists into a single Super-group.

**Prototype:**

**void SuperBegin();**

**Remarks:**

Implemented by the System.

Normally this is NOT needed but in special cases this can be useful. This allows a developer to group a set of Begin()/Accept() sequences to be undone in one operation.

Consider the case of the user using the Shift-Move command to create a new node in the scene. There are two parts to this process. First the node must be cloned and second the position must be tracked as the user moves the mouse to place the new node in the scene. Naturally if the user wanted to Undo this operation, they would expect a single selection of the Undo command would accomplish it. However the process was not a single operation. There was the initial cloning of the node, and then the iterative process of moving the node in the scene, restoring its position, moving it again, restoring it again, etc. Cases like this are handled using methods of **theHold** named **SuperBegin()**, **SuperAccept()** and **SuperCancel()**. These allow the developer to group several restore objects together so that they may be undone via a single selection of Undo. Note that in this example it is only necessary to use **SuperBegin()** and **SuperAccept()** because the move was restoring interactively. Normally a developer does NOT need to use these methods even if they have several operations that modify the database. The undo system will automatically register all the restore objects needed as part of the undo object when **theHold.Accept()** is called and these may be undone using a single UNDO command.

**Sample Code:**

See the sample program

**\MAXSDK\SAMPLES\OBJECTS\BONES.CPP.**

**Prototype:**

**void SuperAccept(int nameID);**

**Remarks:**

Implemented by the System.

When a developer has used **SuperBegin()**, this method is used to Accept. This leaves the database in its modified state and registers the restore object with the undo system. This will allow the user to undo the operation.

**Parameters:**

**int nameID**

The resource ID of the string to appear in the Edit menu next to Undo or Redo.

**Sample Code:**

See the sample program

**\MAXSDK\SAMPLES\OBJECTS\BONES.CPP.**

**Prototype:**

**void SuperAccept(TCHAR \*name);**

**Remarks:**

Implemented by the System.

When a developer has used **SuperBegin()**, this method is used to Accept. This leaves the database in its modified state and registers the restore object with the undo system. This will allow the user to undo the operation.

**Parameters:**

**TCHAR \*name**

The string to appear in the Edit menu next to Undo or Redo.

**Prototype:**

**void SuperCancel();**

**Remarks:**

Implemented by the System.

When a developer has used **SuperBegin()**, this method is used to Cancel. This restores the database to its previous state and throws out the restore object. This cancels the operation.

**Sample Code:**

See the sample program

**\MAXSDK\SAMPLES\OBJECTS\BONES.CPP.**

**Prototype:**

**int GetGlobalPutCount();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the number of times **Put()** has been called in the current session of 3ds max.

## Class BitmapStorage

See Also: [Class BitmapManager](#), [Class BitmapInfo](#), [Class Bitmap](#), [Class BitmapIO](#), [List of Bitmap Error Codes](#).

```
class BitmapStorage : public BaseInterfaceServer
```

### Description:

When an image is loaded the buffer that will hold it is an instance of this class. This class provides methods that allow developers to access the image data in a uniform manner even though the underlying storage might be 1-bit, 8-bit, 16-bit, 32-bit or 64-bit. Standard methods are available for getting / putting pixels:

**Get/PutPixels()**, **Get/Put16Gray()**, **Get/Put64Pixels()**,  
**Get/PutTruePixels()**, **Get/PutIndexPixels()**, etc.

Since a developer accesses the storage through this standard interface, certain plug-in types may not need to allocate memory for the storage. For example, an image loader that creates an image from scratch (such as a gradient generator). This plug-in would simply derive a new type of **BitmapStorage** and provide the pixels through the common methods, creating them as requested.

Note: The "**Get/PutPixels()**" methods of this class access the image a single scanline at a time.

Also note: The following global function may be used by image loader/saver plug-ins to create an instance of **BitmapStorage**:

### Function:

```
BitmapStorage *BMMCreateStorage(BitmapManager *manager,  
UINT type);
```

### Remarks:

This global function will create a new instance of the specified storage type and return a pointer to it.

### Parameters:

**BitmapManager \*manager**

The bitmap manager used for this storage.

**UINT type**

The type of storage to create. One of the following values:

**BMM\_LINE\_ART**  
**BMM\_PALETTED**  
**BMM\_GRAY\_8**  
**BMM\_GRAY\_16**  
**BMM\_TRUE\_16**  
**BMM\_TRUE\_32**  
**BMM\_TRUE\_64**

**Return Value:**

The bitmap storage instance created or NULL if the specified type could not be created.

**Data Members:**

protected:

**int openMode;**

The mode the storage was opened in. See [Bitmap Open Mode Types](#).

**UINT usageCount;**

The number of bitmaps using this storage.

**BitmapManager \*manager;**

The bitmap manager associated with this storage.

**int flags;**

See [List of Bitmap Flags](#).

**int type;**

See [List of Bitmap Types](#).

**BMM\_Color\_48 palette[256];**

The palette entries (256 max). See [Structure BMM\\_Color\\_48](#).

**int paletteSlots;**

The number of palette slots used.

**UWORD \*gammaTable;**

The gamma correction table.

**RenderInfo \*rendInfo;**

A pointer to an instance of RenderInfo. See [Class RenderInfo](#).

public:

**BitmapInfo bi**

Describes the properties of the bitmap associated with this storage.

**Methods:**

**Prototype:**

**inline BitmapManager \*Manager()**

**Remarks:**

Implemented by the System.

Returns the bitmap manager for the storage.

**Prototype:**

**float SetGamma(float gam)**

**Remarks:**

Implemented by the System.

Sets the gamma setting to the value passed.

**Parameters:**

**float gam**

The gamma setting to set.

**Prototype:**

**inline int HasGamma()**

**Remarks:**

Implemented by the System.

Returns nonzero if the gamma table has been allocated; otherwise 0.

**Prototype:**

**void SetHasGamma(BOOL onOff)**

**Remarks:**

Implemented by the System.

This method allocates or deallocates the gamma table.

**Parameters:**

**BOOL onOff**

If TRUE the gamma table is allocated; otherwise it is deleted.

**Prototype:**

**UWORD \*GetInputGammaTable()**

**Remarks:**

Implemented by the System.

This methods returns a pointer to a gamma table that can be used for converting pixels using whatever gamma value is appropriate for the storage. It is typically called inside of the **GetLinearPixels()** method of the particular **BitmapStorage** subclasses which then use the gamma table to convert pixel values to linear values. Plug-In developers in most cases will not need to call this method directly.

**Prototype:**

**inline int OpenMode()**

**Remarks:**

Implemented by the System.

Returns the mode the storage was opened in. See [Bitmap Open Mode Types](#).

**Prototype:**

**inline int Width()**

**Remarks:**

Implemented by the System.

Returns the width (horizontal dimension) of the storage's BitmapInfo instance (**bi.Width()**).

**Prototype:**

**inline int Height()**

**Remarks:**

Implemented by the System.

Returns the height (vertical dimension) of the storage's BitmapInfo instance (**bi.Height()**).

**Prototype:**

## **inline float Aspect()**

### **Remarks:**

Implemented by the System.

Returns the aspect ratio of the storage's BitmapInfo instance (**bi.Aspect()**).

### **Prototype:**

## **inline float Gamma()**

### **Remarks:**

Implemented by the System.

Returns the gamma setting of the storage's BitmapInfo instance (**bi.Gamma()**).

### **Prototype:**

## **inline int Paletted()**

### **Remarks:**

Implemented by the System.

Determines if the image is paletted. If the image has a palette (indexed color), the number of palette slots used is returned; otherwise zero.

### **Prototype:**

## **inline int IsDithered()**

### **Remarks:**

Implemented by the System.

Returns the dithered state of the image. If the image is dithered nonzero is returned; otherwise 0,

### **Prototype:**

## **inline int PreMultipliedAlpha()**

### **Remarks:**

Implemented by the System.

Determines if the image has pre-multiplied alpha. If the image has pre-multiplied alpha nonzero is returned; otherwise 0.

**Prototype:**

**inline int HasAlpha()**

**Remarks:**

Implemented by the System.

Determines if the image has an alpha channel. If the image has an alpha channel nonzero is returned; otherwise 0.

**Prototype:**

**inline void UseScaleColors(int on);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Set whether colors are scaled (on) or clamped (off) when converting from **BMM\_Color\_fl** to **BMM\_Color\_64**.

**Prototype:**

**inline int ScaleColors();**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Returns the last value set by **UseScaleColors**.

**Prototype:**

**inline static void ClampColor(BMM\_Color\_64& out, const BMM\_Color\_fl& in);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out clamping the RGB components to 0 to 65535. The alpha component is not copied.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

**inline static void ClampColorA(BMM\_Color\_64& out, const BMM\_Color\_fl& in);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out clamping the RGB components to 0 to 65535.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

**inline void ScaleColor (BMM\_Color\_64& out, BMM\_Color\_fl in);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out clamping the RGB components to 0 to 65535. The alpha component is not copied.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

**inline void ScaleColorA(BMM\_Color\_64& out, const**

**BMM\_Color\_fl& in);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out clamping the RGB components to 0 to 65535.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

**inline void ClampScaleColor (BMM\_Color\_64& out, const  
BMM\_Color\_fl& in);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out, using the value of **ScaleColors()** to determine the clamping or scaling. The alpha component is not copied.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

**inline void ClampScaleColorA (BMM\_Color\_64& out, const  
BMM\_Color\_fl& in);**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

Converts in to out, using the value of **ScaleColors()** to determine the

clamping or scaling.

**Parameters:**

**BMM\_Color\_64& out**

The result of the conversion.

**BMM\_Color\_fl& in**

The value to convert.

**Prototype:**

**inline int UsageCount()**

**Remarks:**

Implemented by the System.

Returns the number of times this image is being used in the system.

**Prototype:**

**inline int Type()**

**Remarks:**

Implemented by the System.

Returns the type of bitmap managed by this storage. See [List of Bitmap Types](#).

**Prototype:**

**inline int Flags()**

**Remarks:**

Implemented by the System.

Returns the bitmap flags. See [List of Bitmap Flags](#).

**Prototype:**

**inline void SetFlags(DWORD f)**

**Remarks:**

Implemented by the System.

Sets the specified flag bits. See [List of Bitmap Flags](#).

**Parameters:**

## **DWORD f**

The flags to set.

### **Prototype:**

**virtual int MaxRGBLevel() = 0;**

### **Remarks:**

This method returns the number of bits per pixel for each color component. For example a 24-bit TARGA has a **MaxRGBLevel()** of 8 (8 red, 8 green, and 8 blue).

### **Prototype:**

**virtual int MaxAlphaLevel() = 0;**

### **Remarks:**

Returns the number of bits per pixel in the alpha channel.

### **Prototype:**

**virtual int IsHighDynamicRange() = 0;**

### **Remarks:**

This method is available in release 4.0 and later only.

Returns nonzero if this storage uses high dynamic range data; otherwise zero. See the Advanced Topics section [Working With Bitmaps](#) for details on High Dynamic Range bitmaps.

### **Prototype:**

**virtual void \*GetStoragePtr(int \*type)**

### **Remarks:**

Implemented by the Plug-In.

This method is used to get a pointer to the beginning of the image storage. Not all storage types can return a valid pointer. In those cases, this method will set the passed **type** to **BMM\_NO\_TYPE** and return NULL.

### **Parameters:**

**int \*type**

The type of storage is returned here. See [List of Bitmap Types](#).

**Default Implementation:**

```
{ *type = BMM_NO_TYPE; return (NULL); }
```

**Prototype:**

```
virtual void *GetAlphaPtr(int *type);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method will attempt to get a pointer to the beginning of the image alpha storage. Not all storage types can return a valid pointer. In those cases, this call will fail and the user should use some other method described below.

**Parameters:**

**int \*type**

The type of storage is returned here. See [List of Bitmap Types](#).

**Default Implementation:**

```
{ *type = BMM_NO_TYPE; return (NULL); }
```

Below are the standard methods for accessing image pixels. Important Note: The following "GetPixels()" methods operate on a single scanline of the image at a time. Thus the number of pixels+x must be less than the width of the image.

**Prototype:**

```
virtual int Get16Gray(int x, int y, int pixels, WORD *ptr) = 0;
```

**Remarks:**

Implemented by the System.

Retrieves the specified 16 bit grayscale pixels from the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**WORD \*ptr**

Pointer to storage for the retrieved pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

**virtual int Put16Gray( int x, int y, int pixels, WORD \*ptr) = 0;**

**Remarks:**

Implemented by the System.

Stores the 16 bit grayscale pixels to the specified location in the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**WORD \*ptr**

Pointer to storage for the pixels.

**Return Value:**

Nonzero if pixels were stored; otherwise 0.

**Prototype:**

**virtual int Get16Gray(int x, int y, int pixels, float \*ptr) = 0;**

**Remarks:**

Implemented by the System.

Retrieves the specified 16 bit grayscale pixels from the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**float \*ptr**

Pointer to storage for the retrieved pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

**virtual int Put16Gray( int x, int y, int pixels, float \*ptr) = 0;**

**Remarks:**

Implemented by the System.

Stores the 16 bit grayscale pixels to the specified location in the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**float \*ptr**

Pointer to storage for the pixels.

**Return Value:**

Nonzero if pixels were stored; otherwise 0.

**Prototype:**

**virtual int GetLinearPixels(int x,int y,int pixels,BMM\_Color\_64 \*ptr) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method retrieves the specified 64 bit true color pixels from the storage. Pixels returned from this method are NOT gamma corrected. These have linear gamma (1.0). This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to store.

**BMM\_Color\_64 \*ptr**

Pointer to storage for the pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

**inline int GetPixels(int x,int y,int pixels,BMM\_Color\_64 \*ptr)**

**Remarks:**

Retrieves the specified 64-bit pixel values from the bitmap. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_64 \*ptr**

Pointer to storage for the retrieved pixel values. See [Structure BMM\\_Color\\_64](#).

**Return Value:**

Returns nonzero if pixels were retrieved; otherwise 0. If storage has not been

allocated 0 is returned.

**Prototype:**

**int PutPixels(int x,int y,int pixels,BMM\_Color\_64 \*ptr)**

**Remarks:**

Stores the specified 64-bit pixel values into the bitmap's own local storage. The pointer you pass to this method may be freed or reused as soon as the function returns. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**BMM\_Color\_64 \*ptr**

Pixel values to store. See [Structure BMM\\_Color\\_64](#).

**Return Value:**

Returns nonzero if pixels were stored; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**virtual int GetLinearPixels(int x,int y,int pixels,BMM\_Color\_fl \*ptr) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method retrieves the specified 64 bit true color pixels from the storage. Pixels returned from this method are NOT gamma corrected. These have linear gamma (1.0). This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to store.

**BMM\_Color\_fl \*ptr**

Pointer to storage for the pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

**inline int GetPixels(int x,int y,int pixels,BMM\_Color\_fl \*ptr)**

**Remarks:**

Retrieves the specified 64-bit pixel values from the bitmap. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_fl \*ptr**

Pointer to storage for the retrieved pixel values.

**Return Value:**

Returns nonzero if pixels were retrieved; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**int PutPixels(int x,int y,int pixels,BMM\_Color\_fl \*ptr)**

**Remarks:**

Stores the specified 64-bit pixel values into the bitmap's own local storage.

The pointer you pass to this method may be freed or reused as soon as the function returns. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**BMM\_Color\_fl \*ptr**

Pixel values to store.

**Return Value:**

Returns nonzero if pixels were stored; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

```
virtual int GetIndexPixels(int x, int y, int pixels, unsigned char *ptr) = 0;
```

**Remarks:**

Implemented by the System.

Retrieves the specified index color pixels from the storage. This is used to retrieve pixels from a paletted image. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**unsigned char \*ptr**

Pointer to storage for the pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

```
virtual int PutIndexPixels(int x, int y, int pixels, unsigned char  
*ptr) = 0;
```

**Remarks:**

Implemented by the System.

Stores the index color pixels to the specified location in the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**unsigned char \*ptr**

Pointer to the pixels to store.

**Return Value:**

Nonzero if pixels were stored; otherwise 0.

**Prototype:**

```
virtual int CropImage(int width,int height,BMM_Color_64  
fillcolor) = 0;
```

**Remarks:**

Implemented by the Plug-In.

Adjusts the bitmap size to the specified dimensions. The image is not resized to fit; it is cropped or filled with **fillcolor** pixels to accommodate the new size.

**Parameters:**

**int width**

The new horizontal size for the bitmap.

**int height**

The new vertical size for the bitmap.

**BMM\_Color\_64 fillcolor**

If the bitmap's new size is bigger than its current size, this is the color used to fill the new pixels. See [Structure BMM\\_Color\\_64](#).

**Return Value:**

Nonzero if the image was cropped; otherwise 0.

**Prototype:**

**virtual int CropImage(int width, int height, int fillindex) = 0;**

**Remarks:**

Implemented by the Plug-In.

Adjusts the bitmap size to the specified dimensions. The image is not resized to fit; it is cropped or filled with **fillindex** pixels to accommodate the new size.

**Parameters:**

**int width**

The new horizontal size for the bitmap.

**int height**

The new vertical size for the bitmap.

**int fillindex**

If the bitmap's new size is bigger than its current size, this is the color used to fill the new pixels.

**Return Value:**

Nonzero if the image was cropped; otherwise 0.

**Prototype:**

**virtual int ResizeImage(int width,int height,int newpalette) = 0;**

**Remarks:**

Implemented by the Plug-In.

This method is no longer used.

**Prototype:**

**virtual int CopyImage(Bitmap \*from, int operation,  
BMM\_Color\_64 fillcolor)**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped or resized as specified.

**Parameters:**

**Bitmap \*from**

The source bitmap.

**int operation**

The type of copy to perform:

**COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

**COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

**COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

**COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

**BMM\_Color\_64 fillcolor**

Vacant areas of the bitmap are filled with **fillcolor** pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap. See [Structure BMM\\_Color\\_64](#).

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

**virtual int CopyImage(Bitmap \*from, int operation,  
BMM\_Color\_64 fillcolor, BitmapInfo \*bi = NULL) = 0;**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped or resized as specified.

**Parameters:**

**Bitmap \*from**

The source bitmap.

**int operation**

The type of copy to perform:

**COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

**COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

**COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

**COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

**BMM\_Color\_64 fillcolor**

Vacant areas of the bitmap are filled with **fillcolor** pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap. See [Structure BMM\\_Color\\_64](#).

**BitmapInfo \*bi**

When using custom options (resize to fit, positioning, etc.) this is how the flags are passed down to the Bitmap Manager. This is an optional argument -- for simple copy operations, **\*bi** can default to NULL. If present, the code checks the option flags and acts accordingly.

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

```
virtual int CopyImage(Bitmap *from, int operation,  
BMM_Color_fl fillcolor, BitmapInfo *bi = NULL) = 0;
```

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped or resized as

specified.

**Parameters:**

**Bitmap \*from**

The source bitmap.

**int operation**

The type of copy to perform:

**COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

**COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

**COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

**COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

**BMM\_Color\_fl fillcolor**

Vacant areas of the bitmap are filled with **fillcolor** pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap. See [Structure BMM\\_Color\\_fl](#).

**BitmapInfo \*bi**

When using custom options (resize to fit, positioning, etc.) this is how the flags are passed down to the Bitmap Manager. This is an optional argument -- for simple copy operations, **\*bi** can default to NULL. If present, the code checks the option flags and acts accordingly.

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

**virtual int CopyImage( Bitmap \*from,int operation,int fillindex)**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage.

**Parameters:**

**Bitmap \*from**

The source bitmap.

**int operation**

The type of copy to perform.

**COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

**COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

**COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to destination map size (final quality).

**COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

**int fillindex**

Vacant areas of the bitmap are filled with **fillindex** pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap.

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

**virtual int CopyCrop(Bitmap \*from, BMM\_Color\_64 fillcolor);**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped to fit.

**Parameters:****Bitmap \*from**

The bitmap to copy to this bitmap.

**BMM\_Color\_64 fillcolor**

The color to use if the source image is smaller than the destination image. See [Structure BMM\\_Color\\_64](#).

**Return Value:**

Nonzero if the copy/crop was performed; otherwise zero.

**Prototype:**

```
virtual int CopyScaleLow(Bitmap *from);
```

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a lower quality but faster algorithm than

**CopyScaleHigh()**. This is an internal function implemented within BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Parameters:**

**Bitmap \*from**

The source bitmap.

**Return Value:**

Nonzero if the copy/scale was performed; otherwise zero.

**Prototype:**

```
virtual int CopyScaleHigh(Bitmap *from, HWND hWnd,  
BMM_Color_64 **buf = NULL, int w=0, int h=0);
```

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a higher quality but slower algorithm than

**CopyScaleLow()**. This is an internal function implemented within BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Prototype:**

```
virtual int CopyScaleHigh(Bitmap *from, HWND hWnd,  
BMM_Color_fl **buf = NULL, int w=0, int h=0);
```

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a higher quality but slower algorithm than **CopyScaleLow()**. This is an internal function implemented within BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Prototype:**

```
virtual int GetPalette(int start,int count,BMM_Color_48 *ptr) = 0;
```

**Remarks:**

Implemented by the Plug-In.

Retrieves the specified portion of the palette of this storage.

**Parameters:**

**int start**

Zero based index of the first palette entry to retrieve.

**int count**

Number of palette entries to retrieve.

**BMM\_Color\_48 \*ptr**

Points to storage for the palette values. See [Structure BMM\\_Color\\_48](#).

**Return Value:**

Nonzero if the palette was retrieved; otherwise 0.

**Prototype:**

```
virtual int SetPalette(int start, int count, BMM_Color_48 *ptr) = 0;
```

**Remarks:**

Implemented by the Plug-In.

Sets the specified portion of the palette for this storage.

**Parameters:**

**int start**

First palette index entry to store.

**int count**

Number of palette entries to store.

**BMM\_Color\_48 \*ptr**

Points to storage for the palette values. See [Structure BMM\\_Color\\_48](#).

**Return Value:**

Nonzero if the palette was stored; otherwise 0.

**Sample Code:**

See **Load()** in `\MAXSDK\SAMPLES\IO\BMP\BMP.CPP`.

**Prototype:**

```
virtual int GetFiltered(float u, float v, float du, float dv,  
BMM_Color_64 *ptr) = 0;
```

**Remarks:**

Implemented by the Plug-In.

This method uses summed area table or pyramidal filtering to compute an averaged color over a specified area.

**Parameters:**

**float u, float v**

The location in the bitmap to filter. These values go from 0.0 to 1.0 across the size of the bitmap.

**float du, float dv**

The size of the rectangle to sample. These values go from 0.0 to 1.0 across the size of the bitmap.

**BMM\_Color\_64 \*ptr**

The result is returned here - the average over the specified area. See [Structure BMM\\_Color\\_64](#).

**Prototype:**

```
virtual int GetFiltered(float u,float v,float du,float  
dv,BMM_Color_fl *ptr) = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

This method uses summed area table or pyramidal filtering to compute an averaged color over a specified area and outputs to a floating point color structure.

**Parameters:**

**float u, float v**

The location in the bitmap to filter. These values go from 0.0 to 1.0 across the size of the bitmap.

**float du, float dv**

The size of the rectangle to sample. These values go from 0.0 to 1.0 across the size of the bitmap.

**BMM\_Color\_fl \*ptr**

The result is returned here - the average over the specified area.

**Prototype:**

```
virtual int Allocate(BitmapInfo *bi, BitmapManager *manager,  
int openMode) = 0;
```

**Remarks:**

Implemented by the System.

This method is called to allocate image storage.

**Parameters:**

**BitmapInfo \*bi**

Points to an instance of the BitmapInfo class describing the properties of the bitmap.

**BitmapManager \*manager**

Points to the BitmapManager for the bitmap.

**int openMode**

See [Bitmap Open Mode Types](#).

**Return Value:**

Nonzero if storage was allocated; otherwise 0.

**Prototype:**

```
virtual void Scale( float *, int, float *, int );
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method is used internally.

**Prototype:**

**virtual BOOL GetSRow( float \*, int, float \*, int );**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method is used internally.

**Prototype:**

**virtual BOOL PutSRow( float \*, int, float \*, int );**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method is used internally.

**Prototype:**

**virtual BOOL GetSCol( float \*, float \*, int, int );**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method is used internally.

**Prototype:**

**virtual BOOL PutSCol( float \*, float \*, int, int );**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method is used internally.

**Prototype:**

```
virtual BOOL ScaleY( Bitmap *, BMM_Color_fl *, float *, float *,  
HWND, int cw = 0, int ch = 0 );
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method is used internally.

**Prototype:**

```
virtual BOOL ScaleX( Bitmap *, BMM_Color_fl *, float *, float *,  
HWND, int cw = 0, int ch = 0 );
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the System.

This method is used internally.

**Prototype:**

```
virtual int CropImage(int width,int height, BMM_Color_fl  
fillcolor) = 0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

Adjusts the bitmap size to the specified dimensions. The image is not resized to fit; it is cropped or filled with **fillcolor** pixels to accommodate the new size.

**Parameters:****int width**

The new horizontal size for the bitmap.

**int height**

The new vertical size for the bitmap.

**BMM\_Color\_fl fillcolor**

If the bitmap's new size is bigger than its current size, this is the color used to fill the new pixels. See [Structure BMM\\_Color\\_fl](#).

**Return Value:**

Nonzero if the image was cropped; otherwise 0.

**Prototype:**

**virtual int CopyCrop(Bitmap \*from, BMM\_Color\_fl fillcolor) = 0;**

**Remarks:**

This method is available in release 4.0 and later only.

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped to fit.

**Parameters:****Bitmap \*from**

The bitmap to copy to this bitmap.

**BMM\_Color\_fl fillcolor**

The color to use if the source image is smaller than the destination image. See [Structure BMM\\_Color\\_fl](#).

**Return Value:**

Nonzero if the copy/crop was performed; otherwise zero.

## G-Buffer Methods.

### Prototype:

**virtual void \*GetChannel(ULONG channelID, ULONG& chanType)**

### Remarks:

Implemented by the Plug-In.

Returns a pointer to specified geometry/graphics buffer channel, and determines its pixel depth.

### Parameters:

#### **ULONG channelID**

The channel to return a pointer to. See [List of G-Buffer Channels](#).

#### **ULONG& chanType**

The type of the returned channel. One of the following values:

#### **BMM\_CHAN\_TYPE\_UNKNOWN**

Channel not of a known type.

#### **BMM\_CHAN\_TYPE\_1**

1 bit per pixel

#### **BMM\_CHAN\_TYPE\_8**

1 byte per pixel

#### **BMM\_CHAN\_TYPE\_16**

1 word per pixel

#### **BMM\_CHAN\_TYPE\_32**

2 words per pixel

#### **BMM\_CHAN\_TYPE\_48**

3 words per pixel

#### **BMM\_CHAN\_TYPE\_64**

4 words per pixel

#### **BMM\_CHAN\_TYPE\_96**

6 words per pixel

### Default Implementation:

```
{ return NULL;}
```

**Prototype:**

**GBuffer \*GetGBuffer();**

**Remarks:**

Returns a pointer to the G-Buffer associated with this storage.

**Prototype:**

**virtual ULONG CreateChannels(ULONG channelIDs)**

**Remarks:**

Implemented by the Plug-In.

Create the specified channels.

**Parameters:**

**ULONG channelIDs**

Specifies the channels to create. See [List of G-Buffer Channels](#).

**Return Value:**

The channels that are present.

**Default Implementation:**

```
{ return 0;}
```

**Prototype:**

**virtual void DeleteChannels(ULONG channelIDs)**

**Remarks:**

Implemented by the Plug-In.

Delete the specified channels.

**Parameters:**

**ULONG channelIDs**

Specifies the channels to delete. See [List of G-Buffer Channels](#).

**Prototype:**

**virtual ULONG ChannelsPresent()**

**Remarks:**

Implemented by the Plug-In.

Returns the channels that are present. See [List of G-Buffer Channels](#).

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
RenderInfo* AllocRenderInfo();
```

**Remarks:**

Implemented by the Plug-In.

Output bitmaps can get an instance of the class **RenderInfo**, which is written by the renderer. This method will allocate an instance only if a **RenderInfo** doesn't yet exist.

**Return Value:**

A pointer to a **RenderInfo**. See [Class RenderInfo](#).

**Prototype:**

```
RenderInfo* GetRenderInfo();
```

**Remarks:**

Implemented by the Plug-In.

Returns a **RenderInfo** pointer. See [Class RenderInfo](#).

## List of Bitmap Types

See Also: [Class BitmapStorage](#), [Class Color](#), [Class AColor](#), [Structure LogLUV32Pixel](#), [Structure LogLUV24Pixel](#), [Structure RealPixel](#).

The following are the valid types of bitmaps:

This type indicates 'no type' yet.

### **BMM\_NO\_TYPE**

Bitmap has not been allocated yet.

The types below may be both read from or written to:

### **BMM\_LINE\_ART**

1 bit monochrome image.

### **BMM\_PALETTED**

8 bit paletted image. Each pixel value is an index into the color table.

### **BMM\_GRAY\_8**

8 bit grayscale bitmap.

### **BMM\_GRAY\_16**

16 bit grayscale bitmap.

### **BMM\_TRUE\_16**

16 bit true color image.

### **BMM\_TRUE\_32**

32 bit color: 8 bits each for Red, Green, Blue, and Alpha.

### **BMM\_TRUE\_64**

64 bit color: 16 bits each for Red, Green, Blue, and Alpha.

### **BMM\_LOGLUV\_32**

This option is available in release 4.0 and later only.

This format uses a logarithmic encoding of luminance and U' and V' in the CIE perceptively uniform space. It spans 38 orders of magnitude from  $5.43571 \times 10^{-20}$  to  $1.84467 \times 10^{19}$  in steps of about 0.3% luminance steps. It includes both positive and negative colors. A separate 16 bit channel is kept for alpha values.

### **BMM\_LOGLUV\_24**

This option is available in release 4.0 and later only.

This format is similar to **BMM\_LOGLUV\_32** except it uses smaller values to give a span of 5 order of magnitude from 1/4096 to 16 in 1.1%

luminance steps. A separate 8 bit channel is kept for alpha values.

### **BMM\_LOGLUV\_24A**

This option is available in release 4.0 and later only.

This format is identical to **BMM\_LOGUV\_24**, except the 8 bit alpha value is kept with the 24 bit color value in a single 32 bit word.

### **BMM\_REALPIX\_32**

This option is available in release 4.0 and later only.

The "Real Pixel" format.

The following types may be read from, but NOT written to (thus these should not be used when creating bitmaps that you intend to write to or save):

### **BMM\_TRUE\_24**

24 bit color: 8 bits each for Red, Green, and Blue.

### **BMM\_TRUE\_48**

48 bit color: 16 bits each for Red, Green, and Blue.

### **BMM\_YUV\_422**

This is the YUV format - CCIR 601.

### **BMM\_BMP\_4**

4 bit Windows BMP 16 color bitmap

### **BMM\_PAD\_24**

Padded 24 bit (in a 32 bit register).

## Structure BMM\_Color\_fl

See Also: [Class Bitmap](#), [Class BitmapStorage](#), [Class BitmapManager](#), [Working With Bitmaps](#).

### Description:

This structure is available in release 4.0 and later only.

The High Dynamic Range bitmaps introduced in R4 make use of this class to store color information using floating point values.

```
typedef struct {  
    float r,g,b,a;
```

Storage for the floating point color information.

### Prototype:

```
operator float*();
```

### Remarks:

Returns the address of the floating point values.

### Prototype:

```
operator const float*() const;
```

### Remarks:

Returns the address of the floating point values.

### Prototype:

```
static WORD clipColor(float c);
```

### Remarks:

Returns the specified color **c** clipped (limited to) the range 0 to 65535.

```
} BMM_Color_fl;
```

## Structure LogLUV32Pixel

See Also: [Class Color](#).

```
struct LogLUV32Pixel
```

### Description:

This structure is available in release 4.0 and later only.

This structure is a 32 bit pixel format that stores 1 bit for sign, 15 bits for the log of the luminance, and 16 bits of chroma.

This class stores colors in XYZ space. XYZ color space is the space define by the CIE by the red (X), green (Y) and blue (Z) response curves of the eye. So to calculate a color in XYZ space, you take the incoming light, multiply it by each response curvey and integrate the result over the visible spectrum. There are several RGB spaces, all depending on what XYZ coordinates get assigned to the red, green and blue primaries of the space. The transformations between XYZ and RGB space are all linear and can be represented as 3 by 3 matrices.

The mapping used by XYZtoRGB and RGBtoXYZ is for CCIR-709 primaries and was taken from the code in the tiff reader for the LogLUV32 format. Both XYZ and RGB methods are supplied, so developers can supply different XYZ to RGB transforms, if desired.

This transform is important, because in the LogLUV32 format the log is taken of the Y coordinate in XYZ space. So, it is important that Y not be 0. The transform helps guarantee this. In fact, Y is zero only when r, g and b are all 0.

### Methods:

public:

### Data Members:

#### **DWORD32 value;**

Storage for the pixel value.

### Prototype:

**operator Color() const;**

### Remarks:

Returns this pixel format as a Color.

**Prototype:**

```
LogLUV32Pixel& operator=(const float c[3]) { SetRGB(c); return *this; };
```

**Remarks:**

Assignment operator.

**Parameters:**

**const float c[3]**

The array of color values to assign in RGB order.

**Prototype:**

```
void GetRGB(float rgb[3]) const;
```

**Remarks:**

Retrieves the RGB space values.

**Parameters:**

**float rgb[3]**

The results are stored here.

**Prototype:**

```
void SetRGB(const float rgb[3]);
```

**Remarks:**

Sets the RGB space values.

**Parameters:**

**const float rgb[3]**

The values to set.

**Prototype:**

```
void GetXYZ(float xyz[3]) const;
```

**Remarks:**

Retrieves the XYZ space values.

**Parameters:**

**const float xyz[3]**

The results are stored here.

**Prototype:**

**void SetXYZ(const float xyz[3]);**

**Remarks:**

Sets the XYZ space values.

**Parameters:**

**const float xyz[3]**

The values to set.

**Prototype:**

**static void XYZtoRGB(const float xyz[3], float rgb[3]);**

**Remarks:**

This method converts from XYZ space to RGB space.

**Parameters:**

**const float xyz[3]**

The input values to convert.

**float rgb[3]**

The output values are stored here.

**Prototype:**

**static void RGBtoXYZ(const float rgb[3], float xyz[3]);**

**Remarks:**

This method converts from RGB space to XYZ space.

**Parameters:**

**const float rgb[3]**

The input values to convert.

**float xyz[3]**

The output values are stored here.

## Structure LogLUV24Pixel

See Also: [Class Color.](#), [Structure LogUV32Pixel](#)

```
struct LogLUV24Pixel
```

### Description:

This class is available in release 4.0 and later only.

This structure is a 24 bit pixel format that stores 10 bits for log of luminance and 14 bits of chroma.

### Data Members:

**unsigned char value[3];**

Storage for the pixel value.

### Methods:

public:

### Prototype:

**operator Color() const;**

### Remarks:

This method will return the pixel format as a Color.

### Prototype:

**LogLUV24Pixel& operator=(const float c[3]);**

### Remarks:

Assignment operator.

### Parameters:

**const float c[3]**

The array of color values to assign in RGB order.

### Prototype:

**void GetRGB(float rgb[3]) const;**

### Remarks:

This method will return the RGB space values.

**Parameters:**

**float rgb[3]**

The results are stored in this array.

**Prototype:**

**void SetRGB(const float rgb[3]);**

**Remarks:**

This method allows you to set the RGB space values.

**Parameters:**

**const float rgb[3]**

The values to set.

**Prototype:**

**void GetXYZ(float xyz[3]) const;**

**Remarks:**

This method will return the XYZ space values.

**Parameters:**

**float xyz[3]**

The values are stored in this array.

**Prototype:**

**void SetXYZ(const float xyz[3]);**

**Remarks:**

This method allows you to set the XYZ space values.

**Parameters:**

**const float xyz[3]**

The values to set.

**Prototype:**

**static void XYZtoRGB(const float xyz[3], float rgb[3]);**

**Remarks:**

This method will convert from XYZ space to RGB space.

**Parameters:**

**const float xyz[3]**

The input values to convert.

**float rgb[3]**

The output values are stored in this array.

**Prototype:**

**static void RGBtoXYZ(const float rgb[3], float xyz[3]);**

**Remarks:**

This method will convert from RGB space to XYZ space.

**Parameters:**

**const float rgb[3]**

The input values to convert.

**float xyz[3]**

The output values are stored in this array.

## Class BitmapStorageLDR

See Also: [Class BitmapStorage](#).

```
class BitmapStorageLDR : public BitmapStorage
```

### Description:

This class is available in release 4.0 and later only.

This is the base class for the development of plug-in Bitmap Storage plug-ins that don't use High Dynamic Range bitmaps.

Note that **bVirtual** is actually a shortcut macro for **BMMExport virtual**.

All methods of this class are implemented by the System.

### Methods:

public:

### Prototype:

```
bVirtual int IsHighDynamicRange();
```

### Remarks:

This method returns 0 if the bitmap is not a high dynamic range bitmap or 1 if it is.

### Default Implementation:

```
{ return(0); }
```

### Prototype:

```
bVirtual int StraightCopy(Bitmap *from);
```

### Remarks:

Implemented by the System.

This method does a straightforward copy from the specified bitmap.

### Parameters:

**Bitmap \*from**

The bitmap to copy from.

**Prototype:**

**bVirtual int Get16Gray(int x,int y,int pixels,float \*ptr);**

**Remarks:**

Implemented by the System.

Retrieves the specified 16 bit grayscale pixels from the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**float \*ptr**

Pointer to the storage for the retrieved pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

**bVirtual int Put16Gray(int x,int y,int pixels,float \*ptr);**

**Remarks:**

Implemented by the System.

Stores the 16 bit grayscale pixels to the specified location in the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**float \*ptr**

Pointer to the storage for the pixels.

**Return Value:**

Nonzero if pixels were stored; otherwise 0.

**Prototype:**

**bVirtual int GetLinearPixels(int x,int y,int pixels,BMM\_Color\_fl \*ptr);**

**Remarks:**

Implemented by the Plug-In.

This method retrieves the specified 64 bit true color pixels from the storage. Pixels returned from this method are NOT gamma corrected. These have linear gamma (1.0). This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_fl \*ptr**

Pointer to the storage for the retrieved pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

**bVirtual int GetPixels(int x,int y,int pixels,BMM\_Color\_fl \*ptr);**

**Remarks:**

Retrieves the specified 64-bit pixel values from the bitmap. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_fl \*ptr**

Pointer to the storage for the retrieved pixels.

**Return Value:**

Returns nonzero if pixels were retrieved; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**bVirtual int PutPixels(int x,int y,int pixels,BMM\_Color\_fl \*ptr);**

**Remarks:**

Stores the specified 64-bit pixel values into the bitmap's own local storage. The pointer you pass to this method may be freed or reused as soon as the function returns. Note: This method provides access to pixel data one scanline at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**BMM\_Color\_fl \*ptr**

The pixels values to store.

**Return Value:**

Returns nonzero if pixels were stored; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**bVirtual int GetIndexPixels(int x,int y,int pixels,unsigned char**

**\*ptr) = 0;**

**Remarks:**

Implemented by the System.

Retrieves the specified index color pixels from the storage. This is used to retrieve pixels from a paletted image. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**unsigned char \*ptr**

Pointer to the storage for the pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

**bVirtual int PutIndexPixels(int x,int y,int pixels,unsigned char \*ptr) = 0;**

**Remarks:**

Implemented by the System.

Stores the index color pixels to the specified location in the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**unsigned char \*ptr**

The pixels values to store.

**Return Value:**

Nonzero if pixels were stored; otherwise 0.

**Prototype:**

**bVirtual int CropImage(int width,int height,BMM\_Color\_fl  
fillcolor);**

**Remarks:**

Implemented by the Plug-In.

Adjusts the bitmap size to the specified dimensions. The image is not resized to fit; it is cropped or filled with fillcolor pixels to accommodate the new size.

**Parameters:**

**int width**

The new horizontal size for the bitmap.

**int height**

The new vertical size for the bitmap.

**BMM\_Color\_fl fillcolor**

If the bitmap's new size is bigger than its current size, this is the color used to fill the new pixels.

**Return Value:**

Nonzero if the image was cropped; otherwise 0.

**Prototype:**

**bVirtual int CopyCrop(Bitmap \*from, BMM\_Color\_64 fillcolor);**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped to fit.

**Parameters:**

**Bitmap \*from**

The bitmap to copy to this bitmap.

**BMM\_Color\_64 fillcolor**

The color to use if the source image is smaller than the destination image.

**Return Value:**

Nonzero if the copy/crop was performed; otherwise zero.

**Prototype:**

**bVirtual int CopyCrop(Bitmap \*from, BMM\_Color\_fl fillcolor);**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped to fit.

**Parameters:**

**Bitmap \*from**

The bitmap to copy to this bitmap.

**BMM\_Color\_fl fillcolor**

The color to use if the source image is smaller than the destination image.

**Return Value:**

Nonzero if the copy/crop was performed; otherwise zero.

**Prototype:**

**bVirtual int CopyScaleLow(Bitmap \*from);**

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a lower quality but faster algorithm than

**CopyScaleHigh()**. This is an internal function implemented within

BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Parameters:**

**Bitmap \*from**

The bitmap to copy to this bitmap.

**Return Value:**

Nonzero if the copy/scale was performed; otherwise zero.

**Prototype:**

```
bVirtual int CopyScaleHigh(Bitmap *from, HWND hWnd,  
BMM_Color_64 **buf = NULL, int w=0, int h=0);
```

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a higher quality but slower algorithm than

**CopyScaleLow()**. This is an internal function implemented within BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Prototype:**

```
bVirtual int CopyScaleHigh(Bitmap *from, HWND hWnd,  
BMM_Color_fl **buf = NULL, int w=0, int h=0);
```

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a higher quality but slower algorithm than

**CopyScaleLow()**. This is an internal function implemented within BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Prototype:**

```
bVirtual int CopyImage(Bitmap *from,int  
operation,BMM_Color_64 fillcolor, BitmapInfo *bi = NULL);
```

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped or resized as specified.

**Parameters:**

**Bitmap \*from**

The source bitmap.

### **int operation**

The type of copy to perform:

#### **COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

#### **COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

#### **COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

#### **COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

### **BMM\_Color\_64 fillcolor**

Vacant areas of the bitmap are filled with fillcolor pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap.

### **BitmapInfo \*bi = NULL**

When using custom options (resize to fit, positioning, etc.) this is how the flags are passed down to the Bitmap Manager. This is an optional argument -- for simple copy operations, **\*bi** can default to NULL. If present, the code checks the option flags and acts accordingly.

### **Return Value:**

Nonzero if the copy was performed; otherwise 0.

### **Prototype:**

```
bVirtual int CopyImage(Bitmap *from,int  
operation,BMM_Color_fl fillcolor, BitmapInfo *bi = NULL);
```

### **Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage.

### **Parameters:**

#### **Bitmap \*from**

The source bitmap.

#### **int operation**

The type of copy to perform:

**COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

**COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

**COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

**COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

**BMM\_Color\_fl fillcolor**

Vacant areas of the bitmap are filled with fillcolor pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap.

**BitmapInfo \*bi = NULL**

When using custom options (resize to fit, positioning, etc.) this is how the flags are passed down to the Bitmap Manager. This is an optional argument -- for simple copy operations, **\*bi** can default to NULL. If present, the code checks the option flags and acts accordingly.

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

**bVirtual int CopyImage(Bitmap \*from,int operation,int fillindex);**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage.

**Parameters:**

**Bitmap \*from**

The source bitmap.

**int operation**

The type of copy to perform:

**COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

**COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

**COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

**COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

**int fillindex**

Vacant areas of the bitmap are filled with fillcolor pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap.

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

**bVirtual int GetFiltered(float u,float v,float du,float dv,BMM\_Color\_fl \*ptr);**

**Remarks:**

Implemented by the Plug-In.

This method uses summed area table or pyramidal filtering to compute an averaged color over a specified area.

**Parameters:**

**float u, float v**

The location in the bitmap to filter. These values go from 0.0 to 1.0 across the size of the bitmap.

**float du, float dv**

The size of the rectangle to sample. These values go from 0.0 to 1.0 across the size of the bitmap.

**BMM\_Color\_fl \*ptr**

The result is returned here - the average over the specified area.

# Class BitmapStorageHDR

See Also: [Class BitmapStorage](#).

```
class BitmapStorageHDR : public BitmapStorage
```

## Description:

This class is available in release 4.0 and later only.

This is the base class for the development of plug-in Bitmap Storage plug-ins that use High Dynamic Range bitmaps.

All methods of this class are implemented by the System.

## Methods:

public:

## Prototype:

```
bVirtual int IsHighDynamicRange();
```

## Remarks:

Implemented by the System.

This method returns 0 if the bitmap is not a high dynamic range bitmap or 1 if it is.

## Default Implementation:

```
{ return(1); }
```

## Prototype:

```
bVirtual int StraightCopy(Bitmap *from);
```

## Remarks:

Implemented by the System.

This method does a straightforward copy from the specified bitmap.

## Parameters:

```
Bitmap *from
```

The bitmap to copy from.

## Prototype:

**bVirtual int Get16Gray(int x,int y,int pixels,WORD \*ptr);**

**Remarks:**

Implemented by the System.

Retrieves the specified 16 bit grayscale pixels from the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**WORD \*ptr**

Pointer to the storage for the retrieved pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

**bVirtual int Put16Gray(int x,int y,int pixels,WORD \*ptr);**

**Remarks:**

Implemented by the System.

Stores the 16 bit grayscale pixels to the specified location in the storage. This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**WORD \*ptr**

Pointer to the storage for the pixels.

**Return Value:**

Nonzero if pixels were stored; otherwise 0.

**Prototype:**

```
bVirtual int GetLinearPixels(int x,int y,int pixels,BMM_Color_64 *ptr);
```

**Remarks:**

Implemented by the Plug-In.

This method retrieves the specified 64 bit true color pixels from the storage. Pixels returned from this method are NOT gamma corrected. These have linear gamma (1.0). This method operates on a single scanline of the image at a time.

**Parameters:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_64 \*ptr**

Pointer to the storage for the retrieved pixels.

**Return Value:**

Nonzero if pixels were retrieved; otherwise 0.

**Prototype:**

```
bVirtual int GetPixels(int x,int y,int pixels,BMM_Color_64 *ptr);
```

**Remarks:**

**int x**

Source x location.

**int y**

Source y location.

**int pixels**

Number of pixels to retrieve.

**BMM\_Color\_fl \*ptr**

Pointer to the storage for the retrieved pixels.

**Return Value:**

Returns nonzero if pixels were retrieved; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**bVirtual int PutPixels(int x,int y,int pixels,BMM\_Color\_64 \*ptr);**

**Remarks:**

Stores the specified 64-bit pixel values into the bitmap's own local storage. The pointer you pass to this method may be freed or reused as soon as the function returns. Note: This method provides access to pixel data one scanline at a time.

**Parameters:****int x**

Destination x location.

**int y**

Destination y location.

**int pixels**

Number of pixels to store.

**BMM\_Color\_fl \*ptr**

The pixels values to store.

**Return Value:**

Returns nonzero if pixels were stored; otherwise 0. If storage has not been allocated 0 is returned.

**Prototype:**

**bVirtual int CropImage(int width,int height,BMM\_Color\_64 fillcolor);**

**Remarks:**

Implemented by the Plug-In.

Adjusts the bitmap size to the specified dimensions. The image is not resized to fit; it is cropped or filled with fillcolor pixels to accommodate the new size.

**Parameters:****int width**

The new horizontal size for the bitmap.

**int height**

The new vertical size for the bitmap.

**BMM\_Color\_64 fillcolor**

If the bitmap's new size is bigger than its current size, this is the color used to fill the new pixels.

**Return Value:**

Nonzero if the image was cropped; otherwise 0.

**Prototype:**

**bVirtual int CropImage(int width,int height,int fillindex);**

**Remarks:**

Implemented by the Plug-In.

Adjusts the bitmap size to the specified dimensions. The image is not resized to fit; it is cropped or filled with fillcolor pixels to accommodate the new size.

**Parameters:****int width**

The new horizontal size for the bitmap.

**int height**

The new vertical size for the bitmap.

**int fillindex**

If the bitmap's new size is bigger than its current size, this is the color used to fill the new pixels.

**Return Value:**

Nonzero if the image was cropped; otherwise 0.

**Prototype:**

**bVirtual int CopyCrop(Bitmap \*from, BMM\_Color\_64 fillcolor);**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped to fit.

**Parameters:**

**Bitmap \*from**

The bitmap to copy to this bitmap.

**BMM\_Color\_64 fillcolor**

The color to use if the source image is smaller than the destination image.

**Return Value:**

Nonzero if the copy/crop was performed; otherwise zero.

**Prototype:**

**bVirtual int CopyCrop(Bitmap \*from, BMM\_Color\_fl fillcolor);**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped to fit.

**Parameters:**

**Bitmap \*from**

The bitmap to copy to this bitmap.

**BMM\_Color\_fl fillcolor**

The color to use if the source image is smaller than the destination image.

**Return Value:**

Nonzero if the copy/crop was performed; otherwise zero.

**Prototype:**

**bVirtual int CopyScaleLow(Bitmap \*from);**

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a lower quality but faster algorithm than

**CopyScaleHigh()**. This is an internal function implemented within BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Parameters:**

**Bitmap \*from**

The bitmap to copy to this bitmap.

**Return Value:**

Nonzero if the copy/scale was performed; otherwise zero.

**Prototype:**

```
bVirtual int CopyScaleHigh(Bitmap *from, HWND hWnd,  
BMM_Color_64 **buf = NULL, int w=0, int h=0);
```

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a higher quality but slower algorithm than

**CopyScaleLow()**. This is an internal function implemented within BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Prototype:**

```
bVirtual int CopyScaleHigh(Bitmap *from, HWND hWnd,  
BMM_Color_fl **buf = NULL, int w=0, int h=0);
```

**Remarks:**

Implemented by the System.

This method copies the specified bitmap to this storage. The source bitmap is scaled to fit using a higher quality but slower algorithm than

**CopyScaleLow()**. This is an internal function implemented within BMM.DLL for copying bitmaps back and forth. If a developer creates new storage type, they will automatically get these copy functions as these are implemented in the base class.

**Prototype:**

```
bVirtual int CopyImage(Bitmap *from,int  
operation,BMM_Color_64 fillcolor, BitmapInfo *bi = NULL);
```

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage. The image is cropped or resized as specified.

**Parameters:****Bitmap \*from**

The source bitmap.

**int operation**

The type of copy to perform:

**COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

**COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

**COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

**COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

**BMM\_Color\_64 fillcolor**

Vacant areas of the bitmap are filled with fillcolor pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap.

**BitmapInfo \*bi = NULL**

When using custom options (resize to fit, positioning, etc.) this is how the flags are passed down to the Bitmap Manager. This is an optional argument -- for simple copy operations, **\*bi** can default to NULL. If present, the code checks the option flags and acts accordingly.

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

```
bVirtual int CopyImage(Bitmap *from,int  
operation,BMM_Color_fl fillcolor, BitmapInfo *bi = NULL);
```

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage.

**Parameters:**

**Bitmap \*from**

The source bitmap.

**int operation**

The type of copy to perform:

**COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

**COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

**COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

**COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

**BMM\_Color\_fl fillcolor**

Vacant areas of the bitmap are filled with fillcolor pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap.

**BitmapInfo \*bi = NULL**

When using custom options (resize to fit, positioning, etc.) this is how the flags are passed down to the Bitmap Manager. This is an optional argument -- for simple copy operations, **\*bi** can default to NULL. If present, the code checks the option flags and acts accordingly.

**Return Value:**

Nonzero if the copy was performed; otherwise 0.

**Prototype:**

**bVirtual int CopyImage(Bitmap \*from,int operation,int fillindex);**

**Remarks:**

Implemented by the Plug-In.

Copies the specified bitmap to this storage.

**Parameters:**

## **Bitmap \*from**

The source bitmap.

## **int operation**

The type of copy to perform:

### **COPY\_IMAGE\_CROP**

Copy image to current map size using cropping if necessary.

### **COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

### **COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

### **COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo \*).

## **int fillindex**

Vacant areas of the bitmap are filled with fillcolor pixels if the operation specified is **COPY\_IMAGE\_CROP** and one of the source bitmap dimensions is less than the size of this bitmap.

## **Return Value:**

Nonzero if the copy was performed; otherwise 0.

## **Prototype:**

```
bVirtual int GetFiltered(float u,float v,float du,float dv,BMM_Color_64 *ptr);
```

## **Remarks:**

Implemented by the Plug-In.

This method uses summed area table or pyramidal filtering to compute an averaged color over a specified area.

## **Parameters:**

### **float u, float v**

The location in the bitmap to filter. These values go from 0.0 to 1.0 across the size of the bitmap.

### **float du, float dv**

The size of the rectangle to sample. These values go from 0.0 to 1.0 across the size of the bitmap.

**BMM\_Color\_fl \*ptr**

The result is returned here - the average over the specified area.

## Class AColor

See Also: [Class Color](#), [Structure BMM\\_Color\\_24](#), [Structure BMM\\_Color\\_32](#), [Structure BMM\\_Color\\_48](#), [Structure BMM\\_Color\\_64](#), [Structure BMM\\_Color\\_fl](#), [Structure RealPixel](#).

class Acolor

### Description:

This class represents color as four floating point values: **r**, **g**, **b**, and an alpha channel **a**. All methods of this class are implemented by the system.

Also note the following typedef:

```
typedef AColor RGBA;
```

### Data Members:

public:

```
float r,g,b,a;
```

These values are in the range 0.0 to 1.0.

### Methods:

#### Prototype:

```
AColor()
```

#### Remarks:

Constructor. The resulting object should be initialized with one of the initialization methods.

#### Prototype:

```
AColor(float R, float G, float B, float A=1.0f)
```

#### Remarks:

Constructor. Initializes the AColor to the RGBA color values passed.

#### Prototype:

```
AColor(double R, double G, double B, double A=1.0)
```

#### Remarks:

Constructor. Initializes the AColor to the RGBA color values passed (cast to

float).

**Prototype:**

**AColor(int R, int G, int B, int A=0)**

**Remarks:**

Constructor. Initializes the AColor to the RGBA color values passed (cast to float).

**Prototype:**

**AColor(const AColor& c)**

**Remarks:**

Constructor. Initializes the AColor to the AColor passed.

**Prototype:**

**AColor(const Color& c, float alph=1.0f)**

**Remarks:**

Constructor. Initializes the AColor to the Color passed, optionally specifying an alpha value.

**Prototype:**

**AColor(DWORD rgb, float alph=1.0f);**

**Remarks:**

Constructor. Initializes the color to the Windows RGB value, optionally specifying an alpha value.

**Prototype:**

**AColor(float af[4])**

**Remarks:**

Constructor. Initializes the color to the value passed.

**Parameters:**

**float af[3]**

Specifies the color. r=af[0], g=af[1], b=af[2], a=af[3].

**Prototype:**

**AColor(const BMM\_Color\_24& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this AColor from the 24 bit color value passed.

**Parameters:**

**const BMM\_Color\_24& c**

The 24 bit color to initialize from.

**Prototype:**

**AColor(const BMM\_Color\_32& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this AColor from the 32 bit color value passed.

**Parameters:**

**const BMM\_Color\_32& c**

The 32 bit color to initialize from.

**Prototype:**

**AColor(const BMM\_Color\_48& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this AColor from the 48 bit color value passed.

**Parameters:**

**const BMM\_Color\_48& c**

The 48 bit color to initialize from.

**Prototype:**

**AColor(const BMM\_Color\_64& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this AColor from the 64 bit color value passed.

**Parameters:**

**const BMM\_Color\_64& c**

The 64 bit color to initialize from.

**Prototype:**

**AColor(const BMM\_Color\_fl& c);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor. Initializes this AColor from the floating point color passed.

**Parameters:**

**const BMM\_Color\_fl& c**

The floating point color to initialize from. No conversion or scaling is done.

**Prototype:**

**void Black()**

**Remarks:**

Sets this AColor to black.  $r = g = b = 0.0f$ ;  $a = 1.0f$

**Prototype:**

**void White()**

**Remarks:**

Sets the AColor to white.  $r = g = b = a = 1.0f$

**Prototype:**

**void ClampMin();**

**Remarks:**

Makes all the components of the AColor  $\geq 0.0$

**Prototype:**

**void ClampMax();**

**Remarks:**

Makes all the components of the AColor  $\leq 1.0$

**Prototype:**

**void ClampMinMax();**

**Remarks:**

Makes all the components of the AColor fall in the range 0.0 to 1.0.

**Prototype:**

**float& operator[](int i)  
const float& operator[](int i) const**

**Remarks:**

Access operators.

**Parameters:**

**int i**

The index of the component to return.

**Return Value:**

0=r, 1=g, 2=b, 3=a.

**Prototype:**

**operator float\*();**

**Remarks:**

Conversion function.

Returns a pointer to the AColor.

**Prototype:**

**operator DWORD()**

**Remarks:**

Convert the AColor to a Windows RGB color. See [COLORREF](#).

**Return Value:**

A Windows RGB color.

**Prototype:**

**operator Point3();**

**Remarks:**

Convert the AColor to a Point3.

**Return Value:**

A Point3. x=r, y=g, z=b.

**Prototype:**

**operator BMM\_Color\_24();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this AColor to the BMM\_Color\_24 format.

**Prototype:**

**operator BMM\_Color\_32();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this AColor to the BMM\_Color\_32 format.

**Prototype:**

**operator BMM\_Color\_48();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this AColor to the BMM\_Color\_48 format.

**Prototype:**

**operator BMM\_Color\_64();**

**Remarks:**

This method is available in release 4.0 and later only.

Converts this AColor to the BMM\_Color\_64 format.

**Prototype:**

**operator BMM\_Color\_fl();**

**Remarks:**

This method is available in release 4.0 and later only.  
Converts this AColor to the BMM\_Color\_fl format.

**Prototype:**

**AColor operator-() const**

**Remarks:**

Unary - operator.

**Return Value:**

The Color with the components negated, i.e.  
{ return(AColor(-r,-g,-b, -a)); }

**Prototype:**

**AColor operator+() const**

**Remarks:**

Unary + operator.

**Return Value:**

The AColor itself.

**Prototype:**

**inline AColor& operator-=(const AColor&);**

**Remarks:**

Subtracts an AColor from this AColor.

**Return Value:**

A Color that is the difference between two Colors.

**Prototype:**

**inline AColor& operator+=(const AColor&);**

**Remarks:**

Adds an AColor to this AColor.

**Return Value:**

An AColor that is the sum of two AColors.

**Prototype:**

**inline AColor& operator\*=(float);**

**Remarks:**

Multiplies the components of this AColor by a float.

**Return Value:**

An AColor multiplied by a float.

**Prototype:**

**inline AColor& operator/=(float);**

**Remarks:**

Divides the components of this AColor by a float.

**Return Value:**

An AColor divided by a float.

**Prototype:**

**inline AColor& operator\*=(const AColor&);**

**Remarks:**

Performs element-by-element multiplying between two AColors.

**Return Value:**

This AColor element-by-element multiplied by another AColor.

**Prototype:**

**int operator==(const AColor& p) const**

**Remarks:**

Test for equality between two AColors.

**Return Value:**

Nonzero if the AColors are equal; otherwise 0.

**Prototype:**

**int operator!=(const AColor& p) const**

**Remarks:**

Tests for inequality between two AColors.

**Return Value:**

Nonzero if the AColors are not equal; otherwise 0.

**Prototype:**

**inline AColor operator-(const AColor&) const;**

**Remarks:**

Subtracts an AColor from an AColor.

**Return Value:**

An AColor that is the difference between two AColors.

**Prototype:**

**inline AColor operator+(const AColor&) const;**

**Remarks:**

Adds an AColor to an AColor.

**Return Value:**

An AColor that is the difference between two AColors.

**Prototype:**

**inline AColor operator/(const AColor&) const;**

**Remarks:**

Divides an AColor by an AColor.

**Return Value:**

An AColor divided by an AColor. r/r, g/g, b/b, a/a.

**Prototype:**

**inline AColor operator\*(const AColor&) const;**

**Remarks:**

Multiplies an AColor by an AColor.

**Return Value:**

An AColor multiplied by an AColor. r\*r, g\*g, b\*b, a\*a.

**Prototype:**

**inline AColor operator^(const AColor&) const;**

**Remarks:**

Cross product of two AColors.

**Return Value:**

An AColor that is the cross product of two AColors.

**Prototype:**

**int MaxComponent(const Color&)**

**Remarks:**

Returns the index of the component with the maximum absolute value.

**Parameters:**

**const Color&**

The color to check.

**Return Value:**

The index of the component with the maximum absolute value. r=0, g=1, b=2, a=3.

**Prototype:**

**int MinComponent(const Color&)**

**Remarks:**

Returns the index of the component with the minimum absolute value.

**Parameters:**

**const Color&**

The color to check.

**Return Value:**

The index of the component with the minimum absolute value. r=0, g=1, b=2, a=3.

**Prototype:**

**inline AColor operator\*(float f, const AColor& a)**

**inline AColor operator\*(const AColor& a, float f)**

**Remarks:**

Multiplies each component of an AColor by a float.

**Return Value:**

An AColor with each component multiplied by a float.

**Prototype:**

**inline AColor CompOver(const AColor &fg, const AColor& bg)**

**Remarks:**

Composite **fg** over **bg**, assuming associated alpha, i.e. pre-multiplied alpha for both **fg** and **bg**

This is: **fg + (1.0f-fg.a)\*bg**

**Parameters:**

**const AColor &fg**

Specifies the foreground color to composite.

**const AColor& bg**

Specifies the background color to composite over.

**Return Value:**

The resulting AColor.

## Class StdCubic

See Also: [Class Texmap](#).

class StdCubic : public Texmap

### Description:

This class provides access to the parameters of the standard 3ds max Reflect/Refract texture. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual void SetSize(int n, TimeValue t)=0;**

#### Remarks:

Sets the map size parameter.

#### Parameters:

**int n**

The size in pixels.

**TimeValue t**

The time to set the value.

#### Prototype:

**virtual void SetDoNth(BOOL onoff)=0;**

#### Remarks:

Sets the 'Every Nth Frame' or 'First Frame Only' toggle.

#### Parameters:

**BOOL onoff**

TRUE for 'Every Nth Frame'; FALSE for 'First Frame Only'.

#### Prototype:

**virtual void SetNth(int n)=0;**

#### Remarks:

Sets the 'Nth Frame' parameter to the specified value.

#### Parameters:

**int n**

The Nth Frame setting.

**Prototype:**

**virtual void SetApplyBlur(BOOL onoff)=0;**

**Remarks:**

Sets or clears the 'Apply blur' checkbox.

**Parameters:**

**BOOL onoff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetBlur(float b, TimeValue t)=0;**

**Remarks:**

Sets the blur setting to the specified value at the specified time.

**Parameters:**

**float b**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetBlurOffset(float b, TimeValue t)=0;**

**Remarks:**

Sets the blur offset setting to the specified value at the specified time.

**Parameters:**

**float b**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void UseHighDynamicRange(BOOL onoff)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Sets if the reflect / refract texture uses high dynamic range bitmaps or not. See [Working With Bitmaps](#) for details on high dynamic range bitmaps.

**Parameters:**

**BOOL onoff**

Pass TRUE for on; FALSE for off.

**Prototype:**

**virtual int GetSize(TimeValue t)=0;**

**Remarks:**

Returns the size setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual BOOL GetDoNth()=0;**

**Remarks:**

Returns the state of the 'Every Nth Frame' or 'First Frame Only' toggle.

**Return Value:**

**BOOL onoff**

TRUE is 'Every Nth Frame'; FALSE is 'First Frame Only'.

**Prototype:**

**virtual int GetNth()=0;**

**Remarks:**

Returns the Nth Frame setting.

**Prototype:**

**virtual BOOL GetApplyBlur()=0;**

**Remarks:**

Returns the state of the 'Apply blur' checkbox.

**Return Value:**

TRUE is on; FALSE is off.

**Prototype:**

**virtual float GetBlur(TimeValue t)=0;**

**Remarks:**

Returns the blur setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetBlurOffset(TimeValue t)=0;**

**Remarks:**

Returns the blur offset setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

## Class StdMirror

See Also: [Class Texmap](#).

```
class StdMirror : public Texmap
```

### Description:

This class provides access to the 3ds max Flat Mirror material. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
virtual void SetDoNth(BOOL onoff)=0;
```

#### Remarks:

This method determines if 'Every Nth Frame' or 'First Frame Only' is used.

#### Parameters:

**BOOL onoff**

TRUE for Every Nth Frame; FALSE for First Frame Only.

#### Prototype:

```
virtual void SetNth(int n)=0;
```

#### Remarks:

This methods controls the 'Nth Frame' value.

#### Parameters:

**int n**

The number of frames.

#### Prototype:

```
virtual void SetApplyBlur(BOOL onoff)=0;
```

#### Remarks:

This method controls the 'Apply Blur' check box setting.

#### Parameters:

**BOOL onoff**

TRUE to toggle on; FALSE to toggle off.

**Prototype:**

**virtual void SetBlur(float b, TimeValue t)=0;**

**Remarks:**

Sets the specified blur value at the specified time.

**Parameters:**

**float b**

The blur value to set in the range 0.0 to 100.0

**TimeValue t**

The time at which to set the blur value.

**Prototype:**

**virtual void UseHighDynamicRange(BOOL onoff)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Sets if the mirror texture uses high dynamic range bitmaps or not. See [Working With Bitmaps](#) for details on high dynamic range bitmaps.

**Parameters:**

**BOOL onoff**

Pass TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetDoNth()=0;**

**Remarks:**

Determines if 'Every Nth Frame' or 'First Frame Only' is used.

**Return Value:**

TRUE if Every Nth Frame is in use; FALSE if First Frame Only is in use.

**Prototype:**

**virtual int GetNth()=0;**

**Remarks:**

Returns the Nth Frame setting.

**Prototype:**

**virtual BOOL GetApplyBlur()=0;**

**Remarks:**

Returns TRUE if the Apply Blur check box is on; otherwise FALSE.

**Prototype:**

**virtual float GetBlur(TimeValue t)=0;**

**Remarks:**

Returns the blur setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the blur setting.

## Structure BMM\_Color\_24

See Also: [Class Bitmap](#), [Class BitmapStorage](#), [Class BitmapManager](#).

```
typedef struct {  
    BYTE r,g,b;  
    24 bit color: 8 bits each for Red, Green, and Blue.  
} BMM_Color_24;
```

## Structure BMM\_Color\_32

See Also: [Class Bitmap](#), [Class BitmapStorage](#), [Class BitmapManager](#).

```
typedef struct {
```

```
    BYTE r,g,b,a;
```

```
    32 bit color: 8 bits each for Red, Green, Blue, and Alpha.
```

```
} BMM_Color_32;
```

## Structure BMM\_Color\_48

See Also: [Class Bitmap](#), [Class BitmapStorage](#), [Class BitmapManager](#).

```
typedef struct {
```

```
    WORD r,g,b;
```

```
    48 bit color: 16 bits each for Red, Green, and Blue.
```

```
} BMM_Color_48;
```

## Structure BMM\_Color\_64

See Also: [Class Bitmap](#), [Class BitmapStorage](#), [Class BitmapManager](#).

```
typedef struct {
```

```
    WORD r,g,b,a;
```

```
    64 bit color: 16 bits each for Red, Green, Blue, and Alpha.
```

```
} BMM_Color_64;
```

## Structure RealPixel

See Also: [Class Color](#).

```
struct RealPixel
```

### Description:

This structure describes color in terms of **r, g, b, e**.

This is taken from GraphicsGems II, "Real Pixels" by Greg Ward of Lawrence Berkeley Laboratory. What it means is this: "**e**" is the base 2 exponent of the maximum RGB component, and **r,g,b** are the mantissas of R,G,and B, relative to this exponent. It essentially compresses the essential data of a floating point color into 32 bits.

Quoting from Graphics Gems II:

"It appears that this format favors the largest primary value at the expense of accuracy in the other two primaries. This is true, but it also is true that the largest value dominates the displayed

pixel color so that the other primaries become less noticeable"

One GBuffer option is to write out the image in RealPixel format, storing NON CLAMPED colors. This could be used by a Video Post process to detect those areas of the image where the intensity goes beyond 1 and apply halo and flare effects much more realistically.

There are functions for converting between floating point and RealPixel format:

**RealPixel MakeRealPixel(float r, float g, float b);**

**ExpandRealPixel(RealPixel &rp, float& r, float &g, float& b);**

as well as methods in RealPixel and Color.

### Structure Data:

**unsigned char e;**

The base 2 exponent of the maximum RGB component.

**unsigned char r,g,b;**

The mantissas of R,G,and B, relative to this exponent.

### Operators:

### Prototype:

**operator Color();**

**Remarks:**

Converts the RealPixel format to the Color format.

## List of Copy Image Operations

### **COPY\_IMAGE\_CROP**

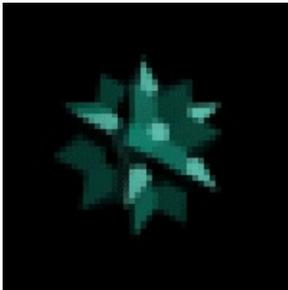
Copy image to current map size using cropping if necessary.

### **COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

Resize the source image to the destination map size (draft quality).

This is a resize from 50x50 to 150x150 using

### **COPY\_IMAGE\_RESIZE\_LO\_QUALITY**

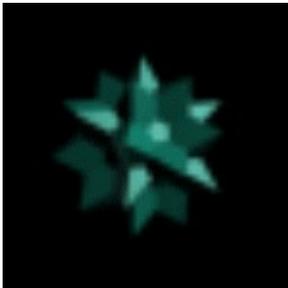


### **COPY\_IMAGE\_RESIZE\_HI\_QUALITY**

Resize source image to the destination map size (final quality).

This is a resize from 50x50 to 150x150 using

### **COPY\_IMAGE\_RESIZE\_HI\_QUALITY**



### **COPY\_IMAGE\_USE\_CUSTOM**

Resize based on the Image Input Options (BitmapInfo pointer).

## Pixel Storage Types

The following structures are defined for storing pixel data:

```
typedef struct {  
    BYTE r,g,b;  
} BMM_Color_24;
```

This is used for storing 24 bit color: 8 bits each for Red, Green, and Blue.

```
typedef struct {  
    BYTE r,g,b,a;  
} BMM_Color_32;
```

This is used for storing 32 bit color: 8 bits each for Red, Green, Blue, and Alpha.

```
typedef struct {  
    WORD r,g,b;  
} BMM_Color_48;
```

This is used for storing 48 bit color: 16 bits each for Red, Green, and Blue.

```
typedef struct {  
    WORD r,g,b,a;  
} BMM_Color_64;
```

This is used for storing 64 bit color: 16 bits each for Red, Green, Blue, and Alpha.

## Class PixelBuf

See Also: [Template Class PixelBufT](#).

class PixelBuf

### Description:

This class lets you set up a buffer for pixels that will automatically deallocate the buffer when it goes out of scope. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**inline PixelBuf(int width)**

#### Remarks:

Constructor. This allocates the pixel buffer using the specified width.

#### Parameters:

**int width**

The number of pixels to allocate for the buffer.

#### Prototype:

**inline ~PixelBuf();**

#### Remarks:

Destructor. The pixel buffer is deallocated.

#### Prototype:

**inline BMM\_Color\_64 \*Ptr();**

#### Remarks:

Returns the address of the pixel buffer.

#### Prototype:

**int Fill(int start, int count, BMM\_Color\_64 color)**

#### Remarks:

Fills the specified portion of the pixel buffer with the specified color.

**Parameters:**

**int start**

The start location for the fill.

**int count**

The number of pixels to fill.

**T color**

The color to use as the fill.

**Return Value:**

Nonzero if filled; otherwise 0.

## Template Class PixelBufT

See Also: [Structure BMM\\_Color\\_24](#), [Structure BMM\\_Color\\_32](#), [Structure BMM\\_Color\\_48](#), [Structure BMM\\_Color\\_64](#).

```
template <Structure T> Structure PixelBufT
```

### Description:

These templated classes allow you to set up a buffer for pixels that will automatically deallocate the buffer when they goes out of scope. All methods of this class are implemented by the system.

Note the following typedefs set up for the standard pixel storage formats.

```
typedef PixelBufT<UBYTE> PixelBuf8;  
typedef PixelBufT<USHORT> PixelBuf16;  
typedef PixelBufT<BMM_Color_24> PixelBuf24;  
typedef PixelBufT<BMM_Color_32> PixelBuf32;  
typedef PixelBufT<BMM_Color_48> PixelBuf48;  
typedef PixelBufT<BMM_Color_64> PixelBuf64;
```

### Methods:

#### Prototype:

```
inline PixelBufT(int width);
```

#### Remarks:

Constructor. This allocates the pixel buffer using the specified width.

#### Parameters:

**int width**

The number of pixels to allocate for the buffer.

#### Prototype:

```
inline ~PixelBufT();
```

#### Remarks:

Destructor. The pixel buffer is deallocated.

#### Prototype:

**inline T\* Ptr();**

**Remarks:**

Returns the address of the pixel buffer.

**Prototype:**

**int Fill(int start, int count, T color)**

**Remarks:**

Fills the specified portion of the pixel buffer with the specified color.

**Parameters:**

**int start**

The start location for the fill.

**int count**

The number of pixels to fill.

**T color**

The color to use as the fill.

**Return Value:**

Nonzero if filled; otherwise 0.

**Operators:**

**Prototype:**

**inline T& operator[](int i);**

**Remarks:**

Array operator. This allows access to the pixel buffer using the [ ] operator.

**Parameters:**

**int i**

The index to access.

## List of Bitmap Filter Types

Specifies the type of filtering to perform. One of the following values:

### **BMM\_FILTER\_NONE**

Specifies no filtering should be performed.

### **BMM\_FILTER\_SUM**

Specifies summed area filtering.

### **BMM\_FILTER\_PYRAMID**

Specifies pyramidal filtering.

### **BMM\_FILTER\_DUMMY**

This is no longer used.

The Pyramidal and Summed Area options provide two methods of pixel averaging that antialias the bitmaps in mapped materials.

Both methods require approximately the same rendering time. Summed-area filtering generally yields superior results but requires much more memory.

Pyramidal filtering requires the program to allocate memory equal to approximately 133% of the size of the bitmap. By comparison, summed-area filtering requires the program to allocate approximately 400% of the size of the bitmap.

Use summed-area filtering only for smaller bitmaps, and avoid using any more such bitmaps in a scene than necessary.

Pyramidal filtering is quite adequate for most purposes. However, because it applies filtering as a function of distance, irregular antialiasing may occur on detailed texture maps that are applied to a plane receding into the distance.

The effect of pyramidal filtering on extreme perspectives such as this is even more noticeable in animations, where portions of the texture map appear to "swim."

## List of Video Color Check Utilities

These functions may be used to correct a pixel with RGB values that will give "unsafe" values of chrominance signal or composite signal amplitude when encoded into an NTSC or PAL color signal. This happens for certain high-intensity high-saturation colors that are rare in real scenes, but can easily be present in computer generated images.

### Prototype:

```
void BuildHotTable(int video_type = VID_NTSC);
```

### Remarks:

Implemented by the System.

A developer will never to need to call this method. It is maintained by MAX. The table that is stored depends only on the state of the NTSC/PAL switch and 3ds max updates it whenever this switch is changed, and on startup.

### Parameters:

```
int video_type = VID_NTSC
```

The type of color checking to perform. One of the following values:

```
VID_NTSC
```

```
VID_PAL
```

### Prototype:

```
int HotLimit(Color48 *thepix, int method = HOT_SCALE_LUM);
```

### Remarks:

Implemented by the System.

This method is called to perform the video color check for each pixel.

### Parameters:

```
Color48 *thepix
```

The pixel to check and correct if necessary.

```
int method = HOT_SCALE_LUM
```

One of the following values:

```
HOT_FLAG
```

Flag the pixel as black.

**HOT\_SCALE\_LUM**

Correct by scaling the luminance.

**HOT\_SCALE\_SAT**

Correct by scaling the saturation.

**Return Value:**

Nonzero if the color was corrected; otherwise if no problems then zero.

## Class SetXFormPacket

See Also: [Class Control](#), [Class Matrix3](#), [Class Quat](#), [Class AngAxis](#), [Class Point3](#).

class SetXFormPacket

### Description:

This class is used to allow a transform (**Matrix3**) controller to know that it is being specifically moved, rotated, or scaled.

When **SetValue()** is called on a controller, the **val** pointer is passed in for a certain data type. For a transform (**Matrix3**) controller **SetValue()** passes in a pointer to an instance of this. This provides higher level information to the transform controller than what is provided by passing a matrix. For example, if rotation is taking place, the **XFORM\_ROTATE** command would be used. In this way the PRS transform controller would not make position or scale keys since it knows only rotation is taking place. Typically one of the different constructors is used depending on the command needed. All methods of this class are implemented by the system.

### Data Members:

public:

#### **SetXFormCommand command;**

The command. The transform controller takes the **val** pointer and casts it to an instance of this class and looks at this data member to see which operation is being performed.

One of the following values:

#### **XFORM\_MOVE**

The move command. An incremental move is being applied to the matrix.

#### **XFORM\_ROTATE**

The rotate command. An incremental rotation is being applied to the matrix.

#### **XFORM\_SCALE**

The scale command. An incremental scaling is being applied to the matrix.

#### **XFORM\_SET**

To just set the matrix without telling the controller any other higher level information this command may be used. This just sets the value of the

matrix (it is not incremental). Any time a node modifies a **Matrix3** controller, it will set the method to get **CTRL\_RELATIVE**, and the packet command is set to **XFORM\_SET**.

**Matrix3 tmParent;**

The parent matrix.

**Matrix3 tmAxis;**

This usually represents the coordinate system one is moving, rotating, or scaling in. However, if the **command** is **XFORM\_SET**, then **tmAxis** is the actual matrix being set.

**Point3 p;**

If the **command** is **XFORM\_MOVE** or **XFORM\_SCALE**, then this contains the amount of the move or scale.

**Quat q;**

If the command is **XFORM\_ROTATE** then this contains the amount of the rotation.

**AngAxis aa;**

If the command is **XFORM\_ROTATE** this will also contain the amount of the rotation. This form can represent multiple revolutions however (as opposed to **Quat q**).

**BOOL localOrigin;**

Indicates the local axis is being used. If TRUE it is; otherwise it is not. If the rotation or scaling is occurring about the pivot point this is TRUE.

**Methods:**

**Prototype:**

```
SetXFormPacket(const Matrix3& mat,const Matrix3&
par=Matrix3(1))
```

**Remarks:**

Constructor. The **XFORM\_SET** command.

**Parameters:**

**const Matrix3& mat**

The **tmAxis** value.

**const Matrix3& par=Matrix3(1)**

The **tmParent** value.

**Prototype:**

**SetXFormPacket(Point3 pt, const Matrix3& par=Matrix3(1),  
const Matrix3& a=Matrix3(1))**

**Remarks:**

Constructor. The **XFORM\_MOVE** command.

**Parameters:**

**Point3 pt**

The **p** value.

**const Matrix3& par=Matrix3(1)**

The **tmParent** value.

**const Matrix3& a=Matrix3(1)**

The **tmAxis** value.

**Prototype:**

**SetXFormPacket(Quat qt, BOOL l, const Matrix3&  
par=Matrix3(1),  
const Matrix3& a=Matrix3(1))**

**Remarks:**

Constructor. The **XFORM\_ROTATE** command.

**Parameters:**

**Quat qt**

The **q** value.

**BOOL l**

The **localOrigin** value.

**const Matrix3& par=Matrix3(1)**

The **tmParent** value.

**const Matrix3& a=Matrix3(1)**

The **tmAxis** value.

**Prototype:**

**SetXFormPacket(AngAxis aA, BOOL l, const Matrix3&  
par=Matrix3(1),  
const Matrix3& a=Matrix3(1))**

**Remarks:**

Constructor. The **XFORM\_ROTATE** command.

**Parameters:**

**AngAxis aA**

The **aa** value.

**BOOL l**

The **localOrigin** value.

**const Matrix3& par=Matrix3(1)**

The **tmParent** value.

**const Matrix3& a=Matrix3(1)**

The **tmAxis** value.

**Prototype:**

**SetXFormPacket(Point3 pt, BOOL l, const Matrix3&  
par=Matrix3(1),  
const Matrix3& a=Matrix3(1))**

**Remarks:**

Constructor. The **XFORM\_SCALE** command.

**Parameters:**

**Point3 pt**

The **p** value.

**BOOL l**

The **localOrigin** value.

**const Matrix3& par=Matrix3(1)**

The **tmParent** value.

**const Matrix3& a=Matrix3(1)**

The **tmAxis** value.

**Prototype:**

**SetXFormPacket();**

**Remarks:**

Constructor. This constructor is provided in case you want to set the data members yourself.

## Class IKDeriv

See Also: [Class Control](#).

class IKDeriv

### Description:

This class provides method that a plug-in calls in its implementation of the Control method **CompDerivs()**. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual int NumEndEffectors()=0;**

#### Remarks:

This method returns the number of end effectors. There may be multiple end effectors if there is branching in the IK chain. For example if the plug-in is a controller controlling a torso there might be two end effectors - the two feet. The plug-ins implementation of **CompDerivs()** should loop through each end effector and call **DP()** and **DR()** for each end effector. Thus this method tells the plug-in how many times it needs to loop.

#### Return Value:

The number of end effectors

#### Prototype:

**virtual Point3 EndEffectorPos(int index)=0;**

#### Remarks:

If a plug-in needs to know the position of an end effector to calculate its derivative it may call this method to retrieve it. This method is used to return the position of the end effector whose index is passed.

#### Parameters:

**int index**

The index of the end effector whose position will be returned.

**Prototype:**

**virtual void DP(Point3 dp,int index)=0;**

**Remarks:**

The plug-in calls this method to specify the derivative of the position of the end effector with respect to the parameter whose index is passed.

**Parameters:****Point3 dp**

The derivative of the position of the end effector with respect to the parameter.

**int index**

The index of the end effector.

**Prototype:**

**virtual void DR(Point3 dr,int index)=0;**

**Remarks:**

Implemented by the Plug-In.

Allows the plug-in to specify the derivative of the orientation in terms of Euler angle of the end effector with respect to the parameter.

**Parameters:****Point3 dr**

The derivative of the orientation in terms of Euler angles of the end effector with respect to the parameter.

**int index**

The index of the end effector.

**Prototype:**

**virtual void NextDOF()=0;**

**Remarks:**

This method is called after a plug-in has called the above methods **DP()** and **DR()** for one of its parameters and it needs to call them again for the next parameter.

## Class IKEnumCallback

See Also: [Class Control](#).

class IKEnumCallback

### Description:

This class is for enumerating IK parameters. This callback is called once for each parameter a controller has. This callback is implemented by the system and passed into the method **EnumIKParams()** of the controller.

### Methods:

### Prototype:

```
virtual void proc(Control *c, int index)=0;
```

### Remarks:

Implemented by the System.

The plug-in calls this method once for each parameter (degree of freedom it has). It passes a pointer to itself and the index of the parameter.

### Parameters:

#### **Control \*c**

The controller itself is passed here.

#### **int index**

The index of the parameter. For example a position controller with three degrees of freedom (X, Y, Z) would call this method three times passing it and index of 0, then 1, then 2.

## Class JointParams

See Also: Class [AnimProperty](#), [Class Control](#), [Class InterpCtrlUI](#).

```
class JointParams : public AnimProperty
```

### Description:

This class handles the data storage and user interface for inverse kinematic joint parameters. The default 3ds max controllers use this data structure to store their IK information. Plug-in controllers don't have to unless they want to.

Note the following is a dialog proc for handling joint parameters that is exported for use by plug-ins.

```
BOOL CALLBACK JointParamDlgProc(HWND hWnd,UINT  
message,  
WPARAM wParam,LPARAM lParam);
```

### Data Members:

```
public:
```

```
float *min, *max;
```

Pointers to an array of floats corresponding to the number of degrees of freedom These are the From and To parameters.

```
float *damping;
```

Pointer to an array of floating point Damping parameters for each degree of freedom.

```
float scale;
```

This is a scale factor applied to the values in the spinner edit fields. This is to make them more sensible to the user. For example a percentage that is stored internally as 0.0 to 1.0 could be presented to the user as 0.0 to 100.0 by using a scale of 100.

```
DWORD flags;
```

One or more of the following values:

```
JNT_XACTIVE
```

```
JNT_YACTIVE
```

```
JNT_ZACTIVE
```

```
JNT_XLIMITED
```

```
JNT_YLIMITED
```

**JNT\_ZLIMITED**  
**JNT\_XEASE**  
**JNT\_YEASE**  
**JNT\_ZEASE**  
**JNT\_LIMITEXACT**  
**JNT\_ROLLOPEN**  
**JNT\_ROT**  
**JNT\_POS**

**int dofs;**

The number of degrees of freedom the plug-in has.

**Operators:**

**Prototype:**

**JointParams& operator=(JointParams& j);**

**Remarks:**

Assignment operator.

**Methods:**

**Prototype:**

**JointParams(DWORD type=JNT\_POS,int dofs=3,float s=1.0f);**

**Remarks:**

Constructor. The data members are initialized to the values passed.

**Prototype:**

**JointParams(const JointParams &j);**

**Remarks:**

Constructor. The data members are initialized to those of the JointParam passed.

**Prototype:**

**~JointParams();**

**Remarks:**

Destructor.

**Prototype:**

**DWORD ID();**

**Remarks:**

Implemented by the System.

Returns the ID of the AnimProperty - **PROPID\_JOINTPARAMS**

**Prototype:**

**BOOL IsDefault();**

**Remarks:**

Implemented by the System.

Returns TRUE if the current state of the parameters are the defaults.

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

Implemented by the System.

This method may be called to save the joint properties to the 3ds max file.

**Parameters:**

**ISave \*isave**

This pointer may be used to call methods to write data to disk. See [Class ISave](#).

**Return Value:**

One of the following values:

**IO\_OK** - The result was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

Implemented by the System.

This method is called to load the joint properties from the 3ds max file.

**Parameters:**

**ILoad \*iLoad**

This pointer may be used to call methods to load data from disk. See [Class ILoad](#).

**Return Value:**

One of the following values:

**IO\_OK** - The result was acceptable - no errors.

**IO\_ERROR** - This is returned if an error occurred.

**Prototype:**

**float ConstrainInc(int index,float v,float delta);**

**Remarks:**

Implemented by the System.

This method applies constraints to the given delta based on parameters and the current value v. It uses the current min/max limits to constrain the result:  $v+\text{delta}$  so that  $v+\text{delta} < \text{max}$  and  $v+\text{delta} > \text{min}$ . It returns a new delta such that the previous will both be TRUE. If ease is turned on, then the values will be slowed down as they approach the limits. It also applies the damping if turned on.

**Parameters:**

**int index**

This is the index of the parameter. For example on a position controller the index could be 0 (x), 1 (y), or 2 (z).

**float v**

The current value of the parameter.

**float delta**

The increment to apply.

**Return Value:**

A new delta value. Usually it will return **delta**, but if the value was constrained, then the value may be smaller or larger.

**Prototype:**

## **BOOL Active(int i);**

### **Remarks:**

Implemented by the System.

Returns TRUE if the specified joint is active; otherwise FALSE.

### **Parameters:**

**int i**

One of the following values:

**0** = X

**1** = Y

**2** = Z

### **Return Value:**

TRUE if the joint is active; otherwise FALSE.

### **Prototype:**

**BOOL Limited(int i)**

### **Remarks:**

Implemented by the System.

Returns TRUE if the joint is limited; otherwise FALSE.

### **Parameters:**

**int i**

One of the following values:

**0** = X

**1** = Y

**2** = Z

### **Return Value:**

TRUE if the joint is limited; otherwise FALSE.

### **Prototype:**

**BOOL Ease(int i)**

### **Remarks:**

Implemented by the System.

Returns TRUE if the joint has the Ease property set; otherwise FALSE.

**Parameters:**

**int i**

One of the following values:

**0** = X

**1** = Y

**2** = Z

**Return Value:**

TRUE if the joint has the Ease property set; otherwise FALSE.

**Prototype:**

**DWORD Type()**

**Remarks:**

Implemented by the System.

Returns the type of joint, sliding or rotation. This will either be **JNT\_POS** for sliding joints or **JNT\_ROT** for rotational joints.

**Prototype:**

**BOOL RollupOpen()**

**Remarks:**

Implemented by the System.

Returns TRUE if the rollup page is open; otherwise FALSE.

**Prototype:**

**void SetActive(int i,BOOL s)**

**Remarks:**

Implemented by the System.

Sets the specified joint to the specified active or inactive state.

**Parameters:**

**int i**

One of the following values:

**0** = X

**1** = Y

**2 = Z**

**BOOL s**

TRUE to set the joint active; otherwise FALSE.

**Prototype:**

**void SetLimited(int i,BOOL s)**

**Remarks:**

Implemented by the System.

Sets the specified joint to the specified limited or not limited state.

**Parameters:**

**int i**

One of the following values:

**0 = X**

**1 = Y**

**2 = Z**

**BOOL s**

TRUE to set the joint as limited; otherwise FALSE.

**Prototype:**

**void SetEase(int i,BOOL s)**

**Remarks:**

Implemented by the System.

Sets the specified joint to the specified eased or not eased state.

**Parameters:**

**int i**

One of the following values:

**0 = X**

**1 = Y**

**2 = Z**

**BOOL s**

TRUE to set the joint as eased; otherwise FALSE.

**Prototype:**

**void SetType(DWORD type)**

**Remarks:**

Implemented by the System.

Sets the type of joint.

**Parameters:**

**DWORD type**

Specifies the type of joint. One of the following values:

**JNT\_POS** - Sliding joint.

**JNT\_ROT** - Rotating joint.

**Prototype:**

**void SetRollOpen(BOOL open)**

**Remarks:**

Implemented by the System.

Set the rollup page as open or closed.

**Parameters:**

**BOOL open**

TRUE to open the page; FALSE to close it.

**Prototype:**

**virtual void SpinnerChange(InterpCtrlUI \*ui,WORD id,  
ISpinnerControl \*spin,BOOL interactive);**

**Remarks:**

Implemented by the Plug-In.

This is called when the user is interactively manipulating one of the spinner controls or enters a value into a spinner's edit field. This method has a default implementation.

**Parameters:**

**InterpCtrlUI \*ui**

This is simply a container class to hold some data while the controllers parameters are being edited.

**WORD id**

The spinner control id.

**ISpinnerControl \*spin**

A pointer to the spinner control.

**BOOL interactive**

TRUE if the user is doing an interactive adjustment; otherwise FALSE.

**These methods manage the joint parameters dialog.**

**Prototype:**

**void InitDialog(InterpCtrlUI \*ui);**

**Remarks:**

Implemented by the System.

This is used internally.

**Prototype:**

**void EndDialog(InterpCtrlUI \*ui,BOOL dontDel=FALSE);**

**Remarks:**

Implemented by the System.

This is used internally.

**Prototype:**

**void SpinnerDown(InterpCtrlUI \*ui,WORD id,ISpinnerControl \*spin);**

**Remarks:**

Implemented by the System.

This is used internally.

**Prototype:**

**void SpinnerUp(InterpCtrlUI \*ui,WORD id,ISpinnerControl \*spin,  
BOOL accept);**

**Remarks:**

Implemented by the System.

This is used internally.

**Prototype:**

**void Command(InterpCtrlUI \*ui,WORD notify, WORD id,**

**HWND hCtrl);**

**Remarks:**

Implemented by the System.

This is used internally.

**Prototype:**

**void EnableDisable(InterpCtrlUI \*ui);**

**Remarks:**

Implemented by the System.

This is used internally.

## Class EaseCurveList

See Also: [Class ReferenceTarget](#), [Class Control](#).

```
class EaseCurveList : public ReferenceTarget
```

### Description:

This class represents a list of ease curves.

The macro used to access this class is defined as follows:

```
#define GetEaseListInterface(anim) ((EaseCurveList*)anim->GetInterface(I_EASELIST))
```

This may be used to access the methods of this class as follows:

```
EaseCurveList *el = GetEaseListInterface(client);  
if (el) {  
    int num = el->NumEaseCurves();  
    // ...
```

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
EaseCurveList();
```

#### Remarks:

Constructor.

#### Prototype:

```
~EaseCurveList();
```

#### Remarks:

Destructor. All the references are deleted from this class.

#### Prototype:

```
TimeValue ApplyEase(TimeValue t,Interval &valid);
```

#### Remarks:

Returns a **TimeValue** that reflects the **TimeValue** passed modified by each of the enabled ease curves in the list.

**Parameters:**

**TimeValue t**

The base time which is eased by the curves.

**Interval &valid**

The validity interval which is updated by each of the ease curves in the list.

**Prototype:**

**void AppendEaseCurve(Control \*cont);**

**Remarks:**

Adds the specified ease curve to the end of the ease curve list.

**Parameters:**

**Control \*cont**

Points to the ease curve to append.

**Prototype:**

**void DeleteEaseCurve(int i);**

**Remarks:**

Deletes the 'i-th' ease curve in the list.

**Parameters:**

**int i**

The index of the ease curve to delete.

**Prototype:**

**void DisableEaseCurve(int i);**

**Remarks:**

Disables the 'i-th' ease curve in the list.

**Parameters:**

**int i**

The index of the ease curve to disable.

**Prototype:**

**void EnableEaseCurve(int i);**

**Remarks:**

Enables the 'i-th' ease curve in the list.

**Parameters:**

**int i**

The index of the ease curve to enable.

**Prototype:**

**BOOL IsEaseEnabled(int i);**

**Remarks:**

Returns TRUE if the 'i-th' ease curve is enabled; otherwise FALSE.

**Parameters:**

**int i**

The index of the ease curve to check.

**Prototype:**

**int NumEaseCurves();**

**Remarks:**

Returns the number of ease curves in the list.

## Class MultCurveList

See Also: [Class ReferenceTarget](#), [Class Control](#).

```
class MultCurveList : public ReferenceTarget
```

### Description:

This class is a list of multiplier curves.

The macro used to access this class is defined as follows:

```
#define GetMultListInterface(anim) ((MultCurveList*)anim->GetInterface(I_MULTLIST))
```

This may be used to access the methods of this class as follows:

```
MultCurveList *ml = GetMultListInterface(client);  
if (ml) {  
    int num = ml->NumMultCurves();  
    // ...
```

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
MultCurveList();
```

#### Remarks:

Constructor.

#### Prototype:

```
~MultCurveList();
```

#### Remarks:

Destructor. All references are removed from this class.

#### Prototype:

```
float GetMultVal(TimeValue t,Interval &valid);
```

#### Remarks:

This method starts with a value of **1.0f**, multiplies it by each enabled multiplier curve value in the list, and returns the resulting value.

**Parameters:**

**TimeValue t**

The time at which to get the multiplier values.

**Interval &valid**

The interval that is adjusted to reflect the validity of all the multiplier curve controllers validity.

**Prototype:**

**void AppendMultCurve(Control \*cont);**

**Remarks:**

Adds the specified multiplier curve to the end of the multiplier curve list.

**Parameters:**

**Control \*cont**

Points to the multiplier curve to append.

**Prototype:**

**void DeleteMultCurve(int i);**

**Remarks:**

Deletes the 'i-th' multiplier curve.

**Parameters:**

**int i**

The index of the multiplier curve to delete.

**Prototype:**

**void DisableMultCurve(int i);**

**Remarks:**

Disables the 'i-th' multiplier curve.

**Parameters:**

**int i**

The index of the multiplier curve to disable.

**Prototype:**

**void EnableMultCurve(int i);**

**Remarks:**

Enables the 'i-th' multiplier curve.

**Parameters:**

**int i**

The index of the multiplier curve to enable.

**Prototype:**

**BOOL IsMultEnabled(int i);**

**Remarks:**

Returns TRUE if the 'i-th' multiplier curve is enabled; otherwise FALSE.

**Parameters:**

**int i**

The index of the multiplier curve to check.

**Prototype:**

**int NumMultCurves();**

**Remarks:**

Returns the number of multiplier curves in the list.

# Class INoiseControl

See Also: [Class Control](#).

```
class INoiseControl : public StdControl
```

## Description:

This class is available in release 2.0 and later only.

This class provides access to noise controller's parameters. All noise controllers are derived from this class.

All methods of this class are implemented by the system.

## Methods:

### Prototype:

```
virtual void SetSeed(int seed)=0;
```

### Remarks:

Sets the seed value for the noise controller.

### Parameters:

**int seed**

The seed value, greater than or equal to zero.

### Prototype:

```
virtual int GetSeed()=0;
```

### Remarks:

Returns the seed value.

### Prototype:

```
virtual void SetFrequency(float f)=0;
```

### Remarks:

Sets the frequency parameter.

### Parameters:

**float f**

The value to set, greater than zero.

**Prototype:**

**virtual float GetFrequency()=0;**

**Remarks:**

Returns the frequency value.

**Prototype:**

**virtual void SetFractal(BOOL f)=0;**

**Remarks:**

Sets the fractal setting on or off.

**Parameters:**

**BOOL f**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetFractal()=0;**

**Remarks:**

Returns the state of the fractal setting. TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetRoughness(float f)=0;**

**Remarks:**

Sets the roughness setting.

**Parameters:**

**float f**

The value to set, between 0.0 and 1.0.

**Prototype:**

**virtual float GetRoughness()=0;**

**Remarks:**

Returns the roughness setting.

**Prototype:**

**virtual void SetRampIn(TimeValue in)=0;**

**Remarks:**

Sets the ramp in setting.

**Parameters:**

**TimeValue in**

The value to set, greater than or equal to zero.

**Prototype:**

**virtual TimeValue GetRampIn()=0;**

**Remarks:**

Returns the ramp in setting.

**Prototype:**

**virtual void SetRampOut(TimeValue out)=0;**

**Remarks:**

Sets the ramp out setting.

**Parameters:**

**TimeValue out**

The value to set, greater than or equal to zero.

**Prototype:**

**virtual TimeValue GetRampOut()=0;**

**Remarks:**

Returns the ramp out setting.

**Prototype:**

**virtual void SetPositiveOnly(int which,BOOL onOff)=0;**

**Remarks:**

Sets the positive only setting (>0) for the specified axis to the specified value.

**Parameters:**

**int which**

Specifies the axis. One of the following values:

**0:** X, **1:** y, **2:** Z.

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

```
virtual BOOL GetPositiveOnly(int which)=0;
```

**Remarks:**

Returns the positive only setting (>0) for the specified axis to the specified value.

**Parameters:****int which**

Specifies the axis. One of the following values:

**0:** X, **1:** y, **2:** Z.

**Prototype:**

```
virtual Control *GetStrengthController()=0;
```

**Remarks:**

Returns a pointer to the controller for the strength parameter.

**Prototype:**

```
virtual void SetStrengthController(Control *c)=0;
```

**Remarks:**

Sets the controller used for the strength parameter.

**Parameters:****Control \*c**

Points to the controller to set.

# Class ISurfPosition

See Also: [Class Control](#), [Class INode](#).

```
class ISurfPosition : public Control
```

## Description:

This class is available in release 2.0 and later only.

This class provides access to the surface position controller's parameters.

The following values may be used to access the surface controller's references.

**SURFCONT\_U\_REF**

**SURFCONT\_V\_REF**

**SURFCONT\_SURFOBJ\_REF**

All methods of this class are implemented by the system.

## Methods:

### Prototype:

```
virtual void SetSurface(INode *node)=0;
```

### Remarks:

Sets the node that this controller uses as the surface object.

### Parameters:

**INode \*node**

Points to the node to set.

### Prototype:

```
virtual int GetAlign()=0;
```

### Remarks:

Returns the alignment setting.

### Return Value:

One of the following values:

**0:** No Alignment.

**1:** Align to U.

**2:** Align to V.

**Prototype:**

**virtual void SetAlign(int a)=0;**

**Remarks:**

Sets the alignment setting.

**Parameters:**

**int a**

One of the following values:

**0:** No Alignment.

**1:** Align to U.

**2:** Align to V.

**Prototype:**

**virtual BOOL GetFlip()=0;**

**Remarks:**

Returns the flip setting. TRUE if on; FALSE if off.

**Prototype:**

**virtual void SetFlip(BOOL f)=0;**

**Remarks:**

Sets the flip setting.

**Parameters:**

**BOOL f**

TRUE for on; FALSE for off.

## Class ILinkCtrl

See Also: [Class Control](#).

class ILinkCtrl : public Control, public FPMixinInterface

### Description:

This class is available in release 2.0 and later only.

This class represents the interface to the Link Controller. You can obtain a pointer to the link controller interface using; **GetLinkConstInterface(cd)**.

This macro will return

**(LinkConstTransform\*)(CD)-**

**>GetFPInterface(LINK\_CONSTRAINT\_INTERFACE).**

Developers may use the following values to access the references of the Link controller.

**LINKCTRL\_CONTROL\_REF**

The TM controller

**LINKCTRL\_FIRSTPARENT\_REF**

The index of the first parent node.

**LINKCTRL\_PBLOCK\_REF**

The parameter block.

All methods of this class are Implemented by the System.

### Methods:

public:

### Prototype:

**virtual int GetNumTargets()=0;**

### Remarks:

This method is available in release 4.0 and later only.

This method returns the number of parents (links).

### Prototype:

**virtual TimeValue GetLinkTime(int i)=0;**

### Remarks:

Returns the start time associated with the 'i-th' link.

**Parameters:**

**int i**

Specifies which parent (link).

**Prototype:**

**virtual void SetLinkTime(int i, TimeValue t)=0;**

**Remarks:**

Sets the start time associated with the 'i-th' link. See **LinkTimeChanged()** below. Note: This method also sorts the indices according to increasing time values.

**Parameters:**

**int i**

Specifies which parent (link).

**TimeValue t**

The time to set.

**Prototype:**

**virtual void LinkTimeChanged()=0;**

**Remarks:**

This method should be called after changing link times. Note: This method also sorts the indices according to increasing time values.

**Prototype:**

**virtual void AddNewLink(INode \*node, TimeValue t)=0;**

**Remarks:**

Adds a new link at the specified time.

**Parameters:**

**INode \*node**

Points to the node of the link to add.

**TimeValue t**

The time to change to this link.

**Prototype:**

**virtual void DeleteTarget(int selection)=0;**

**Remarks:**

This method will delete the specified link.

**Parameters:**

**int selection**

Specifies which link to delete.

**Prototype:**

**virtual int GetFrameNumber(int targetNumber)=0;**

**Remarks:**

This method returns the start frame of the specified target.

**Parameters:**

**int targetNumber**

The target number for which to get the start frame.

**Prototype:**

**virtual BOOL SetFrameNumber(int targetNumber, int  
frameNumber)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the start frame of the specified target.

**Parameters:**

**int targetNumber**

The target number for which to set the start frame.

**int frameNumber**

The time value.

**Return Value:**

TRUE if the start frame is set, otherwise FALSE.

**Prototype:**

**virtual BOOL SetTarget(INode \*target, int targetNumber)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

Sets one of the link nodes that the link constraint controller targets, specified by **targetNumber**. If **targetNumber** is greater than the number of targets in the current list, it returns a FALSE. In this case use the function **AppendTarget**.

**Parameters:**

**INode \*target**

Points to the node to follow

**int targetNumber**

The node number in the link target list to be set.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual BOOL AddTarget(INode \*target, int frameNo)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to append a target.

**Parameters:**

**INode \*target**

The target node to append.

**int frameNo**

The frame time.

**Return Value:**

TRUE if the target was appended, otherwise FALSE.

**Prototype:**

**virtual INode\* GetNode(int targetNumber)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method returns one of the link nodes that the link constraint controller targets, specified by **targetNumber**.

**Parameters:**

**int targetNumber**

The node number in the link target list to be obtained.

## Class ILookatControl

See Also: [Class Control](#).

```
class ILookatControl : public Control
```

### Description:

This class is available in release 2.0 and later only.

This class provides access to the LookAt Controller parameters.

Developers may use the following values to access the references of the Look At controller.

**LOOKAT\_TARGET\_REF**

**LOOKAT\_POS\_REF**

**LOOKAT\_ROLL\_REF**

**LOOKAT\_SCL\_REF**

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
virtual void SetFlip(BOOL f)=0;
```

#### Remarks:

Sets the flip setting.

#### Parameters:

**BOOL f**

TRUE for on; FALSE for off.

#### Prototype:

```
virtual BOOL GetFlip()=0;
```

#### Remarks:

Returns the state of the flip setting. TRUE if on; FALSE if off.

#### Prototype:

```
virtual void SetAxis(int a)=0;
```

#### Remarks:

Sets the axis setting.

**Parameters:**

**int a**

One of the following values:

**0:** X axis.

**1:** Y axis.

**2:** Z axis.

**Prototype:**

**virtual int GetAxis()=0;**

**Remarks:**

Returns the axis setting.

**Return Value:**

One of the following values:

**0:** X axis.

**1:** Y axis.

**2:** Z axis.

## List of Additional Controller Related Functions

See Also: [Class Control](#), [Class StdControl](#).

### Description:

The following functions may be used to control the playing and suspension of animation, get and set the start and end animation times, and returns new instances of default controllers of various types.

### Functions:

The functions below allow a developer to control the state of the 'Animate' button in the user interface, and also set if animate mode is on or off. The pseudo code below demonstrate how these methods might be used. If you need to do something associated with a controller but did not want keys to be generated you could write:

```
SuspendAnimate();
```

```
AnimateOff();
```

```
// do some things...
```

```
ResumeAnimate();
```

This ensures that animate mode is unchanged after you're through. The animate button never changes appearance so the user doesn't have to be aware that anything is happening.

### Prototype:

```
int Animating();
```

### Remarks:

Determines if the animate button is on. Returns nonzero if on; zero if off.

### Prototype:

```
void AnimateOn();
```

### Remarks:

This method logically turns animating on but does not change the appearance of the button.

### Prototype:

**void AnimateOff();**

**Remarks:**

This method logically turns animating off but does not change the appearance of the button.

**Prototype:**

**void SuspendAnimate();**

**Remarks:**

Suspend the animation from running. This method uses a stack so if several calls are made to the **SuspendAnimate()**, they must all be resumed before animation will resume.

**Prototype:**

**void ResumeAnimate();**

**Remarks:**

Resume suspended animation. The method **SuspendAnimate()** uses a stack so if several calls are made to **SuspendAnimatie()**, they must all be resumed using this method before the animation will resume.

**Prototype:**

**TimeValue GetAnimStart();**

**Remarks:**

Retrieves the start time of the animation.

**Prototype:**

**TimeValue GetAnimEnd();**

**Remarks:**

Retrieves the end time of the animation.

**Prototype:**

**void SetAnimStart(TimeValue s);**

**Remarks:**

Sets the start time of the animation.

**Prototype:**

**void SetAnimEnd(TimeValue e);**

**Remarks:**

Sets the end time of the animation.

**Prototype:**

**Control \*NewDefaultFloatController();**

**Remarks:**

Returns an instance of the default float controller.

**Prototype:**

**Control \*NewDefaultPoint3Controller();**

**Remarks:**

Returns an instance of the default Point3 controller.

**Prototype:**

**Control \*NewDefaultMatrix3Controller();**

**Remarks:**

Returns an instance of the default Matrix3 controller.

**Prototype:**

**Control \*NewDefaultPositionController();**

**Remarks:**

Returns an instance of the default position controller.

**Prototype:**

**Control \*NewDefaultRotationController();**

**Remarks:**

Returns an instance of the default rotation controller.

**Prototype:**

**Control \*NewDefaultScaleController();**

**Remarks:**

Returns an instance of the default scale controller.

**Prototype:**

**Control \*NewDefaultColorController();**

**Remarks:**

Returns an instance of the default color controller.

**Prototype:**

**Control \*NewDefaultBoolController();**

**Remarks:**

Returns an instance of the default boolean controller.

**Prototype:**

**Control\* CreateInterpFloat();**

**Remarks:**

Creates and returns a pointer to a new Bezier float controller.

**Prototype:**

**Control\* CreateInterpPosition();**

**Remarks:**

Creates and returns a pointer to a new Bezier position controller.

**Prototype:**

**Control\* CreateInterpPoint3();**

**Remarks:**

Creates and returns a pointer to a new Bezier Color controller.

**Prototype:**

**Control\* CreateInterpRotation();**

**Remarks:**

Creates and returns a pointer to a new TCB Rotation controller.

**Prototype:**

**Control\* CreateInterpScale();**

**Remarks:**

Creates and returns a pointer to a new Bezier Scale controller.

**Prototype:**

**Control\* CreatePRSControl();**

**Remarks:**

Creates and returns a pointer to a new PRS transform controller.

**Prototype:**

**Control\* CreateLookatControl();**

**Remarks:**

Creates and returns a pointer to a new Look At transform controller.

**Prototype:**

**void SetDefaultController(SClass\_ID sid, ClassDesc \*desc);**

**Remarks:**

Sets the default controller of the specified type to the controller whose ClassDesc is passed.

**Parameters:**

**SClass\_ID sid**

The Super Class ID. See [List of Super Class IDs](#).

**ClassDesc \*desc**

Points to the Class Descriptor for the controller. See [Class ClassDesc](#).

**Prototype:**

**ClassDesc \*GetDefaultController(SClass\_ID sid);**

**Remarks:**

Returns a pointer to the Class Descriptor for the default controller of the specified type.

**Parameters:**

**SClass\_ID sid**

The Super Class ID. See [List of Super Class IDs](#).

**Prototype:**

**void SetDefaultColorController(ClassDesc \*desc);**

**Remarks:**

Sets the default Color controller.

**Parameters:**

**ClassDesc \*desc**

Points to the Class Descriptor for the controller. See [Class ClassDesc](#).

**Prototype:**

**void SetDefaultBoolController(ClassDesc \*desc);**

**Remarks:**

Sets the default Boolean controller.

**Parameters:**

**ClassDesc \*desc**

Points to the Class Descriptor for the controller. See [Class ClassDesc](#).

**Prototype:**

**void ApplyScaling(Matrix3 &m, const ScaleValue &v);**

**Remarks:**

This method may be used to apply a ScaleValue to the specified Matrix3. This is used in controller implementation of **GetValue()** when the **GetSetMethod** is **CTRL\_RELATIVE** and the controller is a scale controller.

**The global functions below provide access to the default tangent types for both the Bezier and TCB controllers.**

**Prototype:**

**void GetBezierDefaultTangentType(int &in, int &out);**

**Remarks:**

This function is available in release 2.0 and later only.

Retrieves the default tange types for the Bezier controller.

**Parameters:**

**int &in**

The in tangent value.

**int &out**

The out tangent value.

**Prototype:**

**void SetBezierDefaultTangentType(int in, int out);**

**Remarks:**

This function is available in release 2.0 and later only.

Sets the default tange types for the Bezier controller.

**Parameters:**

**int &in**

The in tangent value.

**int &out**

The out tangent value.

**Prototype:**

**void GetTCBDefaultParams(float &t, float &c, float &b, float &easeIn, float &easeOut);**

**Remarks:**

This function is available in release 2.0 and later only.

Retrieves the default tange types for the TCB controller.

**Parameters:**

**float &t**

The tension setting.

**float &c**

The continuity setting.

**float &b**

The bias setting.

**float &easeIn**

The ease in value.

**float &easeOut**

The ease out value.

**Prototype:**

```
void SetTCBDefaultParams(float t, float c, float b, float easeIn,  
float easeOut);
```

**Remarks:**

This function is available in release 2.0 and later only.

Sets the default tange types for the TCB controller.

**Parameters:****float t**

The tension setting.

**float c**

The continuity setting.

**float b**

The bias setting.

**float easeIn**

The ease in value.

**float easeOut**

The ease out value.

## Class DefaultLight

See Also: [Class Renderer](#), [Class Matrix3](#), [Structure LightState](#).

class DefaultLight

### Description:

This class describes a default light. An array of these default lights is passed into the method **Renderer::Open()**.

### Data Members:

public:f

#### **LightState ls;**

Describes the properties of the light.

#### **Matrix3 tm;**

This is the transformation of the light that controls its position in the scene.

Note: In 3ds max 3.0 the production renderer has been modified so that if a DefaultLight is passed into **Renderer::Open()** with a transformation matrix that is all zeros, the renderer will interpret this to mean that on each frame it should create a light located at the view point, pointing in the view direction. This allows the implementation of the new viewport 1-light option so that it tracks the camera during an animated camera move.

## Structure LightState

See Also: [Class GenLight](#), [Class LightObject](#), [Class Color](#), [Class Matrix3](#).

### Description:

This structure describes the properties of a light.

```
struct LightState {
```

```
    LightType type;
```

One of the following values:

(from **enum LightType { OMNI\_LGT, SPOT\_LGT, DIRECT\_LGT, AMBIENT\_LGT }**);

**OMNI\_LGT** - Omnidirectional

**SPOT\_LGT** - Spot (cone)

**DIRECT\_LGT** - Directional (parallel)

**AMBIENT\_LGT** - Global

```
    Matrix3 tm;
```

The transformation matrix of the light.

```
    Color color;
```

The color of the light (its intensity).

```
    float intens;
```

The multiplier applied to the **color**.

```
    float hotsize;
```

The hotspot size in degrees.

```
    float fallsize;
```

The hotspot falloff size in degrees.

```
    int useNearAtten;
```

This data member is available in release 2.0 and later only.

Nonzero if near attenuation is used; otherwise zero.

```
    float nearAttenStart;
```

This data member is available in release 2.0 and later only.

The near attenuation start value.

```
    float nearAttenEnd;
```

This data member is available in release 2.0 and later only.

The near attenuation end value.

**int useAtten;**

Nonzero if (far) attenuation is used; otherwise zero.

**float attenStart;**

The (far) start attenuation value.

**float attenEnd;**

The (far) end attenuation value.

**int shape;**

One of the following values:

**RECT\_LIGHT**

**CIRCLE\_LIGHT**

**float aspect;**

The aspect ratio of the light.

**BOOL overshoot;**

TRUE if the light supports overshoot; otherwise FALSE.

**BOOL shadow;**

TRUE if shadows are on; otherwise FALSE.

**BOOL on;**

TRUE if the light is on; otherwise FALSE.

**BOOL affectDiffuse;**

TRUE if affect diffuse is on; otherwise FALSE.

**BOOL affectSpecular;**

TRUE if affect specular is on; otherwise FALSE.

**};**

## Class LightDesc

See Also: [Class RenderData](#), [Class LightRayTraversal](#), [Class ObjLightDesc](#), [Class ShadeContext](#), [Class Color](#), [Class Point3](#).

```
class LightDesc : public RenderData
```

### Description:

This class has a method **Illuminate()** used to determine the color and direction of the light striking the point **sc.P()** and a method to get the position of the light. It also has two public data members that determine if the diffuse and specular colors of objects are affected.

### Data Members:

public:

#### **BOOL affectDiffuse;**

This data member is available in release 2.0 and later only.

If TRUE the light affects the diffuse color; otherwise it does not.

#### **BOOL affectSpecular;**

This data member is available in release 2.0 and later only.

If TRUE the light affects the specular color; otherwise it does not.

#### **BOOL ambientOnly;**

This data member is available in release 3.0 and later only.

If TRUE the light affects the ambient color only; otherwise it does not.

#### **DWORD extra;**

This data member is available in release 3.0 and later only.

This is not currently used and is available for use in the future.

### Methods:

### Prototype:

```
LightDesc();
```

### Remarks:

Constructor. The **affectDiffuse** and **affectSpecular** data members are set to TRUE.

### Prototype:

**virtual BOOL Illuminate(ShadeContext& sc, Point3& normal,  
Color& color, Point3 &dir, float &dot\_nl, float &diffuseCoef);**

**Remarks:**

Implemented by the Plug-In.

This method is called to determine the color and direction of illumination from the light that is hitting the point (**sc.P()**).

**Parameters:**

**ShadeContext& sc**

Describes the properties of the point to shade. The point itself is **sc.P()**.

**Point3& normal**

The normal to the surface in camera space.

**Color& color**

The color that is returned. This is the brightness of light striking the point (**sc.P()**)

**Point3 &dir**

The direction that the light is coming from.

**float &dot\_nl**

This provides a bit of optimization as most lights will have to calculate this quantity. This is the dot product between the light vector and the normal to the surface.

**float &diffuseCoef**

This parameter should be set by the Illuminate function. The default value is the same as dot\_nl. It will be used by shading functions instead of dot\_nl to compute the diffuse illumination of the surface. The built-in lights use the new "Contrast" parameter (which has a range of [0..100]) to compute the diffuseCoef from the dot\_nl by the Contrast function:

**// precomputed:**

**float a = contrast/200.0f + 0.5f; // so "a" varies from .5 to 1.0**

**kA = (2.0f-1.0f/a);**

**kB = 1.0f-kA;**

**// called by Illuminate() to compute diffuseCoef from dot\_nl.**

```
float ContrastFunc(float nl) {  
    // the "Bias" function described in Graphics Gems IV, pp. 401ff  
    return (contrast==0.0f)? nl : nl/(kA*nl+kB);  
}
```

**Return Value:**

Returns FALSE if the hitpoint is outside the effective range of the light or if the normal of the surface faces away from the light. This is a hint to the material that the light did not calculate its illumination because it is assumed it wasn't going to be used. If TRUE the point is being illuminated.

**Default Implementation:**

```
{ return 0;}
```

**Prototype:**

```
virtual Point3 LightPosition();
```

**Remarks:**

This method is available in release 2.0 and later only.  
Returns the position of the light.

**Default Implementation:**

```
{ return Point3(0,0,0); }
```

## Class LightRayTraversal

See Also: [Class ObjLightDesc](#).

```
class LightRayTraversal
```

### Description:

This is a callback class that can be given to a **ObjLightDesc** to have a ray traced through the light volume. A plug-in derives a class from this one and passes it as a callback in the **ObjLightDesc** method **TraverseVolume()**. This allows a developer to integrate the illumination of a segment due to a light. **t0** and **t1** define the segment in terms of the given ray.

This is what the 3ds max spotlights do: First they break the segment up into three main pieces. The first piece is from the camera to where the ray intersects the lights cone volume. The callback **Step()** is called once over this segment (**t0** and **t1** will have this first piece). The illumination is constant over this entire segment from **t0** to **t1**. It is a constant black since the light is not illuminating it at all.

The next segment is inside the cone. This segment will be broken up into small pieces. First as it's stepping along it will be between the falloff and the hotspot. The illumination over this segment goes from black to brighter and brighter as it moves towards to hotspot. Across the entire hotspot region the illumination may be constant. Then as it steps from the hotspot to the falloff the illumination will go back down to black.

Inside the hotspot region, if shadows are turned on, the light may be brighter or darker depending on if it's inside a shadow or on the edge of a shadow. The light handles all of this. It takes care of the shadows, attenuation, etc.

Now consider how the 3ds max atmospheric effects such as the volume lights use this information. For each light that they are bound to, they call the method **TraverseVolume()** on the light. The volume light atmospheric effect passes this callback to the **TraverseVolume()** method. The light then calls this **Step()** method of the callback for each partial segment of the ray. Given the illumination on the segment (**illum**) it computes the fog density over that piece. The density may be constant if noise is not turned on, or it may vary if noise is turned on. Using the fog density and the illumination it computes the light reflected off the atmosphere for the segment.

## Methods:

### Prototype:

**virtual BOOL Step(float t0, float t1, Color illum , float distAtten)=0;**

### Remarks:

Implemented by the Plug-In.

This method is called for every step defined by **t0** and **t1**. The illumination over this segment is passed in **illum**.

### Parameters:

#### **float t0**

The start of the segment. This is a distance along the ray. The ray is made up of a point **p** and a unit length direction vector **dir**. The point defined by **t0** is thus **ray.p+t0\*ray.dir**.

#### **float t1**

The end of the segment. This is a distance along the ray. The ray is made up of a point **p** and a unit length direction vector **dir**. The point defined by **t1** is thus **ray.p+t1\*ray.dir**.

#### **Color illum**

The light intensity over the entire segment. It can be assumed that the light intensity is constant for the segment.

#### **float distAtten**

This parameter is available in release 2.0 and later only.

This parameter may be used so that volume effects can use the distance attenuation value as an input variable to their effects. For instance, the volume light uses this to change the fog color based on the distance from the light.

### Return Value:

TRUE to continue; FALSE to halt the integration (stop the traversal).

## Class ParamDlg

See Also: [Class Mtl](#), [Class Texmap](#), [Working with Materials and Textures](#).

class ParamDlg : public InterfaceServer

### Description:

Every MtlBase sub-class defines a ParamDlg to manage its part of the material editor.

### Methods:

#### Prototype:

**virtual Class\_ID ClassID()=0;**

#### Remarks:

Implemented by the Plug-In.

Returns the unique Class\_ID of the plug-in this dialog manages. See [Class Class\\_ID](#).

#### Prototype:

**virtual void SetThing(ReferenceTarget \*m)=0;**

#### Remarks:

Implemented by the Plug-In.

This sets the current material or texture being edited to the material or texture passed.

#### Parameters:

**ReferenceTarget \*m**

The material or texture to set as current.

#### Prototype:

**virtual ReferenceTarget\* GetThing()=0;**

#### Remarks:

Implemented by the Plug-In.

This method returns the current material or texture being edited.

**Prototype:**

**virtual void SetTime(TimeValue t)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called when the current time has changed. This gives the developer an opportunity to update any user interface data that may need adjusting due to the change in time.

**Parameters:**

**TimeValue t**

The new current time.

**Prototype:**

**virtual void ReloadDialog()=0;**

**Remarks:**

Implemented by the Plug-In.

This method should stuff values into all the parameter dialog's controls, edit fields etc. In fact this method is now only called after doing a "Reset". It is also useful inside the material/map when a **NotifyRefChanged()** is processed it may need to be called.

**Prototype:**

**virtual void ActivateDlg(BOOL onOff)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called when the dialog box becomes active or inactive.

Currently this is used when working with color swatch custom controls. Color swatches need to know when the dialog box becomes active or inactive because they change their method of drawing themselves. When the dialog is active, color swatches are drawn in pure color (this requires a color register in the palette). When the dialog becomes inactive the color swatches are drawn using dithered painting. This is needed because there are only 8 available color registers.

A method of the color swatch control is called to indicate if it is in an active or

inactive dialog. See the sample code below. Also see [Class IColorSwatch](#).

**Parameters:**

**BOOL onOff**

TRUE if the dialog is active; otherwise FALSE.

**Sample Code:**

```
void NoiseDlg::ActivateDlg(BOOL onOff) {  
    for (int i=0; i<NCOLS; i++)  
        cs[i]->Activate(onOff);  
}
```

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called by the system to delete an instance of this class. A developer must use the same memory manager to delete the item as they did to allocate it (in **MtlBase::CreateParamDlg()**). For example if a developer used the **new** operator to allocate the memory, this method would be implemented as **{ delete this; }**

**Prototype:**

**virtual int FindSubTexFromHWND(HWND hwnd);**

**Remarks:**

This method is available in release 2.0 and later only.

This method needs to be implemented if the plug-in texmap is using a **TexDADMgr**. It should return the index of the sub-texmap corresponding to the window whose handle is passed. If the handle is not valid return -1.

**Parameters:**

**HWND hwnd**

The window handle to check.

**Default Implementation:**

```
{ return -1;}
```

**Sample Code:**

```
int CellTexParamDlg::FindSubTexFromHWND(HWND hw) {
    if (hw==GetDlgItem(pmap-
>GetHWnd(),IDC_CELLTEX_CELLCOLOR_MAP))
        return 0;
    if (hw==GetDlgItem(pmap-
>GetHWnd(),IDC_CELLTEX_DIVCOL1_MAP))
        return 1;
    if (hw==GetDlgItem(pmap-
>GetHWnd(),IDC_CELLTEX_DIVCOL2_MAP))
        return 2;
    return -1;
}
```

**Prototype:**

```
virtual int FindSubMtlFromHWND(HWND hwnd);
```

**Remarks:**

This method is available in release 2.0 and later only.

This method needs to be implemented if the plug-in material is using a **MtlDADMgr**. It should return the index of the sub-map corresponding to the window whose handle is passed. If the handle is not valid return -1.

**Parameters:**

**HWND hwnd**

The window handle to check.

**Default Implementation:**

```
{ return -1;}
```

## Class UVGen

See Also: [Class MtlBase](#), [Class ShadeContext](#), [Class MapSampler](#), [Class Point2](#), [Class Point3](#), [Class Matrix3](#).

```
class UVGen : public MtlBase
```

### Description:

Most texture maps that use UV coordinates will use an instance of this class. This class encapsulates much of the user interface functionality for setting mirroring, tiling and so on. These are the settings found in the 'Coordinate' and 'Noise' rollup pages. This class generates UV coordinates based on the results of a UV source provided by the plug-in and the user specified transformations. An instance of this class is referenced by all the 2D texture maps. All methods of this class are implemented by the system.

### Methods:

The following four methods are the only ones a plug-in needs to call typically:

### Prototype:

```
virtual AColor EvalUVMap(ShadeContext &sc, MapSampler*  
samp,  
  BOOL filter=TRUE)=0;
```

### Remarks:

This method is called to evaluate the color of the map based on the ShadeContext. It takes into account the users selections for mirroring, tiling, noise, etc. from the UVGen rollups.

### Parameters:

#### **ShadeContext &sc**

Describes the properties of the point to evaluate.

#### **MapSampler\* samp**

This is a callback used to sample the map. The plug-in creates an instance of this class, and implements its methods. A pointer to the instance is passed here. The methods of the class sample the texture for a single UV coordinate.

#### **BOOL filter=TRUE**

If TRUE the texture will be filtered.

### Return Value:

The result as an AColor.

**Prototype:**

```
virtual float EvalUVMapMono(ShadeContext &sc, MapSampler*  
somp,  
  BOOL filter=TRUE)=0;
```

**Remarks:**

This method is called to evaluate the value of the map based on the ShadeContext. It takes into account the users selections for mirroring, tiling, noise, etc. from the UVGen rollups.

**Parameters:**

**ShadeContext &sc**

Describes the properties of the point to evaluate.

**MapSampler\* somp**

This is a callback used to sample the map. The plug-in creates an instance of this class, and implements its methods. A pointer to the instance is passed here. The methods of the class sample the texture for a single UV coordinate.

**BOOL filter=TRUE**

If TRUE the texture will be filtered.

**Return Value:**

The result as a float.

The following two methods are called for bump mapping to retrieve a normal perturbation. The calling sequence for these methods is shown in the sample code below. This code takes the sum of the U derivative times the U bump vector, and the V derivative times the V bump vector. The result is the normal perturbation returned from **EvalNormalPerturb()**.

```
Point3 BMTex::EvalNormalPerturb(ShadeContext& sc) {  
  Point3 dPdu, dPdv;  
  if (thebm==NULL)  
    return Point3(0,0,0);  
  uvGen->GetBumpDP(sc,dPdu,dPdv); // get bump basis vectors  
  Point2 dM =(.01f)*uvGen-  
>EvalDeriv(sc,&mysomp,filterType!=FILTER_NADA);
```

```

return texout->Filter(dM.x*dPdu+dM.y*dPdv);
}

```

Note: In the following code you'll see the 'text book' Blinn's algorithm and the 'Lazy' approach that is actually used. During testing it was determined that these two methods are visually identical. The 'Lazy' algorithm avoids the cross products however.

```

Point3 Gradient::EvalNormalPerturb(ShadeContext& sc) {
    Point3 dPdu, dPdv;
    Point2 dM = uvGen->EvalDeriv(sc,&mysamp);
    uvGen->GetBumpDP(sc,dPdu,dPdv);
    #if 0
    // Blinn's algorithm
    Point3 N = sc.Normal();
    Point3 uVec = CrossProd(N,dPdv);
    Point3 vVec = CrossProd(N,dPdu);
    return dM.x*uVec-dM.y*vVec;
    #else
    // Lazy algorithm
    return texout->Filter(dM.x*dPdu+dM.y*dPdv);
    #endif
}

```

#### Prototype:

```

virtual void GetBumpDP(ShadeContext& sc, Point3& dPdu,
    Point3& dPdv)=0;

```

#### Remarks:

This method may be called to compute **dPdu** and **dPdv** bump basis vectors for bump mapping. These are two vector in 3D space that give you the U and V axis of the map space. This is like taking the map space into 3D space so you can determine where the U axis is and where the V axis is.

Note: In 3ds max 3.0 and later these vectors are always normalized. This change makes it so bump mapping is invariant when a Rescale World Units is performed.

See the Advanced Topics section on [Working with Materials and Textures](#) for

more information on bump mapping.

**Parameters:**

**ShadeContext& sc**

Describes the properties of the point to evaluate.

**Point3& dPdu, Point3& dPdv**

The bump basis vectors are returned here.

**Prototype:**

**virtual Point2 EvalDeriv(ShadeContext& sc, MapSampler\* samp, BOOL filter=TRUE)=0;**

**Remarks:**

This gets the amount that U and V are changing. It takes into account the users selections for mirroring, tiling, noise, etc. from the UVGen rollups.

**Parameters:**

**ShadeContext& sc**

Describes the properties of the point to evaluate.

**MapSampler\* samp**

This is a callback used to sample the map. The plug-in creates an instance of this class, and implements its methods. A pointer to the instance is passed here. The methods of the class sample the texture for a single UV coordinate.

**BOOL filter=TRUE**

If TRUE the texture will be filtered.

**Return Value:**

**df/du, df/dv** as a Point2.

**Prototype:**

**virtual void GetUV(ShadeContext& sc, Point2& UV, Point2& dUV)=0;**

**Remarks:**

This method is not normally called directly. It retrieves the UV coordinates (depending on if they are explicit, shrinked-wrapped environment, etc.) and transforms them using the offsets and noise parameters entered by the user. It does not do the mirroring or tiling. It then returns the texture coordinates and

derivatives for antialiasing.

**Parameters:**

**ShadeContext& sc**

Describes the properties of the point to evaluate.

**Point2& UV**

The UV texture coordinates for the point.

**Point2& dUV**

The derivatives of UV for the point to shade. This describes how big of a range in UV space the current pixel covers.

**Prototype:**

**virtual void GetUVTransform(Matrix3 &uvtrans)=0;**

**Remarks:**

This method is used internally to provide information to the interactive renderer.

**Prototype:**

**virtual int GetTextureTiling()=0;**

**Remarks:**

This method is used internally to provide information to the interactive renderer.

**Prototype:**

**virtual int GetUVWSource()=0;**

**Remarks:**

This method returns the source for the texture vertices.

**Return Value:**

One of the following values:

**UVWSRC\_EXPLICIT**

Use explicit mesh texture vertices from channel 1.

**UVWSRC\_EXPLICIT2**

Use explicit mesh texture vertices from channel 2.

## **UVWSRC\_OBJXYZ**

Generate planar UVW mapping coordinates from the object XYZ on-the-fly.

### **Prototype:**

```
virtual void SetUVWSource(int s)=0;
```

### **Remarks:**

This method sets the source for the texture vertices.

### **Parameters:**

**int s**

One of the following values:

#### **UVWSRC\_EXPLICIT**

Use explicit mesh texture vertices from one of the mapping channels (as established by **SetMapChannel()** below).

#### **UVWSRC\_EXPLICIT2**

Use explicit mesh texture vertices from the Vertex Color Channel.

#### **UVWSRC\_OBJXYZ**

Generate planar UVW mapping coordinates from the object XYZ on-the-fly.

#### **UVWSRC\_WORLDXYZ**

This parameter is available in release 3.0 and later only.

Generate planar uvw mapping from world xyz on-the-fly.

### **Prototype:**

```
virtual void SetMapChannel(int s);
```

### **Remarks:**

This method is available in release 3.0 and later only.

Sets the mapping channel used when **UVWSRC\_EXPLICIT** is in effect.

### **Parameters:**

**int s**

The mapping channel value to set.

### **Default Implementation:**

```
{}
```

**Prototype:**

**virtual int SymFlags()=0;**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual void SetSymFlags(int f)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method is used internally.

**Prototype:**

**virtual void SetClipFlag(BOOL b)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method sets the clip flag. The clip flag controls whether the U,V values passed to **MapSampler** by **EvalUVMap()** and **EvalUVMapMono()** are clipped to the [0..1] interval or not. It defaults to ON (i.e., clipped).

If an object is covered by a texture, and that texture repeats (is tiled), this methods can be used to tell you which tile you're in. This allows one to do something depending on which tile it is. This is done, for example, in the Bitmap texture, when 'Jitter Placement' is on, so each tile can be moved about slightly. For an example of this see

**\MAXSDK\SAMPLES\MATERIALS\BMTEX.CPP**. If you need this information, you can just call **uvGen->SetClipFlag(FALSE)** (for instance in your **Update()** method) and then use **int(u)** and **int(v)** to get the this info. (You'd use **frac(u)** and **frac(v)** to get clipped versions of u,v. )

**Parameters:**

**BOOL b**

TRUE to clip; FALSE to not clip.

**Prototype:**

**virtual BOOL GetClipFlag()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the clip flag. The clip flag controls whether the U,V values passed to **MapSampler** by **EvalUVMMap()** and **EvalUVMMapMono()** are clipped to the [0..1] interval or not. It defaults to ON (i.e., clipped). See **SetClipFlag()** above.

**Return Value:**

TRUE for clipped; FALSE if not clipped.

**Prototype:**

```
virtual int GetAxis()=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Return the axis (mapping coordinate system) used.

**Return Value:**

One of the following values:

**AXIS\_UV**

**AXIS\_VW**

**AXIS\_WU**

**Prototype:**

```
virtual void SetAxis(int ax)=0;
```

**Remarks:**

This method is available in release 2.0 and later only.

Sets the axis (mapping coordinate system) used.

**Parameters:**

**int ax**

One of the following values:

**AXIS\_UV**

**AXIS\_VW**

**AXIS\_WU**

**Prototype:**

```
virtual void InitSlotType(int sType)=0;
```

**Remarks:**

Developers typically don't need to call this method. In the Coordinates rollup in the user interface for a texture map are two options. These options are Texture or Environment. The slot type is one of these two options. There are a variety of texture coordinate types. There are the type assigned to the object and the environment type (Spherical, Cylindrical, Shrink-wrap, Screen). This method is used initialize the radio button in the dialog based on the **sType** passed and update the drop down list.

**Parameters:**

**int sType**

One of the following values:

**MAPSLOT\_TEXTURE**

This is either texture coordinates.

**MAPSLOT\_ENVIRON**

Environment coordinates.

**Prototype:**

```
virtual void SetRollupOpen(BOOL open)=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the UVGen rollup state to open or closed.

**Parameters:**

**BOOL open**

TRUE for open; FALSE for closed.

**Prototype:**

```
virtual BOOL GetRollupOpen()=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the open or closed state of the UVGen rollup.

**Return Value:**

TRUE is open; FALSE is closed.

# Class MapSampler

See Also: [Class UVGen](#), [Class ShadeContext](#).

## Description:

A texture map implements this class and passes it into the **UVGen** methods **EvalUVMMap()**, **EvalUVMMapMono()**, and **EvalDeriv()** to evaluate itself with tiling & mirroring. Each of the methods of this class are used to sample the map at a single UV coordinate.

## Methods:

### Prototype:

```
virtual AColor Sample(ShadeContext& sc, float u, float v)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is required. It is called to sample the map at a single UV coordinate. This method should not try to antialias the map, it should just return the value. Take for example a bitmap texture implementing this method. If the **u** values passed was 0.5 and the **v** value passed was 0.5, the plug-in would just return the pixel value at the center of the bitmap as an **AColor**.

### Parameters:

**ShadeContext& sc**

The ShadeContext.

**float u**

The U value in the range of 0.0 to 1.0

**float v**

The V value in the range of 0.0 to 1.0

### Return Value:

The sampled value as an **AColor**.

### Prototype:

```
virtual AColor SampleFilter(ShadeContext& sc, float u,  
float v, float du, float dv)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is required. It is called to return a value from the map that is the averaged value over the specified region of width **du** and height **dv**, centered at **u**, **v**. Certain map types may use the **du** and **dv** values directly to perform antialiasing of the map.

**Parameters:**

**ShadeContext& sc**

The ShadeContext.

**float u**

The U value in the range of 0.0 to 1.0

**float v**

The V value in the range of 0.0 to 1.0

**float du**

This parameter can be considered the width of a small rectangle center on u,v. The u and v value are guaranteed to be inside this rectangle.

**float dv**

This parameter can be considered the height of a small rectangle center on u,v. The u and v value are guaranteed to be inside this rectangle.

**Return Value:**

The sampled value as an **AColor**.

**Prototype:**

**virtual float SampleMono(ShadeContext& sc, float u, float v)**

**Remarks:**

Implemented by the Plug-In.

This method is optional as a default implementation is provided. It may be implemented however if a certain map type can be more efficient in evaluating a mono channel. For example a noise function that produces a single floating point value as a function of UV. This method is called to sample the map at a single UV coordinate. This method should not try to antialias the map, it should just return the value.

**Parameters:**

**ShadeContext& sc**

The ShadeContext.

**float u**

The U value in the range of 0.0 to 1.0

**float v**

The V value in the range of 0.0 to 1.0

**Return Value:**

The sampled value as an float.

**Default Implementation:**

```
{ return Intens(Sample(sc,u,v)); }
```

**Prototype:**

```
virtual float SampleMonoFilter(ShadeContext& sc, float u,  
float v, float du, float dv)
```

**Remarks:**

Implemented by the Plug-In.

This method is optional. It is called to return a value from the map that is the averaged value over the specified region of width **du** and height **dv**, centered at **u, v**. Certain map types may use the **du** and **dv** values directly to perform antialiasing of the map.

**Parameters:****ShadeContext& sc**

The ShadeContext.

**float u**

The U value in the range of 0.0 to 1.0

**float v**

The V value in the range of 0.0 to 1.0

**float du**

This parameter can be considered the width of a small rectangle center on **u,v**. The **u** and **v** value are guaranteed to be inside this rectangle.

**float dv**

This parameter can be considered the height of a small rectangle center on **u,v**. The **u** and **v** value are guaranteed to be inside this rectangle.

**Return Value:**

The sampled value as an float.

**Default Implementation:**

```
{ return Intens(SampleFilter(sc,u,v,du,dv)); }
```

## Class XYZGen

See Also: [Class MtlBase](#), [Class ShadeContext](#), [Class Point3](#).

```
class XYZGen : public MtlBase
```

### Description:

This class generates **Point3** coordinates based on the **ShadeContext**. A reference to one of these is referenced by all 3D texture maps. **XYZGen** does for 3D Texmaps what **UVGen** does for 2D Texmaps. It puts up the 3D "Coordinates" rollup, and supplies the 3D Texmap with transformed 3D coordinates. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
virtual void GetXYZ(ShadeContext& sc, Point3& p, Point3& dp)=0;
```

#### Remarks:

This method retrieves coordinates and derivatives for antialiasing. It returns the coordinates in the "Object Coordinate System", scaled and rotated by the parameters given in the Coordinates rollup. The default transformation is the identity transform, so you just get Object coordinates. For instance, if you have a sphere of radius 100, you will get coordinates ranging from -100 to +100.

A developer might wonder why 3ds max don't use some normalized coordinates space. This was considered, but it has some real problems. Say you make something by assembling several boxes, of varying sizes, and then apply a wood texture to the whole thing. You want the wood texture to be at the same scale for all the pieces, not smaller on the smaller pieces, and thus that doesn't work.

#### Parameters:

**ShadeContext& sc**

Describes the properties of the point to be shaded.

**Point3& p**

The 3D point is returned here.

**Point3& dp**

The derivative of the 3D point is returned here.

**Prototype:**

**virtual BOOL IsStdUVGen();**

**Remarks:**

This method is available in release 2.0 and later only.

This method asks the question 'Is this class an instance of **StdUVGen**?'. The derived class **StdUVGen** implements this to return TRUE. Others use the default implementation to return FALSE.

**Default Implementation:**

**{ return FALSE; }**

**Prototype:**

**virtual void SetRollupOpen(BOOL open)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the XYZGen rollup state to open or closed.

**Parameters:**

**BOOL open**

TRUE for open; FALSE for closed.

**Prototype:**

**virtual BOOL GetRollupOpen()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the open or closed state of the XYZGen rollup.

**Return Value:**

TRUE is open; FALSE is closed.

**Prototype:**

**virtual void GetBumpDP(ShadeContext& sc, Point3\* dP);**

**Remarks:**

This method is available in release 3.0 and later only.

This method returns the transformed bump unit vectors for 3D bump mapping. The 3D textures need to use these in computing the gradient.

**Parameters:**

**ShadeContext& sc**

Describes the properties of the point to be shaded.

**Point3\* dP**

The 3 unit vectors for computing differentials are returned here.

**Default Implementation:**

```
{
```

**Sample Code:**

Here is a typical use of XYZGen->GetBumpDP() to compute the bump mapping for a 3D texture:

```
Point3 Marble::EvalNormalPerturb(ShadeContext& sc) {
    float del,d;
    Point3 p,dp;
    if (!sc.doMaps) return Point3(0,0,0);
    if (gbufID) sc.SetGBufferID(gbufID);
    xyzGen->GetXYZ(sc,p,dp);
    if (size==0.0f) size=.0001f;
    p *= FACT/size;
    d = MarbleFunc(p);
    del = 20.0f;
    Point3 np;
    Point3 M[3];
    xyzGen->GetBumpDP(sc,M);
    np.x = (MarbleFunc(p+del*M[0]) - d)/del;
    np.y = (MarbleFunc(p+del*M[1]) - d)/del;
    np.z = (MarbleFunc(p+del*M[2]) - d)/del;
    np *= 100.0f;
    return sc.VectorFrom(np,REF_OBJECT);
}
```

## Class TextureOutput

See Also: [Class MtlBase](#), [Class AColor](#), [Class Point3](#).

```
class TextureOutput : public MtlBase
```

### Description:

This class is used by texture maps to put up the 'Output' rollup in the materials editor, and perform the output filtering. Currently this provides control over the Output Amount, RGB Level, and RGB Offset. In the future this may be enhanced to include other things that are often desirable on the output stage. These are things like tinting, color shifting, etc. All methods of this class are implemented by the system.

A plug-in will typically call these methods from the implementations of **EvalColor()**, **EvalMono()**, etc. This is just done as a final adjustment to scale or offset the value. The code below is from the Gradient texture map.

```
AColor Gradient::EvalColor(ShadeContext& sc) {  
    return texout->Filter(uvGen->EvalUVMMap(sc,&mysamp));  
}  
float Gradient::EvalMono(ShadeContext& sc) {  
    return texout->Filter(uvGen-  
>EvalUVMMapMono(sc,&mysamp));  
}
```

### Methods:

#### Prototype:

```
virtual AColor Filter(AColor c) = 0;
```

#### Remarks:

Filters the specified AColor and returns it.

#### Parameters:

**AColor c**

The color to filter.

#### Return Value:

The filtered color as an AColor.

**Prototype:**

**virtual float Filter(float f) = 0;**

**Remarks:**

Filters the specified float value and returns it.

**Parameters:**

**float f**

The value to filter.

**Return Value:**

The filtered value.

**Prototype:**

**virtual Point3 Filter(Point3 p) = 0;**

**Remarks:**

Filters the specified Point3 value and returns it.

**Parameters:**

**Point3 p**

The Point3 to filter.

**Return Value:**

The filtered Point3.

**Prototype:**

**virtual float GetOutputLevel(TimeValue t) = 0;**

**Remarks:**

Returns the output level (amount) at the specified time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the output level.

**Prototype:**

**virtual void SetOutputLevel(TimeValue t, float v) = 0;**

**Remarks:**

Sets the output level at the specified time.

**Parameters:**

**TimeValue t**

The time to set the output level.

**float v**

The value of the output level.

**Prototype:**

**virtual void SetInvert(BOOL onoff)=0;**

**Remarks:**

Sets the state of the 'Invert' toggle in the Output rollup.

**Parameters:**

**BOOL onoff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual BOOL GetInvert()=0;**

**Remarks:**

Returns the state of the 'Invert' toggle in the Output rollup.

**Return Value:**

TRUE is on; FALSE is off.

**Prototype:**

**virtual void SetRollupOpen(BOOL open)=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the 'Output' rollup page to open or closed.

**Parameters:**

**BOOL open**

TRUE to open the rollup; FALSE to close it.

**Prototype:**

**virtual BOOL GetRollupOpen()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the open or closed state of the 'Output' rollup.

**Return Value:**

TRUE for open; FALSE for closed.

## Class MultiMtl

See Also: [Class Mtl](#).

```
class MultiMtl : public Mtl
```

### Description:

This class provides access to the developer alterable properties of the 3ds max Multi/Sub-Object material.

### Methods:

#### Prototype:

```
virtual void SetNumSubMtls(int n)=0;
```

#### Remarks:

Sets the number of sub-materials for the multi-material.

#### Parameters:

**int n**

The number of sub-materials.

#### Prototype:

```
virtual void GetSubMtlName(int mtlid, TSTR &s)=0;
```

#### Remarks:

This method is available in release 4.0 and later only.

Retrieves the name of the sub-material whose ID is passed.

#### Parameters:

**int mtlid**

The zero based index of the sub-material.

**TSTR &s**

The name is returned here.

#### Prototype:

```
virtual void SetSubMtlAndName(int mtlid, Mtl *m, TSTR  
&subMtlName)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Retrieves the name and pointer to the material for the specified sub-material.

**Parameters:****int mtlid**

The zero based index of the sub-material.

**Mtl \*m**

Points to the sub-material.

**TSTR &subMtlName**

The name is returned here.

# Class StdUVGen

See Also: [Class UVGen](#).

```
class StdUVGen : public UVGen
```

## Description:

This class provides access to the parameters of the 3ds max UVGen class. These are the settings in the 'Coordinates' and 'Noise' rollups such as UV offsets, angle, blur, noise level, etc. All methods of this class are implemented by the system.

## Methods Groups:

The following hyperlinks take you to the beginning of groups of related method within the class.

[Setting properties](#)

[Retrieving properties](#)

## Methods:

### Setting properties

#### Prototype:

```
virtual void SetCoordMapping(int)=0;
```

#### Remarks:

This method is available in release 2.0 and later only.

Sets the mapping type to one of the specified values.

#### Parameters:

**int**

The mapping type. One of the following values:

**UVMAP\_EXPLICIT**

Explicit Texture mapping.

**UVMAP\_SPHERE\_ENV**

Spherical Environment mapping.

**UVMAP\_CYL\_ENV**

Cylindrical Environment mapping.

**UVMAP\_SHRINK\_ENV**

Shrink Wrap Environment mapping.

## **UVMAP\_SCREEN\_ENV**

Screen Environment mapping.

### **Prototype:**

**virtual void SetUOffs(float f, TimeValue t)=0;**

### **Remarks:**

Sets the U Offset setting to the specified value at the time passed.

### **Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

### **Prototype:**

**virtual void SetVOffs(float f, TimeValue t)=0;**

### **Remarks:**

Sets the V Offset setting to the specified value at the time passed.

### **Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

### **Prototype:**

**virtual void SetUScl(float f, TimeValue t)=0;**

### **Remarks:**

Sets the U tiling setting to the specified value at the time passed.

### **Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetVScl(float f, TimeValue t)=0;**

**Remarks:**

Sets the V tiling setting to the specified value at the time passed.

**Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetAng(float f, TimeValue t)=0;**

**Remarks:**

Sets the angle setting to the specified value at the time passed.

**Parameters:**

**float f**

The value to set in radians.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetUAng(float f, TimeValue t)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the U Angle setting the specified value at the time passed.

**Parameters:**

**float f**

The angle to set in radians.

**TimeValue t**

The time at which to set the angle.

**Prototype:**

**virtual void SetVAng(float f, TimeValue t)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the V Angle setting the specified value at the time passed.

**Parameters:**

**float f**

The angle to set in radians.

**TimeValue t**

The time at which to set the angle.

**Prototype:**

**virtual void SetWAng(float f, TimeValue t)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the W Angle setting the specified value at the time passed.

**Parameters:**

**float f**

The angle to set in radians.

**TimeValue t**

The time at which to set the angle.

**Prototype:**

**virtual void SetBlur(float f, TimeValue t)=0;**

**Remarks:**

Sets the blur to the specified value at the time passed.

**Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetBlurOffs(float f, TimeValue t)=0;**

**Remarks:**

Sets the blur offset to the specified value at the time passed.

**Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetNoiseAmt(float f, TimeValue t)=0;**

**Remarks:**

Sets the noise amount to the specified value at the time passed.

**Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetNoiseSize(float f, TimeValue t)=0;**

**Remarks:**

Sets the noise size to the specified value at the time passed.

**Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetNoiseLev(int i, TimeValue t)=0;**

**Remarks:**

Sets the noise level to the specified value at the time passed.

**Parameters:****int i**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:****virtual void SetNoisePhs(float f, TimeValue t)=0;****Remarks:**

Sets the noise phase to the specified value at the time passed.

**Parameters:****float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:****virtual void SetTextureTiling(int tiling)=0;****Remarks:**

Sets the texture tiling setting.

**Parameters:****int tiling**

See [List of Texture Symmetry Flags](#).

**Prototype:****virtual void SetMapChannel(int i)=0;****Remarks:**

This method is available in release 3.0 and later only.

Sets the mapping channel to the specified value.

**Parameters:****int i**

The channel to set.

**Prototype:****virtual void SetFlag(ULONG f, ULONG val)=0;****Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the UVGen flags.

**Parameters:****ULONG f**See [List of Texture Symmetry Flags](#).**ULONG val**

The value to set.

**Prototype:****virtual void SetHideMapBackFlag(BOOL b)=0;****Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the hide map back flag.

**Parameters:****BOOL b**

TRUE to set the flag; FALSE to disable.

**Retrieving properties****Prototype:****virtual int GetCoordMapping(int)=0;****Remarks:**

Retrieves the coordinate mapping type.

**Parameters:****int**

This parameter is not used.

**Return Value:**

One of the following values:

**UVMAP\_EXPLICIT**  
**UVMAP\_SPHERE\_ENV**  
**UVMAP\_CYL\_ENV**  
**UVMAP\_SHRINK\_ENV**  
**UVMAP\_SCREEN\_ENV**

**Prototype:**

**virtual float GetUOffs(TimeValue t)=0;**

**Remarks:**

Retrieves the U Offset setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetVOffs(TimeValue t)=0;**

**Remarks:**

Retrieves the V Offset setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetUScl(TimeValue t)=0;**

**Remarks:**

Retrieves the U Tiling setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetVScl(TimeValue t)=0;**

**Remarks:**

Retrieves the V Tiling setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetAng(TimeValue t)=0;**

**Remarks:**

Retrieves the angle setting in radians.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetUAng(TimeValue t)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the U Angle setting (in radians) at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the angle.

**Prototype:**

**virtual float GetVAng(TimeValue t)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the V Angle setting (in radians) at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the angle.

**Prototype:**

**virtual float GetWAng(TimeValue t)=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the W Angle setting (in radians) at the specified time.

**Parameters:****TimeValue t**

The time at which to return the angle.

**Prototype:**

**virtual float GetBlur(TimeValue t)=0;**

**Remarks:**

Retrieves the blur setting at the specified time.

**Parameters:****TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetBlurOffs(TimeValue t)=0;**

**Remarks:**

Retrieves the blur offset setting at the specified time.

**Parameters:****TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetNoiseAmt(TimeValue t)=0;**

**Remarks:**

Retrieves the noise amount setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetNoiseSize(TimeValue t)=0;**

**Remarks:**

Retrieves the noise size setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual int GetNoiseLev(TimeValue t)=0;**

**Remarks:**

Retrieves the noise level setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual float GetNoisePhs(TimeValue t)=0;**

**Remarks:**

Retrieves the noise phase setting at the specified time.

**Parameters:**

**TimeValue t**

The time to retrieve the value.

**Prototype:**

**virtual int GetTextureTiling()=0;**

**Remarks:**

Returns the texture tiling flags.

**Return Value:**

See [List of Texture Symmetry Flags](#).

**Prototype:**

```
virtual int GetMapChannel()=0;
```

**Remarks:**

This method is available in release 3.0 and later only.

Returns the mapping channel.

**Prototype:**

```
virtual int GetFlag(ULONG f)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the flag status of the UVGen.

**Parameters:**

**ULONG f**

See [List of Texture Symmetry Flags](#).

**Prototype:**

```
virtual BOOL GetHideMapBackFlag()=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method returns the state of the hide map back flag.

## Class StdXYZGen

See Also: [Class XYZGen](#).

```
class StdXYZGen: public XYZGen
```

### Description:

This class is available in release 2.0 and later only.

This class provides access to the parameters of the 3ds max XYZGen class. The XYZGen class puts the 3D 'Coordinates' rollup.

### Methods:

#### Prototype:

```
BOOL IsStdXYZGen();
```

#### Remarks:

This is an implementation of a method of **XYZGen**. This returns TRUE to indicate this is a **StdXYZGen** instance.

#### Prototype:

```
virtual void SetCoordSystem(int s)=0;
```

#### Remarks:

Sets the coordinate system used.

#### Parameters:

**int s**

The coordinate system to set. One of the following values:

**XYZ\_COORDS** - Object XYZ

**UVW\_COORDS** - Explicit Map Channel

**UVW2\_COORDS** - Vertex Color Channel

**XYZ\_WORLD\_COORDS** - World XYZ. This option is available in release 3.0 and later only.

#### Prototype:

```
virtual void SetBlur(float f, TimeValue t)=0;
```

#### Remarks:

Sets the Blur setting to the specified value at the specified time.

**Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetBlurOffs(float f, TimeValue t)=0;**

**Remarks:**

Sets the Blur Offset setting to the specified value at the specified time.

**Parameters:**

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetOffs(int axis, float f, TimeValue t)=0;**

**Remarks:**

Sets the specified Offset setting to the specified value at the specified time.

**Parameters:**

**int axis**

The axis to set. **0** for X, **1** for Y, **2** for Z.

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:**

**virtual void SetScl(int axis, float f, TimeValue t)=0;**

**Remarks:**

Sets the specified Tiling (Scale) setting to the specified value at the specified time.

**Parameters:****int axis**

The axis to set. **0** for X, **1** for Y, **2** for Z.

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:****virtual void SetAng(int axis, float f, TimeValue t)=0;****Remarks:**

Sets the specified Angle setting to the specified value at the specified time.

**Parameters:****int axis**

The axis to set. **0** for X, **1** for Y, **2** for Z.

**float f**

The value to set.

**TimeValue t**

The time at which to set the value.

**Prototype:****virtual void SetMapChannel(int i)=0;****Remarks:**

This method is available in release 3.0 and later only.

Sets the mapping channel to the specified value.

**Parameters:****int i**

The mapping channel to set.

**Prototype:**

**virtual int GetCoordSystem()=0;**

**Remarks:**

Returns the coordinate system in use. One of the following values:

**XYZ\_COORDS** - Object XYZ

**UVW\_COORDS** - Explicit Map Channel

**UVW2\_COORDS** - Vertex Color Channel

**XYZ\_WORLD\_COORDS** - World XYZ. This option is available in release 3.0 and later only.

**Prototype:**

**virtual float GetBlur(TimeValue t)=0;**

**Remarks:**

Returns the Blur setting at the specified time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the value.

**Prototype:**

**virtual float GetBlurOffs(TimeValue t)=0;**

**Remarks:**

Returns the Blur Offset setting at the specified time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the value.

**Prototype:**

**virtual float GetScl(int axis, TimeValue t)=0;**

**Remarks:**

Returns the Tiling (Scale) setting for the specified axis at the specified time.

**Parameters:**

**int axis**

The axis to set. **0** for X, **1** for Y, **2** for Z.

**TimeValue t**

The time at which to retrieve the value.

**Prototype:**

**virtual float GetAng(int axis, TimeValue t)=0;**

**Remarks:**

Returns the Angle setting for the specified axis at the specified time.

**Parameters:**

**int axis**

The axis to set. **0** for X, **1** for Y, **2** for Z.

**TimeValue t**

The time at which to retrieve the value.

**Prototype:**

**virtual int GetMapChannel()=0;**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the mapping channel.

## Class MtlBaseLib

See Also: [Class ReferenceTarget](#), [Class MtlBaseList](#), [Class MtlBase](#), [Class Interface](#).

```
class MtlBaseLib : public ReferenceTarget, public MtlBaseList
```

### Description:

This class is a library of **MtlBase** entries. To get a list of the currently loaded materials use:

### **MtlBaseLib& Interface::GetMaterialLibrary()**.

When you load a material library, you can enumerate all the materials that it contains. **MtlBaseLib** is subclassed off of **MtlBaseList** which is a **Tab<MtlHandle>**, so for example, you can do the following to work with each material:

```
for (int i=0; i<mlib.Count(); i++) { DoSomething(mlib[i]); }
```

For additional information on this class see the Advanced Topics section in [Working with Materials and Textures](#). All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
virtual void Remove(MtlBase *m);
```

#### Remarks:

Removes the specified MtlBase from the library.

#### Parameters:

**MtlBase \*m**

The MtlBase to remove.

#### Prototype:

```
virtual void Add(MtlBase *m);
```

#### Remarks:

Adds the specified MtlBase to the library.

#### Parameters:

**MtlBase \*m**

The MtlBase to add.

**Prototype:**

**virtual void RemoveDuplicates();**

**Remarks:**

Removes all duplicate MtlBases from the library.

**Prototype:**

**void DeleteAll();**

**Remarks:**

Removes all MtlBases from the library and deletes all references.

**Prototype:**

**~MtlBaseLib();**

**Remarks:**

Destructor. All references are deleted.

## List of Texture Map Indices

See Also: [Class MtlBase](#), [Class Shader](#).

The Standard Material texture map index used by Blinn, Phong, Constant, Metal (defined in `\MAXSDK\INCLUDE\STDMAT.H`). One of the following values:

**ID\_AM** - Ambient (value 0)

**ID\_DI** - Diffuse (value 1)

**ID\_SP** - Specular (value 2)

**ID\_SH** - Shininess (value 3). In R3 and later this is called Glossiness.

**ID\_SS** - Shininess strength (value 4). In R3 and later this is called Specular Level.

**ID\_SI** - Self-illumination (value 5)

**ID\_OP** - Opacity (value 6)

**ID\_FI** - Filter color (value 7)

**ID\_BU** - Bump (value 8)

**ID\_RL** - Reflection (value 9)

**ID\_RR** - Refraction (value 10)

**ID\_DP** - Displacement (value 11)

## Class MultiTex

See Also: [Class Texmap](#), [Class Color](#), [Working with Materials and Textures](#).

```
class MultiTex : public Texmap
```

### Description:

This class provides access to the developer settable properties of the 3ds max multi-textures such as Composite, Tint and Mix. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
virtual void SetNumSubTexmaps(int n);
```

#### Remarks:

Sets the number of sub-textmaps for this texmap.

#### Parameters:

**int n**

The number of sub-textmaps.

#### Prototype:

```
virtual void SetColor(int i, Color c, TimeValue t=0);
```

#### Remarks:

Sets the color of the 'i-th' sub-textmap to the specified color at the time passed.

#### Parameters:

**int i**

The index of the sub-textmap to set.

**Color c**

The color to set.

**TimeValue t=0**

The time at which to set the color.

# Class GradTex

See Also: [Class MultiTex](#), [Class TextureOutput](#).

class GradTex: public MultiTex

## Description:

This class provides access to the 3ds max Gradient texture. All methods of this class are implemented by the system.

## Methods:

### Prototype:

**virtual StdUVGen\* GetUVGen()=0;**

### Remarks:

Retrieves a pointer to the **StdUVGen** interface for this texture. This allows access to the mapping parameters such as UV offsets, blur, angle, noise level, etc.

### Prototype:

**virtual TextureOutput\* GetTexout()=0;**

### Remarks:

Retrieves a pointer to the **TextureOutput** for this texture. This allows access to the RGB level and output and the output amount.

### Prototype:

**virtual void SetMidPoint(float m, TimeValue t=0);**

### Remarks:

Sets the 'Color 2 Position' setting at the time passed.

### Parameters:

**float m**

The value to set in the range of 0.0 to 1.0.

**TimeValue t=0**

The time to set the value.

## Class StdFog

See Also: [Class Atmospheric](#), [Class Texmap](#).

```
class StdFog : public Atmospheric
```

### Description:

This class provides access to the settings of the Standard Fog Atmospheric plugin of 3ds max. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
virtual void SetColor(Color c, TimeValue t)=0;
```

#### Remarks:

Sets the fog color.

#### Parameters:

**Color c**

The color to set.

**TimeValue t**

The time to set the color.

#### Prototype:

```
virtual void SetUseMap(BOOL onoff)=0;
```

#### Remarks:

Sets the state of the 'Use Map' toggle.

#### Parameters:

**BOOL onoff**

TRUE for on; FALSE for off.

#### Prototype:

```
virtual void SetUseOpac(BOOL onoff)=0;
```

#### Remarks:

Sets the state of the use opacity map toggle.

#### Parameters:

**BOOL onoff**

TRUE for on; FALSE for off.

**Prototype:**

```
virtual void SetColorMap(Texture *tex)=0;
```

**Remarks:**

Set the color map used.

**Parameters:**

**Texture \*tex**

The map to set.

**Prototype:**

```
virtual void SetOpacMap(Texture *tex)=0;
```

**Remarks:**

Set the opacity map used.

**Parameters:**

**Texture \*tex**

The map to set.

**Prototype:**

```
virtual void SetFogBackground(BOOL onoff)=0;
```

**Remarks:**

Sets the state of the fog background toggle.

**Parameters:**

**BOOL onoff**

TRUE for on; FALSE for off.

**Prototype:**

```
virtual void SetType(int type)=0;
```

**Remarks:**

Set the type of fog, layered or standard.

**Parameters:**

**int type**

The type of fog: 0 = Standard; 1 = Layered.

**Prototype:**

**virtual void SetNear(float v, TimeValue t)=0;**

**Remarks:**

Sets the standard fog near percentage.

**Parameters:**

**float v**

The value to set in the range 0 to 1.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetFar(float v, TimeValue t)=0;**

**Remarks:**

Sets the standard fog far percentage.

**Parameters:**

**float v**

The value to set in the range 0 to 1.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetTop(float v, TimeValue t)=0;**

**Remarks:**

Sets the layered fog top value.

**Parameters:**

**float v**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetBottom(float v, TimeValue t)=0;**

**Remarks:**

Sets the layered fog bottom value.

**Parameters:**

**float v**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetDensity(float v, TimeValue t)=0;**

**Remarks:**

Sets the layered fog density setting.

**Parameters:**

**float v**

The value to set (> 0).

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetFalloffType(int tv)=0;**

**Remarks:**

Sets the falloff type to top, bottom or none.

**Parameters:**

**int tv**

One of the following values:

**FALLOFF\_TOP**

**FALLOFF\_BOTTOM**

**FALLOFF\_NONE**

**Prototype:**

**virtual void SetUseNoise(BOOL onoff)=0;**

**Remarks:**

Sets the state of the 'Horizon Noise' toggle.

**Parameters:**

**BOOL onoff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetNoiseScale(float v, TimeValue t)=0;**

**Remarks:**

Set the 'Horizon Noise Size' setting.

**Parameters:**

**float v**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetNoiseAngle(float v, TimeValue t)=0;**

**Remarks:**

Set the 'Horizon Noise Angle' setting.

**Parameters:**

**float v**

The value to set in radians.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual void SetNoisePhase(float v, TimeValue t)=0;**

**Remarks:**

Set the 'Horizon Noise Phase' setting.

**Parameters:**

**float v**

The value to set.

**TimeValue t**

The time to set the value.

**Prototype:**

**virtual Color GetColor(TimeValue t)=0;**

**Remarks:**

Returns the fog color at the time passed.

**Parameters:**

**TimeValue t**

The time to get the color.

**Prototype:**

**virtual BOOL GetUseMap()=0;**

**Remarks:**

Returns the state of the 'Use Map' toggle.

**Return Value:**

TRUE is on; FALSE is off.

**Prototype:**

**virtual BOOL GetUseOpac()=0;**

**Remarks:**

Returns the state of the use opacity map toggle.

**Return Value:**

TRUE is on; FALSE is off.

**Prototype:**

**virtual Texmap \*GetColorMap()=0;**

**Remarks:**

Returns the color map used.

**Prototype:**

**virtual Texmap \*GetOpacMap()=0;**

**Remarks:**

Returns the opacity map used.

**Prototype:**

**virtual BOOL GetFogBackground()=0;**

**Remarks:**

Returns the state of the fog background toggle.

**Return Value:**

TRUE is on; FALSE is off.

**Prototype:**

**virtual int GetType()=0;**

**Remarks:**

Returns the type of fog, layered or standard.

**Return Value:**

The type of fog: 0 = Standard; 1 = Layered.

**Prototype:**

**virtual float GetNear(TimeValue t)=0;**

**Remarks:**

Returns the standard fog near percentage.

**Parameters:**

**TimeValue t**

The time to get the value.

**Prototype:**

**virtual float GetFar(TimeValue t)=0;**

**Remarks:**

Returns the standard fog far percentage.

**Parameters:**

**TimeValue t**

The time to get the value.

**Prototype:**

**virtual float GetTop(TimeValue t)=0;**

**Remarks:**

Returns the layered fog top value.

**Parameters:**

**TimeValue t**

The time to get the value.

**Prototype:**

**virtual float GetBottom(TimeValue t)=0;**

**Remarks:**

Returns the layered fog bottom value.

**Parameters:**

**TimeValue t**

The time to get the value.

**Prototype:**

**virtual float GetDensity(TimeValue t)=0;**

**Remarks:**

Returns the layered fog density setting.

**Parameters:**

**TimeValue t**

The time to get the value.

**Prototype:**

**virtual int GetFalloffType()=0;**

**Remarks:**

Returns the falloff type to top, bottom or none.

**Return Value:**

One of the following values:

**FALLOFF\_TOP**

**FALLOFF\_BOTTOM**

**FALLOFF\_NONE**

**Prototype:**

**virtual BOOL GetUseNoise()=0;**

**Remarks:**

Returns the state of the 'Horizon Noise' toggle.

**Return Value:**

TRUE is on; FALSE is off.

**Prototype:**

**virtual float GetNoiseScale(TimeValue t)=0;**

**Remarks:**

Returns the 'Horizon Noise Size' setting.

**Parameters:**

**TimeValue t**

The time to get the value.

**Prototype:**

**virtual float GetNoiseAngle(TimeValue t)=0;**

**Remarks:**

Returns the 'Horizon Noise Angle' setting.

**Parameters:**

**TimeValue t**

The time to get the value.

**Prototype:**

**virtual float GetNoisePhase(TimeValue t)=0;**

**Remarks:**

Returns the 'Horizon Noise Phase' setting.

**Parameters:**

**TimeValue t**

The time to get the value.

## Class AdjEdgeList

See Also: [Class DWORDTab](#), [Template Class Tab](#), [Class MEdge](#), [Class Mesh](#).

class AdjEdgeList

### Description:

This class represents an edge adjacency list for meshes. For any given vertex in a mesh this class has a table of DWORDS. These DWORDS are indices into the edge table (**Tab<MEdge> edge**). The **edges** table stores the edges adjacent to the vertex. So, each vertex has a list of indices into the edge list that give it the list of edges adjacent to the vertex. All methods of the class are implemented by the system.

### Data Members:

public:

**DWORDTab \*list;**

This is an array of DWORDTabs, one per vertex. The Tab is a list of indices into the edge list, one for each edge adjacent to the vertex.

**Tab<MEdge> edges;**

The table of edges.

**int nverts;**

The size of **list**.

### Methods:

### Prototype:

**AdjEdgeList(Mesh& amesh);**

### Remarks:

Constructor. Builds an adjacency list from the specified mesh. This class require the mesh to be constructed so that each edge has exactly one or two faces.

### Parameters:

**Mesh& amesh**

The mesh to build the adjacency list from.

### Prototype:

**~AdjEdgeList();**

**Remarks:**

Destructor. Deletes the list.

**Prototype:**

**int FindEdge(DWORD v0, DWORD v1);**

**Remarks:**

Finds the edge in the edge table that has the two specified vertices.

**Parameters:**

**DWORD v0, DWORD v1**

The vertices.

**Return Value:**

The index into the edge table.

**Prototype:**

**int FindEdge(DWORDTab& vmap, DWORD v0, DWORD v1);**

**Remarks:**

This method is used internally as part of the Optimize modifier.

**Prototype:**

**void TransferEdges(DWORD from, DWORD to, DWORD  
except1,  
DWORD except2, DWORD del);**

**Remarks:**

This method is used internally as part of the Optimize modifier.

**Prototype:**

**void RemoveEdge(DWORD from, DWORD e);**

**Remarks:**

This method is used internally as part of the Optimize modifier.

**Prototype:**

**void OrderVertEdges(DWORD v, Face \*faces, Tab<DWORD> \*flist=NULL);**

**Remarks:**

This method is available in release 3.0 and later only.

Each vertex has a list of edges in the data member AdjEdgeList::list. This method reorders the elements of that list so that the edges are in order going around the vertex. The direction should be counterclockwise as seen from outside the mesh surface, though this necessarily breaks down with some rats' nest situations.

**Parameters:**

**DWORD v**

The vertex whose edges should be ordered.

**Face \*faces**

A pointer to the faces for this mesh.

**Tab<DWORD> \*flist=NULL**

If non-NULL, this points to an array where the faces using this vertex should be stored (in order).

**Prototype:**

**void OrderAllEdges(Face \*faces);**

**Remarks:**

This method is available in release 3.0 and later only.

This method simply calls **OrderVertEdges()** on all the vertices.

**Parameters:**

**Face \*faces**

A pointer to the faces for this mesh.

**Prototype:**

**void GetFaceList(DWORD v, Tab<DWORD> & flist);**

**Remarks:**

This method is available in release 3.0 and later only.

This places a list of all faces using this vertex in flist. The faces are in no

particular order.

**Parameters:**

**DWORD v**

The vertex to check.

**Tab<DWORD> & flist**

The table of faces.

**Prototype:**

**void AddEdgeToVertex(DWORD v,DWORD e);**

**Remarks:**

This method is available in release 3.0 and later only.

Appends the specified edge to the specified vertex.

**Parameters:**

**DWORD v**

The vertex the edge is added to.

**DWORD e**

The edge to add.

**Prototype:**

**void AddEdge(DWORD fi, DWORD v1, DWORD v2);**

**Remarks:**

This is used internally.

**Operators:**

**Prototype:**

**DWORDTab& operator[](int i);**

**Remarks:**

Array access operator. Returns the 'i-th' list element.

## Class AdjFaceList

See Also: [Class AdjFace](#), [Class AdjEdgeList](#).

class AdjFaceList

### Description:

This class represents a face adjacency list for meshes. This class require the mesh to be constructed so that each edge has exactly one or two faces. It will work with other meshes but may give misleading results (developers may wish to call **mesh.RemoveDegenerateFaces()** to attempt to correct the mesh to work with this class). All methods of this class are implemented by the system.

### Data Members:

public:

**Tab<AdjFace> list;**

The table of adjacent faces, one for each face.

### Methods:

#### Prototype:

**AdjFaceList(Mesh& mesh,AdjEdgeList& el);**

#### Remarks:

Constructor. The mesh and edge list passed define the mesh used to construct the face list.

### Operators:

#### Prototype:

**AdjFace& operator[](int i);**

#### Remarks:

Access operator. Returns the set of adjacent face for the i-th face.

# Class FaceElementList

See Also: [Class AdjEdgeList](#), [Class AdjFaceList](#).

class FaceElementList

## Description:

This class may be used to access the various elements that compose a mesh. Given a Mesh object **mesh**, the calls below allow a developer to access the element number for a particular face, and the total number of elements in the mesh.

**AdjEdgeList ae(mesh);**

**AdjFaceList af(mesh, ae);**

**FaceElementList elem(mesh, af);**

All methods of this class are implemented by the system.

## Data Members:

public:

**DWORDTab elem;**

This gives the element number for a particular face, i.e. **elem[i]** gives the element number for face **i**.

**DWORD count;**

The total number of elements.

## Methods:

### Prototype:

**FaceElementList(Mesh &mesh, AdjFaceList& af);**

### Remarks:

Constructor.

### Parameters:

**Mesh &mesh**

The mesh that the element list is being built for.

**AdjFaceList& af**

The face list for the mesh.

## Operators:

**Prototype:**

**DWORD operator[](int i);**

**Remarks:**

Accesses the i-th element of the table.

# Class FaceClusterList

See Also: [Class AdjFaceList](#), [Class BitArray](#).

class FaceClusterList

## Description:

This is a list of face "clusters" for a given mesh. A typical application would be in Edit(able) Mesh, where the user has selected two separate groups of faces on different parts of the mesh and wants to extrude them both, or rotate both around their local centers. Each "cluster" is a contiguous group of selected faces. Like AdjEdgeLists and AdjFaceLists, this class is only defined in relation to some mesh.

This class may be used to group faces together based on the angle between their normals or by their selection status.

All methods of this class are implemented by the system. Note that the functionality provided by this class is not available in the 1.0 release of the SDK. Later releases (1.1, 1.2, etc) do support it.

## Data Members:

public:

### **DWORDTab clust;**

The cluster number (id), one for each face. Non-selected faces have **UNDEFINED** for their id.

The cluster IDs of all the faces -- this table has size **mesh::numFaces**. **clust[i]** is **UNDEFINED** if face **i** is not in any cluster (ie is unselected).

### **DWORD count;**

The number of clusters.

## Methods:

### Prototype:

**FaceClusterList(Mesh \*mesh, AdjFaceList& adj, float angle, BOOL useSel);**

### Remarks:

Constructor. This version separates clusters using a minimum angle and optionally the selection set. A developer creates one of these cluster lists by

specifying the mesh, the face list and an angle. What is built is a cluster number for each face identifying what cluster it is in.

For example, if you create one of these for a sphere and set the angle threshold to 90 degrees, you would get back one cluster, and the cluster id for everything would be 0. If you ran it on a box, and you set the angle to < 90 degrees, you would get back 6 ids. Two faces in the box would have id 0, two would have id 1, etc.

**Parameters:**

**Mesh \*mesh**

The mesh to create the list for.

**AdjFaceList& adj**

The face list for this mesh.

**float angle**

The maximum angle (in radians) that can be used in joining adjacent faces into the same cluster.

**BOOL useSel**

If FALSE, selection is ignored and all faces are grouped into clusters by angle. If TRUE, only selected faces are grouped into clusters, but angle is still relevant. Non-selected faces will have **UNDEFINED** for their id.

**Prototype:**

**FaceClusterList(BitArray& fsel, AdjFaceList& adj);**

**Remarks:**

Constructor. This version separates clusters using the selection set. In this case a cluster is defined as a set of faces that are selected and are adjacent. For example you could have a sphere with some faces selected on one side, and another group of faces selected on the other side. Each group of adjacent and selected faces would comprise clusters within the mesh. This is used for example by the axis tripods in 3ds max where each selected group of faces gets their own coordinate system.

In this case the unselected faces will not be in any cluster. These store the value **UNDEFINED** for their id.

**Parameters:**

**BitArray& fsel**

This bit array defines the face selected state that the clusters will be grouped by. Each bit in the bit array corresponds to the parallel index in the mesh face table.

### **AdjFaceList& adj**

The face list for this mesh.

### **Prototype:**

```
void MakeVertCluster(Mesh &mesh, Tab<DWORD> &vclust);
```

### **Remarks:**

This method is available in release 3.0 and later only.

Creates a list of cluster IDs for vertices.

### **Parameters:**

#### **Mesh &mesh**

The mesh associated with this FaceClusterList.

#### **Tab<DWORD> &vclust**

This is where the output goes: vclust is set to size mesh.numVerts, and the value of each entry in this table tells which cluster the vertex has been assigned to, based on the faces it's on. If vertex "v" is not in any clusters (ie none of the faces that use it are in any clusters), vclust[v] is UNDEFINED.

In cases where a vertex is in two clusters, the larger face index is dominant. (In other words, if a vertex 6 is on faces 2 and 7, which are in two separate clusters, and face 9, which isn't in any cluster, it gets its cluster ID from face 7. This can happen if two selection regions touch at a vertex instead of along an edge.)

### **Prototype:**

```
void GetNormalsCenters(Mesh &mesh, Tab<Point3> &norm,  
Tab<Point3> &ctr);
```

### **Remarks:**

This method is available in release 3.0 and later only.

Computes average normals and centers for all face clusters. Within a cluster, normals are weighted by the area of the face -- a face twice as big contributes twice as much to the cluster normal. (Mathematically, we just total up the non-normalized cross-products of each face, which are equivalent to  $2 \cdot (\text{area}) \cdot (\text{face}$

normal). Then we normalize the cluster total.) Face centers are directly averaged, without weighting.

**Parameters:**

**Mesh &mesh**

The mesh associated with this FaceClusterList.

**Tab<Point3> &norm**

The average normal table to store the results in. This is set to size FaceClusterList::count, the number of clusters.

**Tab<Point3> &ctr**

The average center table to store the results in. This is set to size FaceClusterList::count, the number of clusters.

**Prototype:**

**void GetBorder(DWORD clustID, AdjFaceList &af,  
Tab<DWORD> &cbord);**

**Remarks:**

This method is available in release 3.0 and later only.

Each face cluster is a set of faces connected by shared edges. This method finds a cluster's boundary, which can be expressed as a sequence of edges on faces in the cluster (where the other side of each edge has no face or has a face not in this cluster). If there is more than one boundary, as for instance in the faces that make up the letter "o" in a ShapeMerge with Text, both boundaries are returned in no particular order.

**Parameters:**

**DWORD clustID**

The cluster to get the border of.

**AdjFaceList &af**

The adjacent face list associated with this FaceClusterList.

**Tab<DWORD> &cbord**

The table where the output goes. If there are no borders (as for instance in a sphere with all faces selected), it remains empty. Otherwise, this is filled with a series of edge indices, then an UNDEFINED to mark the end of each border. So for instance if this cluster represents the front face of a default box, cbord will contain 4 edge indices and an UNDEFINED. If the cluster represents all

the side faces of a cylinder, but not the top or bottom, there are two borders: on 24-sided cylinder, you'd get the 24 edge indices representing the bottom lip of the cylinder, then an UNDEFINED, then the 24 edge indices representing the top lip, followed by another UNDEFINED. (As elsewhere, edges are indexed by  $face*3+eid$ , where  $face$  is the face (in the cluster) the edge is on, and  $eid$  is the index of the edge on that face.)

**Prototype:**

```
void GetOutlineVectors(Mesh & m, AdjFaceList &af,  
Tab<Point3> &cnorms, Tab<Point3> &odir);
```

**Remarks:**

This method is available in release 3.0 and later only.

This creates "outline" directions for the vertices on the edge of the clusters. These are used in Edit(able) Mesh's new "Bevel" operation (when you Bevel by "Group"). These vectors, which are all perpendicular to the cluster normals, point in the direction and speed vertices must travel in order to move the edges out at a consistent rate without changing the shape of the outline of the cluster. To see how this works, create a Prism in 3ds max ("Extended Primitives") with dimensions like 20 x 40 x 40 x 40, and apply an Edit Mesh. Select all the faces on the top of the prism, and spin the Bevel spinner up and down. Notice that the vertex at the sharpest point moves faster than the other 2, but that the edges all remain parallel to their original positions. The essence of Outlining is that the edges move at a constant rate, and the vertices move faster or slower to make this happen. (This strategy is also used in the Bevel and Path Bevel modifiers.)

**Parameters:**

**Mesh & m**

The mesh associated with this FaceClusterList

**AdjFaceList &af**

The adjacent face list associated with this FaceClusterList

**Tab<Point3> &cnorms**

The cluster normals, as computed by GetNormalsCenters

**Tab<Point3> &odir**

A table to put the outline direction result in. This is set to size mesh.numVerts.

Entries for vertices that are not on a cluster border are all (0,0,0). Entries for cluster border vertices are scaled, such that if you move all vertices the specified amount, each cluster's border edges will move by one 3ds max unit.

**Operators:**

**Prototype:**

**DWORD operator[](int i)**

**Remarks:**

Access operator. Returns the cluster ID for face **i**.

## Class NURBSObject

See Also: [Class NURBSSurface](#), [Class NURBSControlVertex](#), [Class NURBSPoint](#), [Class NURBSCurve](#), [Class NURBSSet](#), [List of NURBSObjects Types](#).

class NURBSObject

### Description:

This class is available in release 2.0 and later only.

This is the base class for many of the other classes in the NURBS API. It provides a common set of methods that each of them use. It has methods to get and set the name of the item, and methods to deal with error processing. To determine the type of object the derived class is use the method **GetType()**.

All methods of this class are implemented by the system.

### Data Members:

protected:

**TCHAR mName[NURBS\_NAME\_SIZE];**

The name of the NURBS object. The maximum length of this name including the terminating NULL is specified using the following #define: **#define NURBS\_NAME\_SIZE 80**

**NURBSType mType;**

The type of NURBS object this is. See [List of NURBSObjects Types](#).

**NURBSKind mKind;**

The kind of NURBS object this is. See [List of NURBSObject Kinds](#).

**NURBSId mId;**

This is the ID of the NURBS object used to specify the parent object in many of the dependent point, curves and surface classes. This ID is not persistent across sessions and should not be saved to a file. A NURBSId is defined as follows: **typedef unsigned long NURBSId;**

**Object \*mpObject;**

When an object is instantiated in the 3ds max scene this pointer is filled in. For example, if you use the function **CreateNURBSObject()** and pass a **NURBSSet**, this data member is filled in to point to the actual editable NURBS object in 3ds max that this **NURBSObject** is a part of.

**NURBSSet\* mpNSet;**

When an object is instantiated in the 3ds max scene this pointer is filled in. This points to the **NURBSSet** this object is contained within.

**BOOL mSelected;**

TRUE if the object is selected; otherwise FALSE.

**Methods:**

protected:

**Prototype:**

**void Clean(NURBSIdTab ids);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the **mId** to 0 and **mpObject** pointer to NULL. This effectively breaks the relation between this **NURBSObject** and a **NURBSSet**.

**Parameters:**

**NURBSIdTab ids**

This parameter is not used.

public:

**Prototype:**

**NURBSObject();**

**Remarks:**

Constructor. The data members are initialized as follows:

**mName[0] = '\0';**

**mId = 0;**

**mpObject = NULL;**

**mpNSet = NULL;**

**mSelected = FALSE;**

**Prototype:**

**~NURBSObject();**

**Remarks:**

Destructor.

**Prototype:**

**void SetName(TCHAR \*name);**

**Remarks:**

Sets the name of the item to the specified string.

**Parameters:**

**TCHAR \*name**

The name to set.

**Prototype:**

**TCHAR \*GetName();**

**Remarks:**

Returns a pointer to the name of the item.

**Prototype:**

**NURBStype GetType();**

**Remarks:**

Returns the specific type of object this is. See [List of NURBSObjects Types](#).

**Prototype:**

**NURBStype GetKind();**

**Remarks:**

Returns the specific kind of object this is. See [List of NURBSObjects Kinds](#).

**Prototype:**

**NURBSId GetId();**

**Remarks:**

Returns the **NURBSId** of this **NURBSObject**. This ID is not persistent across sessions and should not be saved to a file.

**Prototype:**

**void SetId(NURBSId id);**

**Remarks:**

Sets the **NURBSId** of this **NURBSObject**.

**Parameters:**

**NURBSId id**

The ID to set.

**Prototype:**

**int GetIndex();**

**Remarks:**

Returns the index in the **NURBSSet** of this object or -1 if there isn't an associated **NURBSSet**.

**Prototype:**

**void SetNSet(NURBSSet \*nset);**

**Remarks:**

Sets the pointer to the **NURBSSet** maintained by the object.

**Parameters:**

**NURBSSet \*nset**

The pointer to set.

**Prototype:**

**NURBSSet\* GetNSet();**

**Remarks:**

Returns a pointer to the **NURBSSet** maintained by this class.

**Prototype:**

**void SetObject(Object \*object);**

**Remarks:**

Sets the pointer to the 3ds max editable NURBS object maintained by this object.

**Parameters:**

## **Object \*object**

The pointer to set.

### **Prototype:**

**Object\* GetMAXObject();**

### **Remarks:**

Returns a pointer to the **Object** maintained by this class.

### **Prototype:**

**BOOL IsSelected();**

### **Remarks:**

Returns TRUE if the object is selected; otherwise FALSE.

### **Prototype:**

**void SetSelected(BOOL set);**

### **Remarks:**

Sets the object to selected or not.

### **Parameters:**

**BOOL set**

TRUE to select the object; FALSE to de-select it.

### **Operators:**

#### **Prototype:**

**NURBSObject & operator=(const NURBSObject& pt);**

#### **Remarks:**

Assignment operator.

#### **Parameters:**

**const NURBSObject& pt**

The NURBSObject to assign.

## Class NURBSSurface

See Also: [Class NURBSObject](#), [NURBSTextureSurface](#), [Template Class Tab](#), [Class NURBSTextureChannelSet](#).

```
class NURBSSurface : public NURBSObject
```

### Description:

This class is available in release 2.0 and later only.

This class describes the properties of a NURBS surface. This includes its material ID, texture/tiling options, renderable state, open/closed state, and normal inverted state. The **Evaluate()** method is used to compute points and tangents on the surface.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSCVSurface;  
friend class NURBSPointSurface;  
friend class NURBSBlendSurface;  
friend class NURBSNBlendSurface;  
friend class NURBSOffsetSurface;  
friend class NURBSXFormSurface;  
friend class NURBSMirrorSurface;  
friend class NURBSCapSurface;  
friend class NURBSIsoCurve;  
friend class NURBSProjectVectorCurve;  
friend class NURBSProjectNormalCurve;  
friend class NURBSSurfSurfIntersectionCurve;  
friend class NURBSCurveOnSurface;  
friend class NURBSPointCurveOnSurface;  
friend class NURBSMultiCurveTrimSurface;  
friend class NURBSTextureChannel;  
friend class NURBSTextureChannelSet;
```

## Data Members:

protected:

### **BOOL mGenUVs;**

TRUE if the 'Generate Mapping Coordinates' checkbox is on; otherwise FALSE.

### **BOOL mFlipNormals;**

TRUE if the surface normals are inverted; otherwise FALSE.

### **BOOL mRenderable;**

TRUE if the surface may be rendered; otherwise FALSE.

### **int mMatID;**

The material ID for the surface.

### **BOOL mClosedInU, mClosedInV;**

The surface closed flags.

### **NURBSTextureChannelSet mTextureChannelSet;**

This data member is available in release 3.0 and later only.

This is the set of texture mapping channels used by this surface.

### **BOOL mFlipNormals;**

TRUE if normals are flipped for the surface; otherwise FALSE.

### **BOOL mRenderable;**

TRUE if the surface is renderable; otherwise FALSE.

### **int mMatID;**

The zero based material ID for the surface.

### **BOOL mClosedInU**

TRUE if the surface is closed in U; otherwise FALSE.

### **BOOL mClosedInV;**

TRUE if the surface is closed in V; otherwise FALSE.

### **TessApprox \*mpVTess;**

This data member is available in release 3.0 and later only.

Points to the TessApprox object for the viewport (optional).

### **TessApprox \*mpRTess;**

This data member is available in release 3.0 and later only.

Points to the TessApprox object for the production renderer (optional).

### **TessApprox \*mpRTessDisp;**

This data member is available in release 3.0 and later only.  
Points to the TessApprox object for displacement mapping (optional).

**TessApprox \*mpVTessCurve;**

This data member is available in release 3.0 and later only.  
Points to the TessApprox object for tesselating curves in the viewports(optional).

**TessApprox \*mpRTessCurve;**

This data member is available in release 3.0 and later only.  
Points to the TessApprox object for tesselating curves in the production renderer (optional).

**Methods:**

**Prototype:**

**NURBSSurface();**

**Remarks:**

Constructor. The surface is initialized as follows:

**mKind = kNURBSSurface;**

**mFlipNormals = FALSE;**

**mRenderable = TRUE;**

**mMatID = 1;**

**mpVTess = NULL;**

**mpRTess = NULL;**

**mpRTessDisp = NULL;**

**mpVTessCurve = NULL;**

**mpRTessCurve = NULL;**

**Prototype:**

**~NURBSSurface();**

**Remarks:**

Destructor. Any **TessApprox** objects are freed.

**Prototype:**

**BOOL Renderable();**

**Remarks:**

Returns TRUE if the surface is renderable; otherwise FALSE.

**Prototype:**

**void Renderable(BOOL state);**

**Remarks:**

Sets the renderable flag to the specified state.

**Parameters:**

**BOOL state**

TRUE for renderable; FALSE for non-renderable.

**Prototype:**

**BOOL FlipNormals();**

**Remarks:**

Returns the state of the flip normals flag.

**Prototype:**

**void FlipNormals(BOOL state);**

**Remarks:**

Set the state of the flip normals flag.

**Parameters:**

**BOOL state**

TRUE if the normals should be flipped; FALSE for not flipped.

**Prototype:**

**BOOL GenerateUVs(int channel = 0);**

**Remarks:**

Returns TRUE if the generate UV mapping coordinates flag is set for the specified channel; otherwise FALSE.

**Parameters:**

**int channel = 0**

The channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

**void SetGenerateUVs(BOOL state, int channel = 0);**

**Remarks:**

Sets the state of the generate UV mapping coordinates flag.

**Parameters:**

**BOOL state**

TRUE for on; FALSE for off.

**int channel = 0**

The channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

**int MatID();**

**Remarks:**

Returns the material ID.

**Prototype:**

**void MatID(int id);**

**Remarks:**

Sets the materials ID to the specified value.

**Parameters:**

**int id**

Specifies the material ID to set.

**Prototype:**

**Point2 GetTextureUVs(TimeValue t, int i, int channel = 0);**

**Remarks:**

Returns the specified texture coordinate.

**Parameters:**

**TimeValue t**

The time to get the texture UVs.

**int i**

The zero based index of the coordinate to return. This value must be  $\geq 0$  and  $< 4$ .

**int channel = 0**

The channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

**void SetTextureUVs(TimeValue t, int i, Point2 pt, int channel = 0);**

**Remarks:**

Set the specified texture coordinate to the specified value.

**Parameters:**

**TimeValue t**

The time to set the texture UVs.

**int i**

The zero based index of the texture coordinate to set. This value must be  $\geq 0$  and  $< 4$ .

**Point2 pt**

The texture coordinate value to set.

**int channel = 0**

The channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

**void GetTileOffset(TimeValue t, float &ut, float &vt, float &uo, float &vo, int channel = 0);**

**Remarks:**

Retrieves the texture tiling and offset values for the surface.

**Parameters:**

**TimeValue t**

The time to get the tile offset.

**float &ut**

The tiling value in the U direction.

**float &vt**

The tiling value in the V direction.

**float &uo**

The offset value in the U direction.

**float &vo**

The offset value in the V direction.

**int channel = 0**

The channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

```
void SetTileOffset(TimeValue t, float ut, float vt, float uo, float vo,  
int channel = 0);
```

**Remarks:**

Sets the texture tiling and offset values for the surface.

**Parameters:****TimeValue t**

The time to set the tile offset.

**float ut**

The tiling value in the U direction.

**float vt**

The tiling value in the V direction.

**float uo**

The offset value in the U direction.

**float vo**

The offset value in the V direction.

**int channel = 0**

The channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

**NURBSTextureSurface& GetTextureSurface(int channel);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a reference to the texture surface used by this surface for the specified channel.

**Parameters:**

**int channel**

The channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**Prototype:**

**void SetTextureSurface(int channel, NURBSTextureSurface& texSurf);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the texture surface used by this surface for the specified channel.

**Parameters:**

**int channel**

The channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**NURBSTextureSurface& texSurf**

The texture surface to set.

**Prototype:**

**int NumChannels();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the number of channels used by the surface.

**Prototype:**

**int GetChannelFromIndex(int index);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a channel number corresponding to the specified index into the **NURBSTextureChannelSet**.

**Parameters:**

**int index**

The zero based index into the **NURBSTextureChannelSet**.

**Prototype:**

**BOOL IsClosedInU();**

**Remarks:**

Returns TRUE if the surface is closed in the U direction; otherwise FALSE.

**Prototype:**

**BOOL IsClosedInV();**

**Remarks:**

Returns TRUE if the surface is closed in the V direction; otherwise FALSE.

**Prototype:**

**BOOL Evaluate(TimeValue t, double u, double v, Point3& pt, Point3& dPdU, Point3& dPdV);**

**Remarks:**

Retrieves the point on the surface, and the u and v derivatives based on the parameters u and v.

**Parameters:**

**TimeValue t**

The time at which to evaluate the surface.

**double u**

The value at which to evaluate the surface in u. This value must be between the **uMin** and **uMax** as returned from **GetParameterRange()**.

**double v**

The value at which to evaluate the surface in v. This value must be between the **vMin** and **vMax** as returned from **GetParameterRange()**.

**Point3& pt**

The point on the surface.

**Point3& dPdU**

The derivative along u.

**Point3& dPdV**

The derivative along v.

**Return Value:**

TRUE if the method was able to evaluate the surface; otherwise FALSE.

**Prototype:**

**BOOL Evaluate(TimeValue t, double u, double v, Point3& pt, Point3& dPdU, Point3& dPdV, Point3& d2PdU2, Point3& d2PdV2, Point3& d2PdUdV);**

**Remarks:**

Retrieves the point on the surface, and the u and v derivatives and second derivatives based on the parameters u and v.

**Parameters:****TimeValue t**

The time at which to evaluate the surface.

**double u**

The value at which to evaluate the surface in u. This value must be between the **uMin** and **uMax** as returned from **GetParameterRange()**.

**double v**

The value at which to evaluate the surface in v. This value must be between the **vMin** and **vMax** as returned from **GetParameterRange()**.

**Point3& pt**

The point on the surface.

**Point3& dPdU**

The derivative along u.

### **Point3& dPdV**

The derivative along v.

### **Point3& d2PdU2**

The second derivative along u.

### **Point3& d2PdV2**

The second derivative along v.

### **Point3& d2PdUdV**

#### **Return Value:**

TRUE if the method was able to evaluate the surface; otherwise FALSE.

#### **Prototype:**

```
void GetParameterRange(TimeValue t, double& uMin, double& uMax, double& vMin, double& vMax);
```

#### **Remarks:**

Retrieves the minimum and maximum valid values for **u** and **v** as passed to **Evaluate()**.

#### **Parameters:**

##### **TimeValue t**

The time at which to get the parameter range.

##### **double& uMin**

The minimum value in u is returned here.

##### **double& uMax**

The maximum value in v is returned here.

##### **double& vMin**

The minimum value in u is returned here.

##### **double& vMax**

The maximum value in v is returned here.

#### **Prototype:**

```
BOOL GetNURBSData(TimeValue t, int& degreeInU, int& degreeInV, int& numInU, int& numInV, NURBSCVTab& cvs, int& numKnotsInU, int& numKnotsInV, NURBSKnotTab uKnots,
```

**NURBSKnotTab vKnots);**

**Remarks:**

This method is available in release 2.5 and later only.

Retrieves data about the NURBSSurface at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the NURBS information.

**int& degreeInU**

The degree of the surface in U.

**int& degreeInV**

The degree of the surface in V.

**int& numInU**

The number of CVs in U.

**int& numInV**

The number of CVs in V.

**NURBSCVTab& cvs**

The table of CVs. Note: **typedef Tab<NURBSControlVertex>**

**NURBSCVTab;**

**int& numKnotsInU**

The number of knots in U.

**int& numKnotsInV**

The number of knots in V.

**NURBSKnotTab uKnots**

A table of knots in U. Note: **typedef Tab<double> NURBSKnotTab;**

**NURBSKnotTab vKnots**

A table of knots in V.

**Return Value:**

TRUE if the data was retrieved; otherwise FALSE.

**Prototype:**

**BOOL GetCLPTextureSurfaceData(TimeValue t, int channel, int& degreeInU, int& degreeInV, int& numInU, int& numInV,**

**NURBSCVTab& cvs, int& numKnotsInU, int& numKnotsInV,  
NURBSKnotTab uKnots, NURBSKnotTab vKnots);**

**Remarks:**

This method is available in release 2.5 and later only.

This method retrieves the Chord Length Parameterization Texture Surface data.

**Parameters:**

**TimeValue t**

The time at which to retrieve the data.

**int channel**

The texture channel. This is a number in the range 0 and 98 which correspond to 1 to 99 in the user interface.

**int& degreeInU**

The degree of the surface in U.

**int& degreeInV**

The degree in V.

**int& numInU**

The number of CVs in U.

**int& numInV**

The number of CVs in V.

**NURBSCVTab& cvs**

The table of CVs. Note: **typedef Tab<NURBSControlVertex>**

**NURBSCVTab;**

**int& numKnotsInU**

The number of knots in U.

**int& numKnotsInV**

The number of knots in V.

**NURBSKnotTab uKnots**

A table of knots in U. Note: **typedef Tab<double> NURBSKnotTab;**

**NURBSKnotTab vKnots**

A table of knots in V.

**Return Value:**

TRUE if the data was retrieved; otherwise FALSE.

**Prototype:**

```
int NumTrimLoops(TimeValue t);
```

**Remarks:**

This method is available in release 2.5 and later only.

Returns the number of trim loops at the specified time. Each loop may be made up of several curves.

**Parameters:**

**TimeValue t**

The time at which to return the number.

**Prototype:**

```
int NumCurvesInLoop(TimeValue t, int loop);
```

**Remarks:**

This method is available in release 2.5 and later only.

Returns the number of curves in the specified trim loop.

**Parameters:**

**TimeValue t**

The time at which to return the number.

**int loop**

The zero based index of the trim loop.

**Prototype:**

```
BOOL Get2dTrimCurveData(TimeValue t, int loop, int curve, int&  
degree, int& numCVs, NURBSCVTab& cvs, int& numKnots,  
NURBSKnotTab knots);
```

**Remarks:**

This method is available in release 2.5 and later only.

Retrieves data about the specified 2D trim curve in use by the surface.

**Parameters:**

**TimeValue t**

The time at which to retrieve the data.

**int loop**

The zero based index of the trim loop.

**int curve**

The zero based index of the trim curve within the loop.

**int& degree**

The degree of the curve is returned here.

**int& numCVs**

The number of CVs.

**NURBSCVTab& cvs**

The table of CVs. Note: **typedef Tab<NURBSControlVertex>**

**NURBSCVTab;****int& numKnots**

The number of knots.

**NURBSKnotTab knots**

A table of knots. Note: **typedef Tab<double> NURBSKnotTab;**

**Return Value:**

TRUE if the data was retrieved; otherwise FALSE.

**Prototype:**

**BOOL Get3dTrimCurveData(TimeValue t, int loop, int curve, int& degree, int& numCVs, NURBSCVTab& cvs, int& numKnots, NURBSKnotTab knots);**

**Remarks:**

This method is available in release 2.5 and later only.

Retrieves data about the specified 2D trim curve in use by the surface.

**Parameters:****TimeValue t**

The time at which to retrieve the data.

**int loop**

The zero based index of the trim loop.

**int curve**

The zero based index of the trim curve within the loop.

**int& degree**

The degree of the curve is returned here.

**int& numCVs**

The number of CVs.

**NURBSCVTab& cvs**

The table of CVs. Note: **typedef Tab<NURBSControlVertex>**

**NURBSCVTab;****int& numKnots**

The number of knots.

**NURBSKnotTab knots**

A table of knots. Note: **typedef Tab<double> NURBSKnotTab;**

**Return Value:**

TRUE if the data was retrieved; otherwise FALSE.

**Prototype:**

**TessApprox\* GetProdTess(NURBSTessType  
type=kNTessSurface);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the TessApprox object used for production rendering of the specified type.

**Parameters:**

**NURBSTessType type=kNTessSurface**

The tessellation type. See [List of NURBSTessTypes](#).

**Prototype:**

**TessApprox\* GetViewTess(NURBSTessType  
type=kNTessSurface);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns a pointer to the TessApprox object used for viewport rendering of the specified type.

**Parameters:**

**NURBSTessType type=kNTessSurface**

The tessellation type. See [List of NURBSTessTypes](#).

**Prototype:**

**void SetProdTess(TessApprox& tess, NURBSTessType type=kNTessSurface);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the TessApprox object used for production rendering of the specified type.

**Parameters:**

**TessApprox& tess**

The object to set.

**NURBSTessType type=kNTessSurface**

The tessellation type. See [List of NURBSTessTypes](#).

**Prototype:**

**void SetViewTess(TessApprox& tess, NURBSTessType type=kNTessSurface);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the TessApprox object used for viewport rendering of the specified type.

**Parameters:**

**TessApprox& tess**

The object to set.

**NURBSTessType type=kNTessSurface**

The tessellation type. See [List of NURBSTessTypes](#).

**Prototype:**

**void ClearViewTess(NURBSTessType type=kNTessSurface);**

**Remarks:**

This method is available in release 3.0 and later only.

Clears (deletes) the TessApprox object used for viewport rendering of the specified type.

**Parameters:**

**NURBSTessType type=kNTessSurface**

The tessellation type. See [List of NURBSTessTypes](#).

**Prototype:**

**void ClearProdTess(NURBSTessType type=kNTessSurface);**

**Remarks:**

This method is available in release 3.0 and later only.

Clears (deletes) the TessApprox object used for production rendering of the specified type.

**Parameters:**

**NURBSTessType type=kNTessSurface**

The tessellation type. See [List of NURBSTessTypes](#).

**Operators:**

**Prototype:**

**NURBSSurface & operator=(const NURBSSurface& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSSurface& curve**

The surface to assign.

## Class NURBSControlVertex

See Also: [Class NURBSObject](#), [Class NURBSSurface](#), [Class NURBSSet](#), [Class NURBSPoint](#), [Class NURBSCurve](#), [Class NURBSFuseSurfaceCV](#), [Class NURBSFuseCurveCV](#).

```
class NURBSControlVertex : public NURBSObject
```

### Description:

This class is available in release 2.0 and later only.

This class represents a control vertex in a NURBS curve. Methods are available to get and set the point position, and get/set the weight.

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
NURBSControlVertex();
```

#### Remarks:

Constructor. The name is set to NULL, the points are set to 0.0f and the weight is set to 1.0.

#### Prototype:

```
~NURBSControlVertex();
```

#### Remarks:

Destructor.

#### Prototype:

```
void SetPosition(TimeValue t, Point3 pt);
```

#### Remarks:

Sets the position of the control vertex.

#### Parameters:

**TimeValue t**

The time to set the position.

**Point3 pt**

Specifies the position to set.

**Prototype:**

**void SetPosition(TimeValue t, float x, float y, float z);**

**Remarks:**

Sets the position of the control vertex.

**Parameters:**

**TimeValue t**

The time to set the position.

**float x, float y, float z**

Specifies the position to set.

**Prototype:**

**void SetPosition(TimeValue t, double x, double y, double z);**

**Remarks:**

Sets the position of the control vertex.

**Parameters:**

**TimeValue t**

The time to set the position.

**double x, double y, double z**

Specifies the position to set.

**Prototype:**

**Point3 GetPosition(TimeValue t);**

**Remarks:**

Returns the position of the control vertex.

**Parameters:**

**TimeValue t**

The time to get the position.

**Prototype:**

**void GetPosition(TimeValue t, float& x, float& y, float& z);**

**Remarks:**

Retrieves the position of the control vertex.

**Parameters:**

**TimeValue t**

The time to get the position.

**float& x**

The X coordinates is returned here.

**float& y**

The Y coordinates is returned here.

**float& z**

The Z coordinates is returned here.

**Prototype:**

**void GetPosition(TimeValue t, double& x, double& y, double& z);**

**Remarks:**

Retrieves the position of the control vertex.

**Parameters:**

**TimeValue t**

The time to get the position.

**double& x**

The X coordinates is returned here.

**double& y**

The Y coordinates is returned here.

**double& z**

The Z coordinates is returned here.

**Prototype:**

**void SetWeight(TimeValue t, float wt);**

**Remarks:**

Sets the weight of the control vertex.

**Parameters:**

**TimeValue t**

The time to set the weight.

**float wt**

Specifies the weight to set. This is a value greater than zero. Larger values cause the CV to have a greater effect, thus the curve or surface will try to pass closer to the CV.

**Prototype:**

**void SetWeight(TimeValue t, double wt);**

**Remarks:**

Sets the weight of the control vertex.

**Parameters:**

**TimeValue t**

The time to set the weight.

**double wt**

Specifies the weight to set. . This is a value greater than zero. Larger values cause the CV to have a greater effect, thus the curve or surface will try to pass closer to the CV.

**Prototype:**

**void GetWeight(TimeValue t, float& wt);**

**Remarks:**

Retrieves the weight of the control vertex.

**Parameters:**

**TimeValue t**

The time to get the weight.

**float& wt**

The weight is returned here.

**Prototype:**

**double GetWeight(TimeValue t);**

**Remarks:**

Returns the weight of the control vertex.

**Parameters:**

**TimeValue t**

The time to get the weight.

**Prototype:**

**void GetWeight(TimeValue t, double& wt);**

**Remarks:**

Retrieves the weight of the control vertex.

**Parameters:**

**TimeValue t**

The time to get the weight.

**double& wt**

The weight is returned here.

**Operators:****Prototype:**

**NURBSControlVertex & operator=(const NURBSControlVertex& cv);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSControlVertex& cv**

The control vertex to assign.

**Prototype:**

**BOOL operator==(const NURBSControlVertex& cv);**

**Remarks:**

Equality operator. Compares if the coordinates and weight are the same.

**Parameters:**

**const NURBSControlVertex& cv**

The control vertex to compare.

**Return Value:**

TRUE if the CVs are equal; otherwise FALSE.

**Prototype:**

**BOOL operator!=(const NURBSControlVertex& cv);**

**Remarks:**

Inequality operator. Compares if the coordinates or weight are **not** the same.

**Parameters:**

**const NURBSControlVertex& cv**

The control vertex to compare.

**Return Value:**

TRUE if the CVs are not equal; otherwise FALSE.

## Class NURBSPoint

See Also: [Class NURBSObject](#), [Class NURBSSurface](#), [Class NURBSSet](#), [Class NURBSControlVertex](#), [Class NURBSCurve](#), [Class NURBSFuseSurfaceCV](#), [Class NURBSFuseCurveCV](#).

```
class NURBSPoint : public NURBSObject
```

### Description:

This class is available in release 2.0 and later only.

This class describes a point in 3 space using double precision X, Y and Z coordinates. Methods are available for getting the point coordinates a floats, doubles or a **Point3**.

All methods of this class are implemented by the system.

### Data Members:

protected:

```
double mX, mY, mZ;
```

The X, Y and Z coordinates for the point.

### Methods:

#### Prototype:

```
Point3 GetPosition();
```

#### Remarks:

Returns the point position as a **Point3**.

#### Prototype:

```
void GetPosition(float& x, float& y, float& z);
```

#### Remarks:

Retrieves the point position using single precision.

#### Parameters:

```
float& x
```

The X coordinate is returned here.

```
float& y
```

The Ycoordinate is returned here.

**float& z**

The Z coordinate is returned here.

**Prototype:**

**void GetPosition(double& x, double& y, double& z);**

**Remarks:**

Retrieves the point position using double precision.

**Parameters:**

**double& x**

The X coordinate is returned here.

**double& y**

The Y coordinate is returned here.

**double& z**

The Z coordinate is returned here.

## Class NURBSFuseSurfaceCV

See Also: [Class NURBSSurface](#), [Class NURBSSet](#), [Class NURBSControlVertex](#), [Class NURBSPoint](#), [Class NURBSCurve](#), [Class NURBSFuseCurveCV](#).

class NURBSFuseSurfaceCV

### Description:

This class is available in release 2.0 and later only.

This class may be used with a NURBSSet to fuse two CVs in a surface. This causes the CVs to reference one another so if you move one the other moves with it.

All methods of this class are implemented by the system.

### Data Members:

public:

**int mSurf1, mSurf2;**

The zero based indices of the surfaces to fuse. These may be the same value. Note that this is **not** the index in the **NURBSSet** of these items. Rather it is the index of CV surface in list of CV surfaces in the NURBSSet. For instance, if there were first two CV curves and then two CV surfaces in the **NURBSSet**, **mSurf1** would be 0 and **mSurf2** would be 1 since the CV curves don't count.

**int mRow1, mCol1, mRow2, mCol2;**

The indices of the row and column CVs of surface 1 and surface 2 to fuse.

### Methods:

#### Prototype:

**NURBSFuseSurfaceCV();**

#### Remarks:

Constructor. The data members are initialized as follows:

**mSurf1 = mSurf2 = 0;**

**mRow1 = mCol1 = mRow2 = mCol2 = 0;**

## Class NURBSFuseCurveCV

See Also: [Class NURBSSurface](#), [Class NURBSSet](#), [Class NURBSControlVertex](#), [Class NURBSPoint](#), [Class NURBSCurve](#), [Class NURBSFuseSurfaceCV](#).

```
class NURBSFuseCurveCV
```

### Description:

This class is available in release 2.0 and later only.

This class may be used with a **NURBSSet** to fuse two CVs in a curve. This causes the CVs to reference one another so if you move one the other moves with it.

All methods of this class are implemented by the system.

### Data Members:

public:

**int mCurve1, mCurve2;**

The zero based indices of the curves to fuse. These may be the same value. Note that this is **not** the index in the **NURBSSet** of these items. Rather it is the index of CV curve in list of CV curves in the **NURBSSet**. For instance, if there were first two CV surfaces and then two CV curves in the **NURBSSet**, **mCurve1** would be 0 and **mCurve2** would be 1 since the CV surfaces don't count.

**int mCV1, mCV2;**

The indices of the CVs of curve1 and curve2 to fuse.

### Methods:

#### Prototype:

**NURBSFuseCurveCV();**

#### Remarks:

Constructor. The data members are initialized as follows:

**mCurve1 = mCurve2 = 0;**

**mCV1 = mCV2 = 0;**

## Class NURBSIndependentPoint

See Also: [Class NURBSPoint](#).

```
class NURBSIndependentPoint : public NURBSPoint
```

### Description:

This class is available in release 2.0 and later only.

This class is used to create an independent, free-standing point. There are methods to set the position of the point in various floating point formats and operators to compare points.

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
NURBSIndependentPoint();
```

#### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNPoint;
```

```
mX = mY = mZ = 0.0;
```

#### Prototype:

```
virtual ~NURBSIndependentPoint();
```

#### Remarks:

Destructor.

#### Prototype:

```
void SetPosition(TimeValue t, Point3 pt);
```

#### Remarks:

Sets the position of the point at the specified time (as a Point3).

#### Parameters:

```
TimeValue t
```

Specifies the time to set the position.

```
Point3 pt
```

The position to set.

**Prototype:**

**void SetPosition(TimeValue t, float x, float y, float z);**

**Remarks:**

Sets the position of the point at the specified time (as floats).

**Parameters:**

**TimeValue t**

Specifies the time to set the position.

**float x**

The X position to set.

**float y**

The Y position to set.

**float z**

The Z position to set.

**Prototype:**

**void SetPosition(TimeValue t, double x, double y, double z);**

**Remarks:**

Sets the position of the point at the specified time (as doubles).

**Parameters:**

**TimeValue t**

Specifies the time to set the position.

**double x**

The X position to set.

**double y**

The Y position to set.

**double z**

The Z position to set.

**Operators:**

**Prototype:**

**BOOL operator==(const NURBSIndependentPoint& pt);**

**Remarks:**

Equality operator. Compares if the X, Y and Z coordinates are equal.

**Parameters:**

**const NURBSIndependentPoint& pt**

The point to compare.

**Return Value:**

TRUE if equal; otherwise FALSE.

**Prototype:**

**BOOL operator!=(const NURBSIndependentPoint& pt);**

**Remarks:**

Inequality operator. Compares if the X, Y and Z coordinates are **not** equal.

**Parameters:**

**const NURBSIndependentPoint& pt**

The point to compare.

**Return Value:**

TRUE if any of the coordinates are **not** equal; otherwise FALSE.

**Prototype:**

**NURBSIndependentPoint & operator=(const  
NURBSIndependentPoint& pt);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSIndependentPoint& pt**

The point to assign.

## Class NURBTrimPoint

See Also: [List of NURBS Trim Directions](#).

### Description:

This class is available in release 2.0 and later only.

This class defines a point on a curve used to trim a portion of the curve from the point towards one of the ends of the curve. The trim point is defined by a parameter and a direction. For instance, consider a CV curve that exists in the parameter space from 0.0 to 1.0. If there is a trim point in the middle of the curve the parameter for the point would be 0.5. If the portion that was being trimmed off was from 0.5 to 1.0 then the trim direction would be positive.

All methods of this class are implemented by the system.

### Friend Classes:

friend class NURBSSet;

### Methods:

#### Prototype:

```
NURBTrimPoint(double parameter, NURBTrimDirection direction) : mParameter(parameter), mDirection(direction);
```

### Remarks:

Constructor.

### Parameters:

**double parameter**

This is point in parameter space at which the trim point exists.

**NURBTrimDirection direction**

The positive or negative direction in parameter space of the portion that is being trimmed off. The direction can also be set to 'none' so no trimming is done.

### Prototype:

```
double GetParameter();
```

### Remarks:

Returns the point in parameter space at which the trim point exists.

**Prototype:**

**NURBTrimDirection GetDirection();**

**Remarks:**

Returns the trim direction (positive, negative, or none) describing which portion of the curve is trimmed off.

## Class NURBSPointConstPoint

See Also: [Class NURBSObject](#), [List of NURBSConst Types](#), [Class Point3](#).

```
class NURBSPointConstPoint : public NURBSObject
```

### Description:

This class is available in release 2.0 and later only.

This class is used to create a dependent point that lies at a point or relative to it.

All methods of this class are implemented by the system.

### Data Members:

protected:

**NURBSId mParentId;**

The id of the parent object.

**int mParentIndex;**

The index in the NURBSSet of the parent object.

**NURBSConstType mCType;**

The type of constraint in use.

**Point3 mOffset;**

The offset from the object that the point is.

### Friend Classes:

```
friend class NURBSSet;
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

**Prototype:**

**NURBSPointConstPoint();**

**Remarks:**

Constructor. The data members are initialized as follows:

**mType = kNPointCPoint;**

**mpObject = NULL;**

**mpNSet = NULL;**

**mParentId = 0;**

**mParentIndex = -1;**

**mCType = kNConstOnObject;**

**mOffset = Point3(0,0,0);**

**Prototype:**

**~NURBSPointConstPoint();**

**Remarks:**

Destructor.

**Prototype:**

**void SetParent(int index);**

**Remarks:**

Sets the parent index to the specified value.

**Parameters:**

**int index**

The parent index to set.

**Prototype:**

**void SetParentId(NURBSId id);**

**Remarks:**

Sets the parent ID to the specified value.

**Parameters:**

**NURBSId id**

The parent ID to set.

**Prototype:****int GetParent();****Remarks:**

Returnst the parent index.

**Prototype:****NURBSId GetParentId();****Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Prototype:****void SetPointType(NURBSConstType type);****Remarks:**

Sets the type of point.

**Parameters:****NURBSConstType type**

Specifies the type of point to set. See [List of NURBSConst Types](#).

**Prototype:****NURBSConstType GetPointType();****Remarks:**

Returns the type of construction point this is. See [List of NURBSConst Types](#).

**Prototype:****void SetOffset(TimeValue t, Point3 pt);****Remarks:**

Sets the offset value at the specified time.

**Parameters:**

**TimeValue t**

Specifies the time at which the offset is set.

**Point3 pt**

The offset value to set. This value is specified in object coordinates.

**Prototype:**

**Point3 GetOffset(TimeValue t);**

**Remarks:**

Returns the offset value at the specified time in object coordinates.

**Parameters:**

**TimeValue t**

The time to retrieve the offset.

**Operators:**

**Prototype:**

**NURBSPointConstPoint & operator=(const  
NURBSPointConstPoint& pt);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSPointConstPoint& pt**

The construction point to assign.

## Class NURBSCurveConstPoint

See Also: [Class NURBSObject](#), [List of NURBSConst Types](#), [Class Point3](#).

```
class NURBSCurveConstPoint : public NURBSObject
```

### Description:

This class is available in release 2.0 and later only.

This class is used to create a dependent point that lies on a curve or relative to it.

The point can either be on the curve or off the curve. If it is on the curve, the **U Position** is the only control of its location. The **U Position** specifies a location along the curve (based on the curve's local U axis). There are three ways to displace the point's location relative to the U position:

**Offset** moves the point according to a relative (object space) X,Y,Z location.

**Normal** moves the point along the direction of the curve's normal. (Negative values move it opposite to the normal.)

U Position

**Tangent** moves the point along the tangent of the **U Position**.

All methods of this class are implemented by the system.

### Data Members:

protected:

**NURBSId mParentId;**

The NURBSId of the parent curve.

**int mParentIndex;**

The index in the **NURBSSet** of the parent curve.

**NURBSConstType mCType;**

The type of constraint in use.

**Point3 mOffset;**

The offset amount.

**float mNormal;**

The distance along the normal.

**float mUTangent;**

The distance along the tangent.

**double mUParam;**

Specifies the point along the parent curve.

**BOOL mTrimCurve;**

The point that is constrained to the curve may be used to trim the curve.

**BOOL mFlipTrim;**

If TRUE the curve is trimmed from the point towards low parameter space. If FALSE the curve is trimmed from the point towards high parameter space.

**Friend Classes:**

friend class NURBSSet;

**Methods:**

protected:

**Prototype:**

**void Clean(NURBSIdTab ids);**

**Remarks:**

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

**Parameters:**

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

**Prototype:**

**NURBSCurveConstPoint();**

**Remarks:**

Constructor. The data members are initialized as follows:

**mType = kNCurveCPoint;**

**mpObject = NULL;**

**mpNSet = NULL;**

**mParentId = 0;**

**mParentIndex = -1;**

**mCType = kNConstOnObject;**

```
mOffset = Point3(0,0,0);  
mNormal = 0.0f;  
mUTangent = 0.0f;  
mUParam = 0.0;  
mTrimCurve = FALSE;  
mFlipTrim = FALSE;
```

**Prototype:**

```
virtual ~NURBSCurveConstPoint();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the parent index to the specified value.

**Parameters:**

**int index**

The parent index to set.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the parent ID to the specified value.

**Parameters:**

**NURBSId id**

The parent ID to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the parent index.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Prototype:**

**void SetPointType(NURBSConstType type);**

**Remarks:**

Sets the point type.

**Parameters:**

**NURBSConstType type**

Specifies the construction point type. See [List of NURBSConst Types](#).

**Prototype:**

**NURBSConstType GetPointType();**

**Remarks:**

Returns the point type. See [List of NURBSConst Types](#).

**Prototype:**

**void SetOffset(TimeValue t, Point3 pt);**

**Remarks:**

Sets the offset value at the specified time.

**Parameters:**

**TimeValue t**

Specifies the time at which the offset value is set.

**Point3 pt**

The offset to set in object space.

**Prototype:**

### **Point3 GetOffset(TimeValue t);**

#### **Remarks:**

Returns the offset value in object space at the specified time.

#### **Parameters:**

##### **TimeValue t**

Specifies the time at which the offset value is returned.

#### **Prototype:**

**void SetUParam(TimeValue t, double param);**

#### **Remarks:**

Sets the U Parameter for the point.

#### **Parameters:**

##### **TimeValue t**

Specifies the time at which the value is set.

##### **double param**

The U parameter to set.

#### **Prototype:**

**double GetUParam(TimeValue t);**

#### **Remarks:**

Returns the U parameter at the specified time.

#### **Parameters:**

##### **TimeValue t**

Specifies the time at which the value is returned.

#### **Prototype:**

**void SetNormal(TimeValue t, float dist);**

#### **Remarks:**

Sets the normal distance at the specified time.

#### **Parameters:**

##### **TimeValue t**

Specifies the time at which the value is set.

**float dist**

The distance to set.

**Prototype:**

**float GetNormal(TimeValue t);**

**Remarks:**

Returns the distance along the normal at the specified time.

**Parameters:**

**TimeValue t**

Specifies the time at which the value is returned.

**Prototype:**

**void SetUTangent(TimeValue t, float dist);**

**Remarks:**

Set the U tangent value at the specified time.

**Parameters:**

**TimeValue t**

Specifies the time at which the value is set.

**float dist**

The distance to set.

**Prototype:**

**float GetUTangent(TimeValue t);**

**Remarks:**

Returns the U tangent value at the specified time.

**Parameters:**

**TimeValue t**

Specifies the time at which the value is returned.

**Prototype:**

**BOOL GetTrimCurve();**

**Remarks:**

Returns TRUE if this point is used to trim the curve and FALSE if it is not.

**Prototype:**

**void SetTrimCurve(BOOL trim);**

**Remarks:**

Sets the trim curve boolean.

**Parameters:**

**BOOL trim**

TRUE to indicate this point is used to trim the curve; FALSE to indicate the curve is not trimmed beyond the point.

**Prototype:**

**BOOL GetFlipTrim();**

**Remarks:**

Returns the state of the flip trim boolean.

**Prototype:**

**void SetFlipTrim(BOOL flip);**

**Remarks:**

Sets the state of the flip trim boolean.

**Parameters:**

**BOOL flip**

TRUE to indicate the curve is trimmed from the point towards low parameter space. Use FALSE to indicate the curve is trimmed from the point towards high parameter space.

**Operators:**

**Prototype:**

**NURBSCurveConstPoint & operator=(const  
NURBSCurveConstPoint& pt);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSCurveConstPoint& pt**

The point to assign.

## Class NURBSCurveCurveIntersectionPoint

See Also: [Class NURBSPoint](#).

```
class NURBSCurveCurveIntersectionPoint : public NURBSPoint
```

### Description:

This class is available in release 2.0 and later only.

This class is used to create a dependent point at the intersection of two curves.

All methods of this class are implemented by the system.

### Data Members:

protected:

**NURBSId mParentId[2];**

The ids of the two parent curves.

**int mParentIndex[2];**

The indices of the two parent curves in the **NURBSSet**.

**BOOL mTrimCurve[2];**

Indicates the point that is constrained to the curve may be used to trim the curve.

**BOOL mFlipTrim[2];**

If TRUE the curve is trimmed from the point towards low parameter space. If FALSE the curve is trimmed from the point towards high parameter space.

### Friend Classes:

```
friend class NURBSSet;
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

## **NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### **Prototype:**

**NURBSCurveCurveIntersectionPoint();**

### **Remarks:**

Constructor. The data members are initialized by value:

**mType = kNCurveCurveIntersectionPoint;**

**mpObject = NULL;**

**mpNSet = NULL;**

**mParentId[0] = mParentId[1] = 0;**

**mParentIndex[0] = mParentIndex[1] = -1;**

**mCurveParam[0] = mCurveParam[1] = 0.0;**

**mTrimCurve[0] = mTrimCurve[1] = FALSE;**

**mFlipTrim[0] = mFlipTrim[1] = FALSE;**

### **Prototype:**

**virtual ~NURBSCurveCurveIntersectionPoint();**

### **Remarks:**

Destructor.

### **Prototype:**

**double GetCurveParam(int curveNum);**

### **Remarks:**

Returns the point in parameter space of the specified curve of the point of intersection.

### **Parameters:**

**int curveNum**

The parent curve number: 0 or 1.

### **Prototype:**

**void SetParent(int pnum, int index);**

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

**void SetParentId(int pnum, NURBSId id);**

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified parent. Note that a **NURBSId** won't

be valid until the object has been instantiated in the scene

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**BOOL GetTrimCurve(int pnum);**

**Remarks:**

Returns TRUE if this point is used to trim the specified curve and FALSE if it is not.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**Prototype:**

**void SetTrimCurve(int pnum, BOOL trim);**

**Remarks:**

Sets the trim curve boolean for the specified curve.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**BOOL trim**

TRUE to indicate this point is used to trim the curve; FALSE to indicate the curve is not trimmed beyond the point.

**Prototype:**

**BOOL GetFlipTrim(int pnum);**

**Remarks:**

Returns the state of the flip trim boolean.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**Return Value:**

TRUE indicates the specified curve is trimmed from the point towards low parameter space. FALSE indicates the curve is trimmed from the point towards high parameter space.

**Prototype:**

```
void SetFlipTrim(int pnum, BOOL flip);
```

**Remarks:**

Sets the state of the flip trim boolean for the specified parent curve.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**BOOL flip**

TRUE to indicate the curve is trimmed from the point towards low parameter space. Use FALSE to indicate the curve is trimmed from the point towards high parameter space.

**Operators:****Prototype:**

```
NURBSCurveCurveIntersectionPoint & operator=(const  
NURBSCurveCurveIntersectionPoint &pt);
```

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSCurveCurveIntersectionPoint &pt**

The intersection point to assign.

## Class NURBSSurfConstPoint

See Also: [Class NURBSObject](#), [List of NURBSConst Types](#).

```
class NURBSSurfConstPoint : public NURBSObject
```

### Description:

This class is available in release 2.0 and later only.

This class is used to create a dependent point on a surface or related to it.

All methods of this class are implemented by the system.

### Data Members:

protected:

**NURBSId mParentId;**

The **NURBSId** of the parent surface.

**int mParentIndex;**

The index in the **NURBSSet** of the parent surface.

**NURBSConstType mCType;**

The type of constraint used by the point.

**Point3 mOffset;**

The offset amount.

**float mNormal;**

The distance along the normal.

**float mUTangent;**

The positive tangent offset in U. At the location in parameter space of the constrained point is a tangent to the surface. This is the distance along the positive U tangent in parameter space.

**double mUParam;**

The point is constrained to exist on the parent surface. The surface itself is defined over a parameter range. The point is defined at a point in the parent surface parameter space. This is the location of the point in the parent surface parameter space in U.

**float mVTangent;**

This is the distance along the positive V tangent in parameter space.

**double mVParam;**

This is the location of the point in the parent surface parameter space in V.

**Friend Classes:**

friend class NURBSSet;

**Methods:**

protected:

**Prototype:**

**void Clean(NURBSIdTab ids);**

**Remarks:**

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

**Parameters:**

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

**Prototype:**

**NURBSSurfConstPoint();**

**Remarks:**

Constructor. The data members are initialized as follows:

**mType = kNSurfaceCPoint;**

**mpObject = NULL;**

**mpNSet = NULL;**

**mParentId = 0;**

**mParentIndex = -1;**

**mCType = kNConstOnObject;**

**mOffset = Point3(0,0,0);**

**mNormal = 0.0f;**

**mUTangent = 0.0f;**

**mUParam = 0.0;**

**mVTangent = 0.0f;**

**mVParam = 0.0;**

**Prototype:**

**virtual ~NURBSSurfConstPoint();**

**Remarks:**

Destructor.

**Prototype:**

**void SetParent(int index);**

**Remarks:**

Sets the index in the **NURBSSet** of the parent object.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

**void SetParentId(NURBSId id);**

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent();**

**Remarks:**

Returns the index in the **NURBSSet** of the parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene

**Prototype:**

**void SetPointType(NURBSConstType type);**

**Remarks:**

Sets the type of constrained point this is. See [List of NURBSConst Types](#).

**Parameters:**

**NURBSConstType type**

The type of the constrained point.

**Prototype:**

**NURBSConstType GetPointType();**

**Remarks:**

Returns the type of constrained point this is. See [List of NURBSConst Types](#).

**Prototype:**

**void SetUParam();**

**Remarks:**

Sets the position of the point in the parent U parameter space.

**Parameters:**

**TimeValue t**

The time to set the U parameter.

**double param**

The value to set.

**Prototype:**

**double GetUParam(TimeValue t);**

**Remarks:**

Returns the position of the point in the parent U parameter space.

**Parameters:**

**TimeValue t**

The time to get the U parameter.

**Prototype:**

**void SetVParam(TimeValue t, double param);**

**Remarks:**

Sets the position of the point in the parent V parameter space.

**Parameters:**

**TimeValue t**

The time to set the U parameter.

**double param**

The value to set.

**Prototype:**

**double GetVParam(TimeValue t);**

**Remarks:**

Returns the position of the point in the parent V parameter space.

**Parameters:**

**TimeValue t**

The time to get the Vparameter.

**Prototype:**

**void SetOffset(TimeValue t, Point3 pt);**

**Remarks:**

Sets the offset of the point from the parent surface.

**Parameters:**

**TimeValue t**

The time at which to set the offset.

**Point3 pt**

The offset to set.

**Prototype:**

**Point3 GetOffset(TimeValue t);**

**Remarks:**

Returns the offset of the point from the parent surface at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the offset.

**Prototype:**

**void SetNormal(TimeValue t, float dist);**

**Remarks:**

Sets the distance along the normal to the surface of the point at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the distance.

**float dist**

The distance to set.

**Prototype:**

**float GetNormal(TimeValue t);**

**Remarks:**

Returns the distance along the normal to the surface of the point at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the distance.

**Prototype:**

**void SetUTangent(TimeValue t, float dist);**

**Remarks:**

Sets the distance along the U tangent of the point.

**Parameters:**

**TimeValue t**

The time at which to set the distance.

**float dist**

The distance to set.

**Prototype:**

**float GetUTangent(TimeValue t);**

**Remarks:**

Returns the distance along the U tangent of the point.

**Parameters:**

**TimeValue t**

The time at which to get the distance.

**Prototype:**

**void SetVTangent(TimeValue t, float dist);**

**Remarks:**

Sets the distance along the V tangent of the point.

**Parameters:**

**TimeValue t**

The time at which to set the distance.

**float dist**

The distance to set.

**Prototype:**

**float GetVTangent(TimeValue t);**

**Remarks:**

Returns the distance along the V tangent of the point.

**Parameters:**

**TimeValue t**

The time at which to get the distance.

**Operators:**

**Prototype:**

**NURBSSurfConstPoint & operator=(const**

**NURBSSurfConstPoint& pt);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSSurfConstPoint& pt**

The point to assign.

## Class NURBSPointCurve

See Also: [Class NURBSCurve](#), [Class NURBSIndependentPoint](#).

```
class NURBSPointCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a curve that uses points to describe its shape. All the points lie on the curve itself. There are methods to get/set the number of points in the curve, get/set the points themselves, refine the curve (add points without changing its shape), and to get/set the transformation matrix used by the curve. This matrix is used to set the position of the curve in the **NURBSSet**.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Data Members:

protected:

```
NURBSIndependentPoint *mpPts;
```

Array of independent points.

```
BOOL mClosed;
```

TRUE if the curve is closed; otherwise FALSE.

```
int mNumPts;
```

The number of independent points.

### Methods:

#### Prototype:

```
NURBSPointCurve();
```

#### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNPointCurve;
```

```
mClosed = FALSE;
```

```
mpPts = NULL;
```

**Prototype:**

**virtual ~NURBSPointCurve();**

**Remarks:**

Destructor. If any points were allocated they are freed and the cache is cleared.

**Prototype:**

**void Close();**

**Remarks:**

Closes the point curve.

**Prototype:**

**BOOL IsClosed();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the curve is closed; otherwise FALSE.

**Prototype:**

**void SetNumPts(int num);**

**Remarks:**

Sets the number of points in the point curve. Note that any previous point data is not maintained when the new number is set.

**Parameters:**

**int num**

The number of points in the curve.

**Prototype:**

**int GetNumPts();**

**Remarks:**

Returns the number of points in the curve.

**Prototype:**

**void GetNumPts(int &num);**

**Remarks:**

Retrieves the number of points in the curve.

**Parameters:**

**int &num**

The result is stored here.

**Prototype:**

**NURBSIndependentPoint\* GetPoint(int index);**

**Remarks:**

Returns a pointer to the specified point.

**Parameters:**

**int index**

The zero based index of the point to get.

**Prototype:**

**void SetPoint(int index, NURBSIndependentPoint &pt);**

**Remarks:**

Sets the specified point in the curve.

**Parameters:**

**int index**

The zero based index of the point to set.

**NURBSIndependentPoint &pt**

The point to set.

**Prototype:**

**void SetTransformMatrix(TimeValue t, SetXFormPacket& xPack);**

**Remarks:**

Sets the transformation matrix for the **NURBSPointCurve**. This controls the relative position of the item within a **NURBSSet**.

**Parameters:****TimeValue t**

The time at which to set the matrix.

**SetXFormPacket& xPack**

An instance of the **XFormPacket** class that describes the properties of the transformation. See [Class SetXFormPacket](#).

**Prototype:**

**Matrix3 GetTransformMatrix(TimeValue t);**

**Remarks:**

Returns the transformation matrix that controls the relative position of the point curve in the **NURBSSet**.

**Parameters:****TimeValue t**

The time at which to return the matrix.

**Prototype:**

**void Refine(TimeValue t, double u);**

**Remarks:**

This method adds a new point at the specified location on the curve without changing its shape. The point is specified as a distance in U parameter space. Note that calling this method causes all animation of the curve to be removed.

**Parameters:****TimeValue t**

The time at which to refine the curve. The curve may be animated, and thus the underlying parameter space may be changing. So when the **u** value is specified it must relate to the curve at a specific time.

**double u**

The point at which to refine the curve. Developers should use the method **NURBSCurve::GetParameterRange()** to get the valid range of values that may be passed here.

**Operators:**

**Prototype:**

**NURBSPointCurve & operator=(const NURBSPointCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSPointCurve& curve**

The curve to assign.

## Class NURBSCVCurve

See Also: [Class NURBSCurve](#), [Class NURBSControlVertex](#).

```
class NURBSCVCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a NURBS CV Curve. CV Curves have control vertices. The position of the control vertices (CVs) controls the shape of the curve. Unlike spline vertices, CVs don't necessarily lie on the curve they define. The CVs define a control lattice which surrounds the NURBS curve.

All methods of this class are implemented by the system.

### Data Members:

protected:

```
NURBSControlVertex *mpCVs;
```

Array of control vertices.

```
double *mpKnots;
```

Array of knots.

```
BOOL mClosed;
```

TRUE if the curve is closed; otherwise FALSE.

```
int mOrder;
```

The order of the curve.

```
int mNumKnots;
```

The number of knots.

```
int mNumCVs;
```

The number of control vertices.

```
NURBSAutoParam mAutoParam;
```

This data member is available in release 3.0 and later only.

Controls automatic reparameterization. See [List of NURBSAutoParam Types](#).

### Friend Classes:

```
friend class NURBSSet;
```

### Methods:

public:

**Prototype:**

```
NURBSCVCurve();
```

**Remarks:**

Constructor. The data members are initialized as follows:

```
mType = kNCVCurve;
```

```
mClosed = FALSE;
```

```
mpCVs = NULL;
```

```
mpKnots = NULL;
```

**Prototype:**

```
virtual ~NURBSCVCurve();
```

**Remarks:**

Destructor. If the knots and CV arrays are allocated they are freed and any caches are cleared.

**Prototype:**

```
void Close();
```

**Remarks:**

Closes the curve.

**Prototype:**

```
void SetOrder(int order);
```

**Remarks:**

Sets the order of the curve. This is one more than the degree of polynomial of any segment of the curve. All curves have a degree. The degree of a curve is the highest exponent in the equation used to represent it. A linear equation is degree 1, a quadratic equation degree 2. NURBS curves typically are represented by cubic equations and have a degree of 3.

**Parameters:**

```
int order
```

Specifies the order of the curve.

**Prototype:**

**int GetOrder();**

**Remarks:**

Returns the order of the curve.

**Prototype:**

**void SetNumKnots(int num);**

**Remarks:**

Sets the number of knots in the curve. Note that the previous knot data is NOT maintained

Because they are generated mathematically, NURBS curves have a parameter space in addition to the 3D geometric space in which they are displayed.

Specifically, an array of values called knots specifies the extent of influence of each control vertex (CV) on the curve or surface.

**Parameters:**

**int num**

Specifies the number of knots for the curve.

**Prototype:**

**int GetNumKnots();**

**Remarks:**

Returns the number of knots in the curve.

**Prototype:**

**void SetNumCVs(int num);**

**Remarks:**

Set the number of control vertices in the curve. Note that the previous control vertex data is NOT maintained.

**Parameters:**

**int num**

Specifies the number of control vertices.

**Prototype:**

**void GetNumCVs(int& num);**

**Remarks:**

Retrieves the number of control vertices.

**Parameters:**

**int& num**

The number is stored here.

**Prototype:**

**int GetNumCVs();**

**Remarks:**

Returns the number of control vertices.

**Prototype:**

**double GetKnot(int index);**

**Remarks:**

Returns the knot value whose index is passed.

**Parameters:**

**int index**

Specifies which knot value to return.

**Prototype:**

**void SetKnot(int index, double value);**

**Remarks:**

Sets the specified knot to the specified value.

**Parameters:**

**int index**

The 0 based index of the knot to set.

**double value**

Specifies the value to set.

**Prototype:**

**NURBSControlVertex \*GetCV(int index);**

**Remarks:**

Returns a pointer to the specified control vertex of the curve.

**Parameters:**

**int index**

The 0 based index of the control vertex to return.

**Prototype:**

**void SetCV(int index, NURBSControlVertex &cv);**

**Remarks:**

Sets the specified control vertex to the CV passed.

**Parameters:**

**int index**

The 0 based index of the control vertex to set.

**NURBSControlVertex &cv**

The CV to set.

**Prototype:**

**void SetTransformMatrix(TimeValue t, SetXFormPacket& xPack);**

**Remarks:**

Sets the transformation matrix for the **NURBSCVCurve**. This controls the relative position of the item within a **NURBSSet**.

**Parameters:**

**TimeValue t**

The time at which to set the matrix.

**SetXFormPacket& xPack**

An instance of the **XFormPacket** class that describes the properties of the transformation. See [Class SetXFormPacket](#).

**Prototype:**

**Matrix3 GetTransformMatrix(TimeValue t);**

**Remarks:**

Returns the transformation matrix of the **NURBSCVCurve** at the specified time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the matrix.

**Prototype:**

**NURBSAutoParam AutoParam();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the current settings for automatic reparameterization. See [List of NURBSAutoParam Types](#).

**Prototype:**

**void AutoParam(TimeValue t, NURBSAutoParam param);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the automatic reparameterization settings at the specified time.

**Parameters:**

**TimeValue t**

The time to set them.

**NURBSAutoParam param**

The settings to establish. See [List of NURBSAutoParam Types](#).

**Prototype:**

**void Reparameterize(TimeValue t, NURBSParamaterization param);**

**Remarks:**

This method is available in release 3.0 and later only.

This method reparameterizes this CV sub-object. This can be used to provide a better relation between control point locations and the shape of the sub-object.

**Parameters:**

**TimeValue t**

The time to reparameterize.

**NURBSParamaterization param**

The type of reparameterizing to perform. See [List of NURBSParamaterization Types](#).

**Prototype:**

**void EndsOverlap(BOOL& overlap);**

**Remarks:**

This method determines if the ends of the curve overlap even though the curve may not be closed (that is, the tangents match at the ends).

**Parameters:**

**BOOL& overlap**

The result is returned here: TRUE if the ends overlap; otherwise FALSE.

**Prototype:**

**void Refine(TimeValue t, double u);**

**Remarks:**

By specifying a parameter point on the curve, this method adds a new control vertex to the curve. It moves all the other CVs but maintains the shape of the curve. Note that this causes the loss of any animation on the curve.

**Parameters:**

**TimeValue t**

The time at which to refine the curve. If the curve is animated the **u** parameter below is time dependent.

**double u**

Specifies the distance along the curve to add the CV. See the base class method **GetParameterRange()** for the valid range of values for this parameter.

**Prototype:**

**void Insert(TimeValue t, double u);**

**Remarks:**

This method is available in release 2.5 and later only.

This method places a new CV along the current CV hull at the specified parameter point. This method leaves all the other CVs in place and changes the shape of the curve. This method preserves animation.

**Parameters:****TimeValue t**

The time at which to refine the curve. If the curve is animated the **u** parameter below is time dependent.

**double u**

Specifies the distance along the curve to add the CV. See the base class method **GetParameterRange()** for the valid range of values for this parameter.

**Operators:****Prototype:**

**NURBSCVCurve & operator=(const NURBSCVCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:****const NURBSCVCurve& curve**

The NURBSCVCurve to assign.

## Class NURBSBlendCurve

See Also: [Class NURBSCurve](#).

```
class NURBSBlendCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent blend curve. A blend curve connects the specified end of one curve to the specified end of another, blending the curvature of the parents to create a smooth curve between them. Methods are available to get/set the parent indices and ids, to get/set the ends of the curves used for the blend, and to get/set the tension values used.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSBlendCurve();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNBlendCurve;
```

```
mpObject = NULL;
mpNSet = NULL;
for (int i = 0; i < 2; i++) {
    mParentId[i] = 0;
    mParentIndex[i] = -1;
    mParentEnd[i] = TRUE;
    mTension[i] = 1.0;
}
```

**Prototype:**

```
~NURBSBlendCurve();
```

**Remarks:**

Destructor. The cache is cleared.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**int index**

The index into the **NURBSSet** of the parent object.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**NURBSId id**

The id to set.

**Prototype:****int GetParent(int pnum);****Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

**Parameters:****int pnum**

The parent number: 0 or 1.

**Prototype:****NURBSId GetParentId(int pnum);****Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene

**Parameters:****int pnum**

The parent number: 0 or 1.

**Prototype:****void SetEnd(int pnum, BOOL end);****Remarks:**

Sets if the beginning or end of the specified curve is used for the blend.

**Parameters:****int pnum**

The parent number: 0 or 1.

**BOOL end**

TRUE to use the end of the curve; FALSE to use the beginning. The beginning of the curve has lesser parameter values than the end.

**Prototype:**

**BOOL GetEnd(int pnum);**

**Remarks:**

Indicates if the beginning or end of the specified curve is used for the blend.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Return Value:**

TRUE if the end of the curve is used; FALSE if the beginning is used.

**Prototype:**

**void SetTension(TimeValue t, int pnum, double ten);**

**Remarks:**

Sets the tension value for the specified parent curve.

**Parameters:**

**TimeValue t**

The time at which to set the tension

**int pnum**

The parent number: 0 or 1.

**double ten**

The tension value to set.

**Prototype:**

**double GetTension(TimeValue t, int pnum);**

**Remarks:**

Returns the tension value for the specified parent curve.

**Parameters:**

**TimeValue t**

The time at which to get the tension

**int pnum**

The parent number: 0 or 1.

**Operators:**

**Prototype:**

**NURBSBlendCurve & operator=(const NURBSBlendCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSBlendCurve& curve**

The curve to assign.

## Class NURBSOffsetCurve

See Also: [Class NURBSCurve](#).

```
class NURBSOffsetCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent offset curve. An offset curve is offset from the original, parent curve. It lies in the same plane as its parent, and is normal to the original. Methods are available to get/set the parent index and id and to get/set the distance from the parent curve.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSOffsetCurve();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNOffsetCurve;
```

```
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mDistance = 0.0;
```

**Prototype:**

```
~NURBSOffsetCurve();
```

**Remarks:**

Destructor. The cache is cleared.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the parent object.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent curve.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene

**Prototype:**

**void SetDistance(TimeValue t, double d);**

**Remarks:**

Sets the distance of the offset curve from the original in 3ds max units at the specified time.

**Parameters:**

**TimeValue t**

The time to set the offset distance.

**double d**

The distance to set.

**Prototype:**

**double GetDistance(TimeValue t);**

**Remarks:**

Returns the distance of the offset curve from the parent at the specified time.

**Parameters:**

**TimeValue t**

The time to get the offset distance.

**Operators:**

**Prototype:**

**NURBSOffsetCurve & operator=(const NURBSOffsetCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSOffsetCurve& curve**

The curve to assign.

## Class NURBSXFormCurve

See Also: [Class NURBSCurve](#).

```
class NURBSXFormCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent transform (xform) curve. A transform curve is a copy of the original curve with a different position, rotation, or scale.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

```
protected:
```

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

```
NURBSIdTab ids
```

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

### Prototype:

```
NURBSXFormCurve();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNXFormCurve;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
```

```
mParentId = 0;  
mParentIndex = -1;  
mXForm.IdentityMatrix();
```

**Prototype:**

```
~NURBSXFormCurve();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent curve.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the **NURBSSet** of the parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene

**Prototype:**

**void SetXForm(TimeValue t, Matrix3& mat);**

**Remarks:**

Sets the transformation used to move/rotate/scale the curve from the original.

**Parameters:**

**TimeValue t**

The time to set the transformation.

**Matrix3& mat**

The transformation matrix to set.

**Prototype:**

**Matrix3& GetXForm(TimeValue t);**

**Remarks:**

Returns the transformation used to move/rotate/scale the curve from the original.

**Parameters:**

**TimeValue t**

The time to get the transformation.

**Operators:**

**Prototype:**

**NURBSXFormCurve & operator=(const NURBSXFormCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSXFormCurve& curve**

The curve to assign.

## Class NURBSMirrorCurve

See Also: [Class NURBSCurve](#), [List of NURBSMirrorAxis Types](#).

```
class NURBSMirrorCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent mirror curve. A mirror curve is similar to a mirror object that you create using the Mirror tool (on the 3ds max toolbar) or the Mirror modifier. It is the original curve reflected about one or two axes. Methods are available to get/set the parent index and id, to get/set the reflection axis, and to get/set the offset distance.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

```
NURBSIdTab ids
```

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSMirrorCurve();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNMirrorCurve;
```

```
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mXForm.IdentityMatrix();  
mAxis = kMirrorX;  
mDistance = 0.0;
```

**Prototype:**

```
~NURBSMirrorCurve();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the parent object.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent curve.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the **NURBSSet** of the parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene

**Prototype:**

**void SetAxis(NURBSMirrorAxis axis);**

**Remarks:**

Sets the axis or axes of reflection for the curve.

**Parameters:**

**NURBSMirrorAxis axis**

Specifies the axis or axes of reflection.

**Prototype:**

**NURBSMirrorAxis GetAxis();**

**Remarks:**

Returns the axis or axes of reflection for the curve.

**Prototype:**

**void SetXForm(TimeValue t, Matrix3& mat);**

**Remarks:**

This is an additional transformation applied to the axis specification. This corresponds to the gizmo they user may use interactively to alter the location of the mirror axis. This is exactly equivalent to setting the transform on the gizmo of a mirror modifier.

**Parameters:**

**TimeValue t**

The time at which to set the transformation.

## **Matrix3& mat**

The transformation to set.

### **Prototype:**

**Matrix3& GetXForm(TimeValue t);**

### **Remarks:**

Returns the additional transformation applied to the mirror axis at the specified time.

### **Parameters:**

**TimeValue t**

The time at which to get the transformation matrix.

### **Prototype:**

**void SetDistance(TimeValue t, double d);**

### **Remarks:**

Sets the offset distance of the curve.

### **Parameters:**

**TimeValue t**

The time at which to set the distance.

**double d**

The distance to set.

### **Prototype:**

**double GetDistance(TimeValue t);**

### **Remarks:**

Returns the offset distance of the curve.

### **Parameters:**

**TimeValue t**

The time at which to get the distance.

### **Operators:**

### **Prototype:**

**NURBSMirrorCurve & operator=(const NURBSMirrorCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSMirrorCurve& curve**

The curve to assign.

## Class NURBSFilletCurve

See Also: [Class NURBSCurve](#).

```
class NURBSFilletCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent fillet curve. A fillet is a curve that creates a circular arc corner between two parent curves. Methods are available to get/set the parent indices and ids, get/set the trim state of the curves, get/set which ends of the curves are used, and get/set the radius of the fillet.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSFilletCurve();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNFilletCurve;
```

```
mpObject = NULL;
mpNSet = NULL;
mRadius = 10.0;
for (int i = 0; i < 2; i++) {
    mParentId[i] = 0;
    mParentIndex[i] = -1;
    mParentEnd[i] = TRUE;
    mTrimCurve[i] = TRUE;
    mFlipTrim[i] = FALSE;
}
```

**Prototype:**

```
~NURBSFilletCurve();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**int index**

The index into the **NURBSSet** of the parent curve.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the NURBSSet of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**void SetEnd(int pnum, BOOL end);**

**Remarks:**

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**BOOL end**

TRUE to use the end of the curve; FALSE to use the beginning.

**Prototype:**

**BOOL GetEnd(int pnum);**

**Remarks:**

Returns which end of the specified curve is used for the fillet. TRUE if the end of the curve is used; FALSE if the beginning is used.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**Prototype:**

**void SetRadius(TimeValue t, double radius);**

**Remarks:**

Sets the radius for the fillet at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the radius value.

**double radius**

The radius to set.

**Prototype:**

**double GetRadius(TimeValue t);**

**Remarks:**

Returns the radius of the fillet at the specified time.

**Parameters:**

**TimeValue t**

The time to return the fillet value.

**Prototype:**

**void SetTrimCurve(int pnum, BOOL trim);**

**Remarks:**

Sets if the specified curve is trimmed beyond the fillet.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**BOOL trim**

TRUE to trim the curve beyond the fillet; otherwise FALSE.

**Prototype:**

**BOOL GetTrimCurve(int pnum);**

**Remarks:**

Determines if the specified curve is trimmed beyond the fillet. TRUE if the curve is trimmed; otherwise FALSE.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**Prototype:**

**BOOL GetFlipTrim(int pnum);**

**Remarks:**

Returns the flip state for the specified curve.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**Return Value:**

TRUE if flip is set; FALSE if not.

**Prototype:**

**void SetFlipTrim(int pnum, BOOL flip);**

**Remarks:**

Sets the flip state for the specified curve.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

## **BOOL flip**

TRUE to flip; FALSE to not flip.

### **Operators:**

### **Prototype:**

```
NURBSFilletCurve & operator=(const NURBSFilletCurve&  
curve);
```

### **Remarks:**

Assignment operator.

### **Parameters:**

```
const NURBSFilletCurve& curve
```

The curve to assign.

## Class NURBSChamferCurve

See Also: [Class NURBSCurve](#).

```
class NURBSChamferCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent chamfer curve. A chamfer is a curve that creates a straight line corner between two parent curves. Methods are available to get/set the parent indices and ids, to get/set which ends of the curves are used for the chamfer, get/set the trim settings for each curve, and to get/set the length of the curve back from the selected end that represents the start of the chamfer.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSChamferCurve();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNChamferCurve;
```

```
mpObject = NULL;
mpNSet = NULL;
for (int i = 0; i < 2; i++) {
    mParentId[i] = 0;
    mParentIndex[i] = -1;
    mParentEnd[i] = TRUE;
    mTrimCurve[i] = TRUE;
    mFlipTrim[i] = FALSE;
    mLength[i] = 0.0;
}
```

**Prototype:**

```
~NURBSChamferCurve();
```

**Remarks:**

Destructor. The cache is cleared.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**int index**

The index into the **NURBSSet** of the parent object.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**void SetEnd(int pnum, BOOL end);**

**Remarks:**

Sets which end of the specified curve is used for the chamfer.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**BOOL end**

TRUE to use the end of the curve; FALSE to use the beginning.

**Prototype:**

**BOOL GetEnd(int pnum);**

**Remarks:**

Returns which end of the specified curve is used for the chamfer. TRUE if the end of the curve is used; FALSE if the beginning is used.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**Prototype:**

**void SetLength(TimeValue t, int pnum, double length);**

**Remarks:**

Sets the length for the specified parent curve back from the end that defines the beginning of the chamfer, at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the chamfer length.

**int pnum**

The parent curve number: 0 or 1.

**double length**

The chamfer length to set.

**Prototype:**

**double GetLength(TimeValue t, int pnum);**

**Remarks:**

Returns the length of the chamfer at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the chamfer length.

**int pnum**

The parent curve number: 0 or 1.

**Prototype:**

**void SetTrimCurve(int pnum, BOOL trim);**

**Remarks:**

Sets if the specified curve is trimmed beyond the chamfer.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**BOOL trim**

TRUE to trim the curve beyond the chamfer; otherwise FALSE.

**Prototype:**

**BOOL GetTrimCurve(int pnum);**

**Remarks:**

Determines if the specified curve is trimmed beyond the fillet. TRUE if the curve is trimmed; otherwise FALSE.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**Prototype:**

**BOOL GetFlipTrim(int pnum);**

**Remarks:**

Returns the flip state for the specified curve.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**Return Value:**

TRUE if flip is set; FALSE it not.

**Prototype:**

**void SetFlipTrim(int pnum, BOOL flip);**

**Remarks:**

Sets the flip state for the specified curve.

**Parameters:**

**int pnum**

The parent curve number: 0 or 1.

**BOOL flip**

TRUE to flip; FALSE to not flip.

**Operators:**

**Prototype:**

**NURBSChamferCurve & operator=(const  
NURBSChamferCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSChamferCurve& curve**

The curve to assign.

## Class NURBSIsoCurve

See Also: [Class NURBSCurve](#).

```
class NURBSIsoCurve : public NURBSCurve
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent iso curve. U and V iso curves are dependent curves created along lines of constant parameter value of a NURBS surface. Note the difference between "Iso Lines", which are a display artifact, and "Iso Curves" which are the dependent objects. There are methods available to get/set the parent surface index and id, get/set the direction of the iso curve, and get/set the parameter which determines the location on the surface the curve matches.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSIsoCurve();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNIsoCurve;  
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mIsU = TRUE;  
mParam = 0.0;  
mTrim = FALSE;  
mFlipTrim = FALSE;  
mSeed = Point2(0.0, 0.0);
```

**Prototype:**

```
~NURBSIsoCurve();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent();**

**Remarks:**

Returns the index in the **NURBSSet** of the parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Prototype:**

**void SetDirection(BOOL isU);**

**Remarks:**

Sets the direction of the iso curve, either U or V.

**Parameters:**

**BOOL isU**

TRUE for U; FALSE for V.

**Prototype:**

**BOOL GetDirection();**

**Remarks:**

Returns TRUE if the iso curve is in the U direction; FALSE for the V direction.

**Prototype:**

**void SetParam(TimeValue t, double p);**

**Remarks:**

Sets the parameter which determines where on the surface the iso curve is defined.

**Parameters:**

**TimeValue t**

The time at which to evaluate the surface.

**double p**

The parameter defining the location on the surface.

**Prototype:**

**double GetParam(TimeValue t);**

**Remarks:**

Returns the parameter which determines where on the surface the iso curve is defined.

**Parameters:**

**TimeValue t**

The time at which to evaluate the surface.

**Prototype:**

**BOOL GetTrim();**

**Remarks:**

Returns the state of the trim flag. When set the surface is trimmed against the curve. When not set, the surface isn't trimmed

**Prototype:**

**void SetTrim(BOOL trim);**

**Remarks:**

Sets the state of the trim flag. When set the surface is trimmed against the curve. When not set, the surface isn't trimmed

**Parameters:**

**BOOL trim**

TRUE to trim; FALSE to not trim.

**Prototype:**

**BOOL GetFlipTrim();**

**Remarks:**

Returns the state of the trim flip flag. When set this trims the surface in the

opposite direction

**Prototype:**

**void SetFlipTrim(BOOL flip);**

**Remarks:**

Sets the state of the trim flip flag. When set this trims the surface in the opposite direction

**Parameters:**

**BOOL flip**

TRUE to flip; FALSE to not flip.

**Prototype:**

**Point2 GetSeed();**

**Remarks:**

Returns the UV location of the seed value on the curve.

**Prototype:**

**void SetSeed(Point2& seed);**

**Remarks:**

Sets the UV location of the seed value on the curve.

**Parameters:**

**Point2& seed**

The seed value to set.

**Operators:**

**Prototype:**

**NURBSIsoCurve & operator=(const NURBSIsoCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSIsoCurve& curve**

The curve to assign.

## Class NURBSPointSurface

See Also: [Class NURBSSurface](#), [Class NURBSIndependentPoint](#).

```
class NURBSPointSurface : public NURBSSurface
```

### Description:

This class is available in release 2.0 and later only.

This class defines a surface that uses points to describe its shape. This class has methods to close the surface in U and V, set the number of points in U and V, and get/set the points in U and V. There is also a method to add additional points to the surface. The point surface has a transformation matrix used to set the relative position of the surface in a **NURBSSet**.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

#### Prototype:

```
NURBSPointSurface();
```

#### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNPointSurface;
```

```
mClosedInU = FALSE;
```

```
mClosedInV = FALSE;
```

```
mpPts = NULL;
```

#### Prototype:

```
virtual ~NURBSPointSurface();
```

#### Remarks:

Destructor. Any allocated points are deleted.

#### Prototype:

```
void CloseInU();
```

**Remarks:**

This method closes the surface in the U direction.

**Prototype:**

```
void CloseInV();
```

**Remarks:**

This method closes the surface in the V direction.

**Prototype:**

```
void SetNumPts(int u, int v);
```

**Remarks:**

Sets the number of points in the surface in the U and V directions. Any previously allocated points are not maintained when the new number is set.

**Parameters:**

**int u**

The number of points in U.

**int v**

The number of points in V.

**Prototype:**

```
int GetNumUPts();
```

**Remarks:**

Returns the number of points in the U direction.

**Prototype:**

```
int GetNumVPts();
```

**Remarks:**

Returns the number of points in the V direction.

**Prototype:**

```
void GetNumPts(int &u, int &v);
```

**Remarks:**

Retrieves the number of points in both the U and V directions.

**Parameters:**

**int &u**

The number in U is stored here.

**int &v**

The number in V is stored here.

**Prototype:**

**NURBSIndependentPoint\* GetPoint(int u, int v);**

**Remarks:**

Returns a pointer to the specified point in the surface.

**Parameters:**

**int u**

The zero based index of the point in the U direction.

**int v**

The zero based index of the point in the V direction.

**Prototype:**

**void SetPoint(int u, int v, NURBSIndependentPoint& pt);**

**Remarks:**

Sets the specified point in the surface to the point passed.

**Parameters:**

**int u**

The zero based index of the point in the U direction.

**int v**

The zero based index of the point in the V direction.

**NURBSIndependentPoint& pt**

The point to set.

**Prototype:**

**void SetTransformMatrix(TimeValue t, SetXFormPacket& mat);**

**Remarks:**

Sets the transformation matrix for the **NURBSPointSurface**. This controls the relative position of the surface within a **NURBSSet**.

**Parameters:**

**TimeValue t**

The time at which to set the matrix.

**SetXFormPacket& xPack**

An instance of the **XFormPacket** class that describes the properties of the transformation. See [Class SetXFormPacket](#).

**Prototype:**

**Matrix3 GetTransformMatrix(TimeValue t);**

**Remarks:**

Returns the transformation matrix of the **NURBSPointSurface** at the specified time.

**Parameters:**

**TimeValue t**

The time at which to retrieve the matrix.

**Prototype:**

**void Refine(TimeValue t, double u, double v, int U\_V\_Both);**

**Remarks:**

This method adds a new point at the specified location on the surface without changing the shape of the surface. The location may be specified as a U value or a V value, or both.

If you refine in U (U\_V\_Both = 0) you must specify v

If you refine in V (U\_V\_Both = 1) you must specify u

If you refine in U and V (U\_V\_Both = -1) you must specify u and v

**Parameters:**

**TimeValue t**

The time at which to refine the surface.

**double u**

The location for the point in U space (range 0.0 to 1.0).

**double v**

The location for the point in V space (range 0.0 to 1.0).

**int U\_V\_Both**

This value must be **0**, **1** or **-1**.

If 0 the refinement is done in u (and v is specified).

If 1 the refinement is done in v (and u is specified).

If -1 the refinement is done in both u and v (and both u and v must be specified).

**Operators:****Prototype:**

```
NURBSPointSurface & operator=(const NURBSPointSurface&  
surf);
```

**Remarks:**

Assignment operator.

**Parameters:**

```
const NURBSPointSurface& surf
```

The surface to assign.

## Class NURBSCVSurface

See Also: [Class NURBSSurface](#).

```
class NURBSCVSurface : public NURBSSurface
```

### Description:

This class is available in release 2.0 and later only.

This class defines a surface that uses control vertices (CVs) to describe its shape. The CVs define a control lattice which surrounds the surface. This class has methods to close the surface in U and V, set its order in U and V, set the number of knots and CVs in U and V, and get/set the knots and CVs in U and V. There is also a method to add additional CVs to the surface. The CV surface has a transformation matrix used to position the surface within a NURBSSet.

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
NURBSCVSurface();
```

#### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNCVSurface;
```

```
mRigid = FALSE;
```

```
mClosedInU = FALSE;
```

```
mClosedInV = FALSE;
```

```
mpCVs = NULL;
```

```
mpUKnots = NULL;
```

```
mpVKnots = NULL;
```

```
mNumUCVs = 0;
```

```
mNumVCVs = 0;
```

```
mNumUKnots = 0;
```

```
mNumVKnots = 0;
```

```
mUOrder = 0;
```

```
mVOrder = 0;
```

**mAutoParam = kNotAutomatic;**

**Prototype:**

**virtual ~NURBSCVSurface();**

**Remarks:**

Destructor.

**Prototype:**

**BOOL IsRigid();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns TRUE if the surface is 'rigid'; otherwise FALSE.

**Prototype:**

**void SetRigid(BOOL isRigid);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets the 'rigid' state of the surface.

**Parameters:**

**BOOL isRigid**

TRUE for on; FALSE for off.

**Prototype:**

**NURBSAutoParam AutoParam();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the automatic parameterization setting. See [List of NURBSAutoParam Types](#).

**Prototype:**

**void AutoParam(TimeValue t, NURBSAutoParam param);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the automatic parameterization setting at the specified time.

**Parameters:****TimeValue t**

The time at which to set the parameterization.

**NURBSAutoParam param**

See [List of NURBSAutoParam Types](#).

**Prototype:**

```
void Reparameterize(TimeValue t, NURBSParamaterization  
param);
```

**Remarks:**

This method is available in release 3.0 and later only.

Sets the reparameterization type at the specified time.

**Parameters:****TimeValue t**

The time at which to sets the reparameterization type.

**NURBSParamaterization param**

See [List of NURBSParamaterization Types](#).

**Prototype:**

```
void CloseInU();
```

**Remarks:**

This method closes the surface in the U direction. The aligns the surface edge to edge in U and sets the tangents to match.

**Prototype:**

```
void CloseInV();
```

**Remarks:**

This method closes the surface in the V direction. The aligns the surface edge to edge in V and sets the tangents to match.

**Prototype:**

**void SetUOrder(int order);**

**Remarks:**

Sets the order of the surface in the U direction.

**Parameters:**

**int order**

Specifies the order of the surface in the U direction.

**Prototype:**

**int GetUOrder();**

**Remarks:**

Returns the order of the surface in the U direction.

**Prototype:**

**int GetVOrder();**

**Remarks:**

Returns the order of the surface in the V direction.

**Prototype:**

**void SetVOrder(int order);**

**Remarks:**

Sets the order of the surface in the V direction.

**Parameters:**

**int order**

Specifies the order of the surface in the V direction.

**Prototype:**

**void SetNumUKnots(int num);**

**Remarks:**

Sets the number of knots in the U direction. Note that the knot data is not maintained.

**Parameters:**

**int num**

Specifies the number of knots in the U direction.

**Prototype:**

**void SetNumVKnots(int num);**

**Remarks:**

Sets the number of knots in the V direction. Note that the knot data is not maintained.

**Parameters:**

**int num**

Specifies the number of knots in the V direction.

**Prototype:**

**int GetNumUKnots();**

**Remarks:**

Returns the number of knots in the U direction.

**Prototype:**

**int GetNumVKnots();**

**Remarks:**

Returns the number of knots in the V direction.

**Prototype:**

**void SetNumCVs(int u, int v);**

**Remarks:**

Sets the number of control vertices in both the U and V directions. Note that the CV data is not maintained.

**Parameters:**

**int u**

Specifies the number of control vertices in the U direction.

**int v**

Specifies the number of control vertices in the V direction.

**Prototype:**

**int GetNumUCVs();**

**Remarks:**

Returns the number of control vertices in the U direction.

**Prototype:**

**int GetNumVCVs();**

**Remarks:**

Returns the number of control vertices in the V direction.

**Prototype:**

**void GetNumCVs(int &u, int &v);**

**Remarks:**

Returns the number of control vertices in both the U and V directions.

**Parameters:**

**int &u**

The number of CVs in the U direction is returned here.

**int &v**

The number of CVs in the V direction is returned here.

**Prototype:**

**double GetUKnot(int index);**

**Remarks:**

Returns the specified knot value in the U direction.

**Parameters:**

**int index**

The 0 based index of the knot value to return.

**Prototype:**

**double GetVKnot(int index);**

**Remarks:**

Returns the specified knot value in the V direction.

**Parameters:**

**int index**

The 0 based index of the knot value to return.

**Prototype:**

**void SetUKnot(int index, double value);**

**Remarks:**

Sets the specified knot in the U direction to the specified value.

**Parameters:**

**int index**

The 0 based index of the knot value to set.

**double value**

The value to set.

**Prototype:**

**void SetVKnot(int index, double value);**

**Remarks:**

Sets the specified knot in the U direction to the specified value.

**Parameters:**

**int index**

The 0 based index of the knot value to set.

**double value**

The value to set.

**Prototype:**

**NURBSControlVertex &GetCV(int u, int v);**

**Remarks:**

Returns the specified control vertex of this surface.

**Parameters:**

**int u**

The 0 based index in the U direction.

**int v**

The 0 based index in the V direction.

**Prototype:**

**void SetCV(int u, int v, NURBSControlVertex &cv);**

**Remarks:**

Sets the specified control vertex.

**Parameters:**

**int u**

The 0 based index in the U direction.

**int v**

The 0 based index in the V direction.

**NURBSControlVertex &cv**

The control vertex to set.

**Prototype:**

**void SetTransformMatrix(TimeValue t, SetXFormPacket& mat);**

**Remarks:**

Sets the transformation matrix for the **NURBSCVSurface**. This matrix controls the relative position of the surface within a **NURBSSet**.

**Parameters:**

**TimeValue t**

The time at which to set the matrix.

**SetXFormPacket& xPack**

An instance of the **XFormPacket** class that describes the properties of the transformation (specifically if it's being moved, rotated, or scaled). See [Class SetXFormPacket](#).

**Prototype:**

## **Matrix3 GetTransformMatrix(TimeValue t);**

### **Remarks:**

Returns the transformation matrix of the **NURBSCVSurface** at the specified time.

### **Parameters:**

#### **TimeValue t**

The time at which to retrieve the matrix.

### **Prototype:**

**void EdgesOverlap(BOOL& uOverlap, BOOL& vOverlap);**

### **Remarks:**

This method determines if the edges of the surface overlap in U and/or V even though the surface may not be closed (that is, the tangents match at the edges).

### **Parameters:**

#### **BOOL& uOverlap**

The U result is returned here: TRUE if the edges overlap in U; otherwise FALSE.

#### **BOOL& vOverlap**

The V result is returned here: TRUE if the edges overlap in V; otherwise FALSE.

### **Prototype:**

**void Refine(TimeValue t, double u, double v, int U\_V\_Both);**

### **Remarks:**

This method adds a control vertex at the specified point on the surface without changing the shape of the surface. The other CV points will move to maintain the current shape. The point may be specified as a U value or a V value, or both.

If you refine in U (U\_V\_Both = 0) you must specify v

If you refine in V (U\_V\_Both = 1) you must specify u

If you refine in U and V (U\_V\_Both = -1) you must specify u and v

### **Parameters:**

**TimeValue t**

The time at which to refine the surface.

**double u**

The position for the point in U space.

**double v**

The position for the point in V space.

**int U\_V\_Both**

This value must be **0**, **1** or **-1**.

If 0 the refinement is done in u (and v is specified).

If 1 the refinement is done in v (and u is specified).

If -1 the refinement is done in both u and v (and both u and v must be specified).

**Prototype:**

```
void Insert(TimeValue t, double u, double v, int U_V_Both);
```

**Remarks:**

This method is available in release 2.5 and later only.

This method adds a new CV to the surface and changes its shape. The other CVs will not move as they do in **Refine()**. This method preserves any existing animation of the surface.

**Parameters:****TimeValue t**

The time at which to refine the surface.

**double u**

The position for the point in U space.

**double v**

The position for the point in V space.

**int U\_V\_Both**

This value must be **0**, **1** or **-1**.

If 0 the refinement is done in u (and v is specified).

If 1 the refinement is done in v (and u is specified).

If -1 the refinement is done in both u and v (and both u and v must be specified).

**Operators:**

**Prototype:**

**NURBSCVSurface & operator=(const NURBSCVSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSCVSurface& surf**

The CV surface to assign.

# Class NURBSBlendSurface

See Also: [Class NURBSSurface](#).

```
class NURBSBlendSurface : public NURBSSurface
```

## Description:

This class is available in release 2.0 and later only.

This class defines a dependent blend surface. A blend surface connects the edge of one surface to the edge of another, blending the curvature of the parents to create a smooth surface between them. Methods are available to get/set the parents, parent Ids, tension parameters and surface normal matching state. All methods of this class are implemented by the system.

## Friend Classes:

```
friend class NURBSSet
```

## Methods:

```
protected:
```

## Prototype:

```
void Clean(NURBSIdTab ids);
```

## Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

## Parameters:

```
NURBSIdTab ids
```

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

## Prototype:

```
NURBSBlendSurface();
```

## Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNBlendSurface;
```

```
mpObject = NULL;
mpNSet = NULL;
for (int i = 0; i < 2; i++) {
    mParentId[i] = 0;
    mParentIndex[i] = -1;
    mParentEdge[i] = 0;
    mFlip[i] = FALSE;
    mTension[i] = 1.0;
    mCurveStartParam[i] = 0.0;
}
```

**Prototype:**

```
virtual ~NURBSBlendSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**void SetEdge(int pnum, int edge);**

**Remarks:**

Sets which edge of the specified surface is used for the blend.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**int edge**

One of the following values:

**0:** The low U edge.

**1:** The high U edge.

**2:** The low V edge.

**3:** The high V edge.

**Prototype:**

**int GetEdge(int pnum);**

**Remarks:**

Returns an integer that determines which edge of the specified surface is used for the blend.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**Return Value:**

One of the following values:

**0:** The low U edge.

**1:** The high U edge.

**2:** The low V edge.

**3:** The high V edge.

**Prototype:**

**void SetTension(TimeValue t, int pnum, double ten);**

**Remarks:**

Sets the tension value for the specified parent surface.

**Parameters:**

**TimeValue t**

The time at which to set the tension value.

**int pnum**

The parent number: 0 or 1.

**double ten**

The tension value to set.

**Prototype:**

**double GetTension(TimeValue t, int pnum);**

**Remarks:**

Returns the tension value for the specified parent surface.

**Parameters:**

**TimeValue t**

The time at which to return the tension value.

**int pnum**

The parent number: 0 or 1.

**Prototype:**

**void SetFlip(int pnum, BOOL flip);**

**Remarks:**

This allows one to control the matching of parent surface normals when creating the blend surface. For example, normally when you create a blend surface between two parent surfaces you don't want a 'bow tie' surface (one with the ends rotated 180 degrees so it crosses on itself in the middle). If you simply match the parent normals you'll occasionally get a 'bow tie' surface. To prevent this you use this method to set a state indicating that one or the other should be flipped before it's used. In this way, when the blend is created, a 'bow tie' won't occur.

**Parameters:**

**int pnum**

The number of the parent surface: 0 or 1.

**BOOL flip**

TRUE to match the parent surface normal; FALSE to not match it.

**Prototype:**

**BOOL GetFlip(int pnum);**

**Remarks:**

Returns the flip state of the specified parent surface.

**Parameters:**

**int pnum**

The number of the parent surface: 0 or 1.

**Prototype:**

**void SetCurveStartPoint(TimeValue t, int pnum, double startpoint);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the start point for the specified parent curve. Note: This is only applicable if the parent is a closed curve.

**Parameters:**

**TimeValue t**

The time at which to set the start point.

**int pnum**

The number of the parent surface: 0 or 1.

**double startpoint**

The start point in the range 0.0 to 1.0.

**Prototype:**

**double GetCurveStartPoint(TimeValue t, int pnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the start point of the specified parent curve.

Note: This is only applicable if the parent is a closed curve.

**Parameters:**

**TimeValue t**

The time at which to get the start point.

**int pnum**

The number of the parent surface: 0 or 1.

**Operators:**

**Prototype:**

**NURBSBlendSurface & operator=(const NURBSBlendSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSBlendSurface& surf**

The surface to assign.

# Class NURBSOffsetSurface

See Also: [Class NURBSSurface](#).

```
class NURBSOffsetSurface : public NURBSSurface
```

## Description:

This class is available in release 2.0 and later only.

This class defines a dependent offset surface. An Offset surface is offset a specified distance from the original along the parent surface's normals. Methods are available to get/set the parent index and parent Ids, and to get/set the offset distance of the surface.

All methods of this class are implemented by the system.

## Friend Classes:

```
friend class NURBSSet
```

## Methods:

protected:

## Prototype:

```
void Clean(NURBSIdTab ids);
```

## Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

## Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

## Prototype:

```
NURBSOffsetSurface();
```

## Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNOffsetSurface;
```

```
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mDistance = 0.0;
```

**Prototype:**

```
virtual ~NURBSOffsetSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the Id of the parent surface.

**Parameters:**

**NURBSId id**

The Id to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the **NURBSSet** of the parent surface.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent surface. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Prototype:**

**void SetDistance(TimeValue t, double d);**

**Remarks:**

Sets the offset distance of the surface at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the offset a value.

**double d**

A distance of the offset in 3ds max units.

**Prototype:**

**double GetDistance(TimeValue t);**

**Remarks:**

Returns the distance of the offset at the specified time.

**Parameters:**

**TimeValue t**

The time at which to return the offset a value.

**Operators:**

**Prototype:**

**NURBSOffsetSurface & operator=(const NURBSOffsetSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSOffsetSurface& surf**

The surface to assign.

# Class NURBSXFormSurface

See Also: [Class NURBSSurface](#).

```
class NURBSXFormSurface : public NURBSSurface
```

## Description:

This class is available in release 2.0 and later only.

This class defines a dependent transform (xform) surface. A transform surface is a copy of the original surface with a different position, rotation, or scale.

Methods are available to get/set the indices of the parent surface in the **NURBSSet** and the parent Ids and to specify/retrieve the transformation used on the surface.

All methods of this class are implemented by the system.

## Friend Classes:

```
friend class NURBSSet
```

## Methods:

protected:

## Prototype:

```
void Clean(NURBSIdTab ids);
```

## Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

## Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

## Prototype:

```
NURBSXFormSurface();
```

## Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNXFormSurface;
```

```
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mXForm.IdentityMatrix();
```

**Prototype:**

```
virtual ~NURBSXFormSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the NURBSSet of the parent object.

**Parameters:**

**int index**

The index in the NURBSSet of the parent object.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Set the NURBSId of the parent object.

**Parameters:**

**NURBSId id**

The Id to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the NURBSSet of the parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the NURBSId of the parent object.

**Prototype:**

**void SetXForm(TimeValue t, Matrix3& mat);**

**Remarks:**

Sets the transformation from the parent surface.

**Parameters:**

**TimeValue t**

The time at which to set the transformation.

**Matrix3& mat**

The transformation to set.

**Prototype:**

**Matrix3& GetXForm(TimeValue t);**

**Remarks:**

Returns the transformation from the parent surface.

**Parameters:**

**TimeValue t**

The time at which the transformation is returned.

**Operators:**

**Prototype:**

**NURBSXFormSurface & operator=(const  
NURBSXFormSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSXFormSurface& surf**

The surface to assign.

# Class NURBSMirrorSurface

See Also: [Class NURBSSurface](#), [List of NURBSMirrorAxis Types](#).

```
class NURBSMirrorSurface : public NURBSSurface
```

## Description:

This class is available in release 2.0 and later only.

This class defines a dependent mirror surface. A mirror surface is similar to a mirror object that you create using the Mirror tool (on the 3ds max toolbar) or the Mirror modifier. It is the original surface reflected about one or two axes. Methods are available to get/set the indices of the parent surface in the **NURBSSet** and the parent Ids, to get/set the mirror axes, to get/set the mirror distance, and to set the transformation used to position the surface in the **NURBSSet**.

All methods of this class are implemented by the system.

## Friend Classes:

```
friend class NURBSSet
```

## Methods:

protected:

## Prototype:

```
void Clean(NURBSIdTab ids);
```

## Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

## Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

## Prototype:

```
NURBSMirrorSurface();
```

## Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNMirrorSurface;  
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mXForm.IdentityMatrix();  
mAxis = kMirrorX;  
mDistance = 0.0;
```

**Prototype:**

```
virtual ~NURBSMirrorSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the parent object.

**Parameters:**

**int index**

The index in the **NURBSSet** of the parent object

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the parent object.

**Parameters:**

**NURBSId id**

The **NURBSId** of the parent object.

**Prototype:**

**int GetParent();**

**Remarks:**

Returns the index in the **NURBSSet** of the parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene.

**Prototype:**

**void SetAxis(NURBSMirrorAxis axis);**

**Remarks:**

Sets the mirror axis to the specified constant.

**Parameters:**

**NURBSMirrorAxis axis**

The mirror axis to set.

**Prototype:**

**NURBSMirrorAxis GetAxis();**

**Remarks:**

Returns the axis or axes of reflection for the surface.

**Prototype:**

**void SetXForm(TimeValue t, Matrix3& mat);**

**Remarks:**

This is an additional transformation applied to the axis specification. This corresponds to the gizmo the user may use interactively to alter the location of the mirror axis. This is exactly equivalent to setting the transform on the gizmo of a mirror modifier.

**Parameters:**

**TimeValue t**

The time at which to set the transformation.

**Matrix3& mat**

The transformation to set.

**Prototype:**

**Matrix3& GetXForm(TimeValue t);**

**Remarks:**

Returns the additional transformation applied to the mirror axis at the specified time.

**Parameters:****TimeValue t**

The time at which to get the transformation matrix.

**Prototype:**

**void SetDistance(TimeValue t, double d);**

**Remarks:**

This is just like the offset parameter in the mirror modifier. It is an offset from the center of the local coordinate system for the mirror object that moves the mirror, in the direction specified by the mirror axis.

**Parameters:****TimeValue t**

The time at which to set the offset.

**double d**

The offset distance.

**Prototype:**

**double GetDistance(TimeValue t);**

**Remarks:**

Returns the offset distance of the mirror at the specified time.

**Parameters:****TimeValue t**

The time at which to return the offset.

**Operators:**

**Prototype:**

**NURBSMirrorSurface & operator=(const  
NURBSMirrorSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSMirrorSurface& surf**  
The surface to assign.

## Class NURBSRuledSurface

See Also: [Class NURBSSurface](#).

```
class NURBSRuledSurface : public NURBSSurface
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent ruled surface. A ruled surface is generated from two curve sub-objects. It lets you use curves to design the two opposite borders of a surface. Methods are available to get/set the indices of the parent surface in the **NURBSSet** and the parent Ids and to set the surface normal matching state of the surfaces.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

```
protected:
```

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

```
NURBSIdTab ids
```

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

### Prototype:

```
NURBSRuledSurface();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNRuledSurface;
```

```
mpObject = NULL;  
mpNSet = NULL;  
for (int i = 0; i < 2; i++) {  
  mParentId[i] = 0;  
  mParentIndex[i] = -1;  
  mFlip[i] = FALSE;  
  mCurveStartParam[i] = 0.0;  
}
```

**Prototype:**

```
virtual ~NURBSRuledSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

**int index**

The index in the **NURBSSet** of the specified parent object.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**int pnum**

The parent number: 0 or 1.

## **NURBSId id**

The id to set.

### **Prototype:**

**int GetParent(int pnum);**

### **Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

### **Parameters:**

**int pnum**

The parent number: 0 or 1.

### **Prototype:**

**NURBSId GetParentId(int pnum);**

### **Remarks:**

Returns the **NURBSId** of the specified parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene

### **Parameters:**

**int pnum**

The parent number: 0 or 1.

### **Prototype:**

**void SetFlip(int pnum, BOOL flip);**

### **Remarks:**

This allows one to control the matching of parent surface normals when creating the ruled surface. For example, normally when you create a ruled surface between two parent curves you don't want a 'bow tie' surface (one with the ends rotated 180 degrees so it crosses on itself in the middle). If you simply match the parent normals you'll get a 'bow tie' surface. To prevent this you use this method to set a state indicating that one or the other should be flipped before it's used. In this way, when the ruled surface is created, a 'bow tie' won't occur.

### **Parameters:**

**int pnum**

The number of the parent curve: 0 or 1.

**BOOL flip**

TRUE to match the parent surface normal; FALSE to not match it.

**Prototype:**

**BOOL GetFlip(int pnum);**

**Remarks:**

Returns the flip state of the specified parent curve.

**Parameters:**

**int pnum**

The number of the parent curve: 0 or 1.

**Prototype:**

**void SetCurveStartPoint(TimeValue t, int pnum, double startpoint);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the start point for the specified parent curve. Note: This is only applicable if the parent is a closed curve.

**Parameters:**

**TimeValue t**

The time at which to set the start point.

**int pnum**

The number of the parent curve: 0 or 1.

**double startpoint**

The start point in the range 0.0 to 1.0.

**Prototype:**

**double GetCurveStartPoint(TimeValue t, int pnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the start point for the specified parent curve. Note: This is only applicable if the parent is a closed curve.

**Parameters:**

**TimeValue t**

The time at which to get the start point.

**int pnum**

The number of the parent curve: 0 or 1.

**Operators:**

**Prototype:**

**NURBSRuledSurface & operator=(const NURBSRuledSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSRuledSurface& surf**

The surface to assign.

## Class NURBSULoftSurface

See Also: [Class NURBSSurface](#).

```
class NURBSULoftSurface : public NURBSSurface
```

### Description:

This class is available in release 2.0 and later only.

This class defines a dependent U Loft surface. A U Loft surface interpolates a surface across multiple curve sub-objects. The curves become U-axis contours of the surface. Methods are available to get/set the number of curves used to make the loft, append curves to the list, get/set the parent ids, and get/set the flipped state for each of the curves.

All methods of this class are implemented by the system.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSULoftSurface();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNULOftSurface;
```

```
mpObject = NULL;  
mpNSet = NULL;  
mParentId.SetCount(0);  
mParentIndex.SetCount(0);  
mFlip.SetCount(0);  
mCurveStartParam.SetCount(0);  
mTension.SetCount(0);  
mUseTangents.SetCount(0);  
mFlipTangents.SetCount(0);  
mAutoAlign = FALSE;  
mCloseLoft = FALSE;
```

**Prototype:**

```
virtual ~NURBSULoftSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetNumCurves(int num);
```

**Remarks:**

Sets the number of curves used by the U loft.

**Parameters:**

**int num**

The number of curves to set.

**Prototype:**

```
int GetNumCurves();
```

**Remarks:**

Returns the number of curves used by the surface.

**Prototype:**

**int AppendCurve(int index, BOOL flip);**

**Remarks:**

Adds a curve to the end of the list of curves used to make the U loft surface.

**Parameters:**

**int index**

The index of the curve to add in the **NURBSSet**.

**BOOL flip**

TRUE to flip the orientation of this curve's normal used to build the loft; otherwise FALSE.

**Return Value:**

The number of curves in the list prior to this one being added.

**Prototype:**

**int AppendCurve(NURBSId id, BOOL flip);**

**Remarks:**

Adds a curve to the end of the list of curves used to make the U loft surface.

**Parameters:**

**NURBSId id**

The NURBS id of the curve to add.

**BOOL flip**

TRUE to flip the orientation of this curve's normal used to build the loft; otherwise FALSE.

**Prototype:**

**void SetParent(int pnum, int index);**

**Remarks:**

Sets the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

**void SetParentId(int pnum, NURBSId id);**

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**Prototype:**

**void SetFlip(int pnum, BOOL flip);**

**Remarks:**

This method allows one to control the matching of parent normals when

creating the U loft surface.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**BOOL flip**

TRUE to flip the orientation of the normal when building the surface at this curve; otherwise FALSE.

**Prototype:**

**BOOL GetFlip(int pnum);**

**Remarks:**

Returns the flip state of the specified parent normal.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**Prototype:**

**void SetCurveStartPoint(TimeValue t, int pnum, double startpoint);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the start point for the specified parent curve. Note: This is only applicable if the parent is a closed curve.

**Parameters:**

**TimeValue t**

The time at which to set the start point.

**int pnum**

The parent number, 0, 1, 2, etc.

**double startpoint**

The start point in the range 0.0 to 1.0.

**Prototype:**

**double GetCurveStartPoint(TimeValue t, int pnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the start point for the specified parent curve. Note: This is only applicable if the parent is a closed curve.

**Parameters:**

**TimeValue t**

The time at which to get the start point.

**int pnum**

The parent number, 0, 1, 2, etc.

**Prototype:**

**void SetCurveTension(TimeValue t, int pnum, double tension);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the curve tension for the specified parent curve.

**Parameters:**

**TimeValue t**

The time at which to set the tension.

**int pnum**

The parent number, 0, 1, 2, etc.

**double tension**

The tension value to set.

**Prototype:**

**double GetCurveTension(TimeValue t, int pnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the tension setting of the specified parent curve at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the tension.

**int pnum**

The parent number, 0, 1, 2, etc.

**Prototype:**

**void SetCurveUseSurfaceTangent(int pnum, BOOL useTangent);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the curve use surface tangent setting. If the curve is a curve on surface, turning this on causes the U loft to use the tangency of the surface. This can help blend a loft smoothly onto a surface.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**BOOL useTangent**

TRUE to use the tangent; otherwise FALSE.

**Prototype:**

**BOOL GetCurveUseSurfaceTangent(int pnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the curve use surface tangent setting is on; FALSE if off.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**Prototype:**

**void SetFlipTangent(int pnum, BOOL flipTangent);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the flip tangent setting for the specified parent curve.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**BOOL flipTangent**

TRUE to flip the tangent; otherwise FALSE.

**Prototype:**

**BOOL GetFlipTangent(int pnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the tangent is flipped for the specified parent curve; otherwise FALSE.

**Parameters:**

**int pnum**

The parent number, 0, 1, 2, etc.

**Prototype:**

**void SetAutoAlign(BOOL autoalign);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets if auto align is on or off.

**Parameters:**

**BOOL autoalign**

TRUE for on; FALSE for off.

**Prototype:**

**BOOL GetAutoAlign();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if auto align is on; otherwise FALSE.

**Prototype:**

**void SetCloseLoft(BOOL closeLoft);**

**Remarks:**

This method is available in release 3.0 and later only.  
Sets if the loft is closed or not.

**Parameters:****BOOL closeLoft**

TRUE for closed; FALSE for open.

**Prototype:**

**BOOL GetCloseLoft();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns TRUE if loft is closed; otherwise FALSE.

**Operators:****Prototype:**

**NURBSULoftSurface & operator=(const NURBSULoftSurface&  
surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSULoftSurface& surf**

The surface to assign.

# Class NURBSLatheSurface

See Also: [Class NURBSSurface](#).

```
class NURBSLatheSurface : public NURBSSurface
```

## Description:

This class is available in release 2.0 and later only.

This class defines a dependent lathe surface. A lathe surface is generated from a curve sub-object. It is similar to a surface created with the Lathe modifier. The advantage is that a lathe sub-object is part of the NURBS model, so you can use it to construct other curve and surface sub-objects. Methods are available to get/set the parent index and id, get/set the axis and amount rotation of the lathe. All methods of this class are implemented by the system.

## Friend Classes:

```
friend class NURBSSet
```

## Methods:

protected:

## Prototype:

```
void Clean(NURBSIdTab ids);
```

## Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

## Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

## Prototype:

```
NURBSLatheSurface();
```

## Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNLatheSurface;
```

```
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;  
mXForm.IdentityMatrix();  
mRotation = 360.0;  
mCurveStartParam = 360.0;
```

**Prototype:**

```
virtual ~NURBSLatheSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the parent object.

**Parameters:**

**int index**

The index into the **NURBSSet** of the parent surface.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the specified parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBSId** of the parent. Note that a **NURBSId** won't be valid until the object has been instantiated in the scene

**Prototype:**

**void SetAxis(TimeValue t, Matrix3& ray);**

**Remarks:**

Sets the axis to use for the surface of revolution by specifying a time and a axis system.

**Parameters:**

**TimeValue t**

The time at which the axis is set.

**Matrix3& ray**

Specifies the axis for revolution. See [Class Matrix3](#).

**Prototype:**

**Matrix3& GetAxis(TimeValue t);**

**Remarks:**

Returns the axis system for the surface of revolution.

**Parameters:**

**TimeValue t**

The time at which to return the axis.

**Prototype:**

**void SetRotation(TimeValue t, double degrees);**

**Remarks:**

Sets the amount of rotation for the surface.

**Parameters:**

**TimeValue t**

The time at which to set the amount.

**double degrees**

The angle of the revolution in degrees.

**Prototype:**

**double GetRotation(TimeValue t);**

**Remarks:**

Returns the angle of the revolution in degrees.

**Parameters:**

**TimeValue t**

The time at which to return the angle.

**Prototype:**

**void SetCurveStartPoint(TimeValue t, double startpoint);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the start point at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the start point.

**double startpoint**

The start point to set in the range 0.0 to 1.0.

**Prototype:**

**double GetCurveStartPoint(TimeValue t);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the start point at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the start point.

**Operators:****Prototype:**

```
NURBSLatheSurface & operator=(const NURBSLatheSurface&  
surf);
```

**Remarks:**

Assignment operator.

**Parameters:**

```
const NURBSLatheSurface& surf
```

The surface to assign.

## Class NURBSCurveSurfaceIntersectionPoint

See Also: [Class NURBSPoint](#).

```
class NURBSCurveSurfaceIntersectionPoint : public NURBSPoint
```

### Description:

This class is available in release 2.5 and later only.

This class is used to create a dependent point at the intersection of a curve and a surface.

### Data Members:

protected:

**NURBSId mParentId[2];**

The NURBSIds of the parent surface and curve. Parent 0 should be the surface parent 1 should be the curve.

**int mParentIndex[2];**

The NURBSSet indexes of the parent surface and curve. Parent 0 should be the surface parent 1 should be the curve.

**double mSeed;**

The seed location is a U position along the length of the parent curve.

**BOOL mTrimCurve;**

The trim curve flag.

**BOOL mFlipTrim;**

The trim flip flag.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a

## **NURBSSet.**

### **Parameters:**

#### **NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### **Prototype:**

**NURBSCurveSurfaceIntersectionPoint(void);**

### **Remarks:**

Constructor. The data members are initialized as follows:

**mType = kNCurveSurfaceIntersectionPoint;**

**mpObject = NULL;**

**mpNSet = NULL;**

**mParentId[0] = mParentId[1] = 0;**

**mParentIndex[0] = mParentIndex[1] = -1;**

**mSeed = 0.5;**

**mTrimCurve = FALSE;**

**mFlipTrim = FALSE;**

### **Prototype:**

**virtual ~NURBSCurveSurfaceIntersectionPoint(void);**

### **Remarks:**

Destructor.

### **Prototype:**

**void SetSeed(double seed);**

### **Remarks:**

Sets the seed value.

### **Parameters:**

#### **double seed**

The U position along the length of the parent curve.

### **Prototype:**

**double GetSeed();**

**Remarks:**

Returns the seed value.

**Prototype:**

**void SetParent(int pnum, int index);**

**Remarks:**

Establishes the curve or surface used by specifying its index in the **NURBSSet**.

**Parameters:**

**int pnum**

Pass 0 for the surface; 1 for the curve.

**int index**

The index in the **NURBSSet** of the curve or surface.

**Prototype:**

**void SetParentId(int pnum, NURBSId id);**

**Remarks:**

Establishes the curve or surface used by specifying its **NURBSId**.

**Parameters:**

**int pnum**

Pass 0 for the surface; 1 for the curve.

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the parent curve or surface.

**Parameters:**

**int pnum**

Pass 0 for the surface; 1 for the curve.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the parent curve or surface.

**Parameters:**

**int pnum**

Pass 0 for the surface; 1 for the curve.

**Prototype:**

**BOOL GetTrimCurve();**

**Remarks:**

Returns the state of the trim curve flag. TRUE causes trimming; FALSE does not.

**Prototype:**

**void SetTrimCurve(BOOL trim);**

**Remarks:**

Sets the state of the trim curve flag.

**Parameters:**

**BOOL trim**

TRUE to trim; FALSE to not trim.

**Prototype:**

**BOOL GetFlipTrim();**

**Remarks:**

Returns the state of the trim flip flag. TRUE for flipped; FALSE for not flipped.

**Prototype:**

**void SetFlipTrim(BOOL flip);**

**Remarks:**

Sets the state of the trim flip flag.

**Parameters:**

**BOOL flip**

TRUE to flip; FALSE to not flip.

**Prototype:**

```
NURBSCurveSurfaceIntersectionPoint & operator=(const  
NURBSCurveSurfaceIntersectionPoint &pt);
```

**Remarks:**

Assignment operator.

**Parameters:**

```
const NURBSCurveSurfaceIntersectionPoint &pt
```

The point to assign.

## Class NURBSProjectNormalCurve

See Also: [Class NURBSCurve](#), [Class Point2](#).

```
class NURBSProjectNormalCurve : public NURBSCurve
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the Normal Projected Curve. A Normal Projected curve lies on a surface. It is based on an existing curve, which is projected onto the surface in the direction of the surface's normals.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

```
protected:
```

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

### Prototype:

```
NURBSProjectNormalCurve(void);
```

### Remarks:

Constructor.

### Prototype:

```
virtual ~NURBSProjectNormalCurve(void);
```

### Remarks:

Destructor.

**Prototype:**

**void SetParent(int pnum, int index);**

**Remarks:**

Sets the surface or curve used by specifying its index into the NURBSSet.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**int index**

The index in the NURBSSet of the surface or curve.

**Prototype:**

**void SetParentId(int pnum, NURBSId id);**

**Remarks:**

Sets the surface or curve used by specifying its NURBSId.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**NURBSId id**

The id of the surface or curve.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the NURBSSet of the surface or curve in use.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the NURBSId of the surface or curve in use.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**Prototype:**

**BOOL GetTrim();**

**Remarks:**

Returns the state of the trim flag. When set the surface is trimmed against the curve. When not set, the surface isn't trimmed

**Prototype:**

**void SetTrim(BOOL trim);**

**Remarks:**

Sets the state of the trim flag. When set the surface is trimmed against the curve. When not set, the surface isn't trimmed

**Parameters:**

**BOOL trim**

TRUE to trim; FALSE to not trim.

**Prototype:**

**BOOL GetFlipTrim();**

**Remarks:**

Returns the state of the trim flip flag. When set this trims the surface in the opposite direction

**Prototype:**

**void SetFlipTrim(BOOL flip);**

**Remarks:**

Sets the state of the trim flip flag. When set this trims the surface in the opposite direction

**Parameters:**

**BOOL flip**

TRUE to flip; FALSE to not flip.

**Prototype:**

**void SetSeed(Point2& seed);**

**Remarks:**

Sets the UV location of the seed value on the surface. If there is a choice of projections, the projection closest to the seed point is the one used to create the curve.

**Parameters:**

**Point2& seed**

The seed value to set.

**Prototype:**

**Point2 GetSeed();**

**Remarks:**

Returns the UV location of the seed value on the surface.

**Prototype:**

**NURBSProjectNormalCurve & operator=(const  
NURBSProjectNormalCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSProjectNormalCurve& curve**

The curve to assign.

## Class NURBSProjectVectorCurve

See Also: [Class NURBSCurve](#), [Class Point2](#), [Class Point3](#).

```
class NURBSProjectVectorCurve : public NURBSCurve
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the Vector Projected Curve. A Vector Projected curve lies on a surface. This is almost the same as a Normal Projected curve, except that the projection from the existing curve to the surface is in the direction of a vector that you can control. Vector projected curves may be used for trimming.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSProjectVectorCurve(void);
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNProjectVectorCurve;
```

```
mpObject = NULL;  
mpNSet = NULL;  
mParentId[0] = mParentId[1] = 0;  
mParentIndex[0] = mParentIndex[1] = -1;  
mTrim = FALSE;  
mFlipTrim = FALSE;  
mSeed = Point2(0.0, 0.0);  
mPVec = Point3(0.0, 0.0, 1.0);
```

**Prototype:**

```
virtual ~NURBSProjectVectorCurve(void);
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**int index**

The index in the **NURBSSet** of the surface or curve.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the surface or curve used by specifying its **NURBSId**.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**NURBSId id**

The id of the surface or curve.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index into the **NURBSSet** of surface or curve used.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of surface or curve used.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**Prototype:**

**BOOL GetTrim();**

**Remarks:**

Returns the state of the trim flag. When set the surface is trimmed against the curve. When not set, the surface isn't trimmed

**Prototype:**

**void SetTrim(BOOL trim);**

**Remarks:**

Sets the state of the trim flag. When set the surface is trimmed against the curve. When not set, the surface isn't trimmed

**Parameters:**

**BOOL trim**

TRUE to trim; FALSE to not trim.

**Prototype:**

**BOOL GetFlipTrim();**

**Remarks:**

Returns the state of the trim flip flag. When set this trims the surface in the opposite direction.

**Prototype:**

**void SetFlipTrim(BOOL flip);**

**Remarks:**

Sets the state of the trim flip flag. When set this trims the surface in the opposite direction

**Parameters:**

**BOOL flip**

TRUE to flip; FALSE to not flip.

**Prototype:**

**Point2 GetSeed();**

**Remarks:**

Returns the UV location of the seed value on the surface.

**Prototype:**

**void SetSeed(Point2& seed);**

**Remarks:**

Sets the UV location of the seed value on the surface. If there is a choice of projections, the projection closest to the seed point is the one used to create the curve.

**Parameters:**

**Point2& seed**

The seed value to set.

**Prototype:**

**void SetPVec(TimeValue t, Point3& pvec);**

**Remarks:**

Sets the projection vector used at the time passed.

**Parameters:**

**TimeValue t**

The time at which to set the projection vector.

### **Point3& pvec**

The vector to set.

#### **Prototype:**

**Point3& GetPVec(TimeValue t);**

#### **Remarks:**

Returns the projection vector used at the time passed.

#### **Parameters:**

**TimeValue t**

The time at which to get the projection vector.

#### **Prototype:**

**NURBSProjectVectorCurve & operator=(const  
NURBSProjectVectorCurve& curve);**

#### **Remarks:**

Assignment operator.

#### **Parameters:**

**const NURBSProjectVectorCurve& curve**

The curve to assign.

## Class NURBSSurfaceNormalCurve

See Also: [Class NURBSCurve](#).

```
class NURBSSurfaceNormalCurve : public NURBSCurve
```

### Description:

This class is available in release 2.5 and later only.

This provides access to the Surface Normal Curve. This is a curve created at a specified distance from a surface and normal to it.

Note: The parent curve specified below must have one of the following types: surface-surface intersection, U Iso, V Iso, normal projected, vector projected, CV curve on surface, or point curve on surface.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSSurfaceNormalCurve(void);
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNSurfaceNormalCurve;
```

```
mpObject = NULL;  
mpNSet = NULL;  
mParentId = 0;  
mParentIndex = -1;
```

**Prototype:**

```
virtual ~NURBSSurfaceNormalCurve(void);
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Establishes the parent curve to use by passing its index in the **NURBSSet**.

**Parameters:**

**int index**

The index in the **NURBSSet** of the parent curve to use.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Establishes the parent curve to use by passing its **NURBSId**.

**Parameters:**

**NURBSId id**

The id of the parent curve to use.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the **NURBSSet** of the parent curve.

**Prototype:**

```
NURBSId GetParentId();
```

**Remarks:**

Returns the **NURBSId** of the parent curve.

**Prototype:**

**void SetDistance(TimeValue t, double dist);**

**Remarks:**

Set the distance along the normal of the curve from the surface at the specified time.

**Parameters:**

**TimeValue t**

The time at which to set the distance.

**double dist**

The distance along the normal from the surface to the curve.

**Prototype:**

**double GetDistance(TimeValue t);**

**Remarks:**

Returns the distance along the normal from the surface to the curve.

**Parameters:**

**TimeValue t**

The time at which to get the distance.

**Prototype:**

**NURBSSurfaceNormalCurve & operator=(const  
NURBSSurfaceNormalCurve& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSSurfaceNormalCurve& curve**

The curve to assign.

## Class NURBSSurfSurfIntersectionCurve

See Also: [Class NURBSCurve](#), [Class Point2](#).

```
class NURBSSurfSurfIntersectionCurve : public NURBSCurve
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the Surface-Surface Intersection Curve. This is a curve that is defined by the intersection of two surfaces. You can use surface-surface intersection curves for trimming.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

```
protected:
```

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

### Prototype:

```
NURBSSurfSurfIntersectionCurve(void);
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNSurfSurfIntersectionCurve;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
```

```
mParentId[0] = mParentId[1] = 0;  
mParentIndex[0] = mParentIndex[1] = -1;  
mTrim[0] = mTrim[1] = FALSE;  
mFlipTrim[0] = mFlipTrim[1] = FALSE;  
mSeed = Point2(0.0, 0.0);
```

**Prototype:**

```
virtual ~NURBSSurfSurfIntersectionCurve(void);
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the surface or curve by specifying its index into the NURBSSet.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**int index**

The index into the NURBSSet of the object to set.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the surface or curve by specifying its index into the NURBSSet.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**NURBSId id**

The id of the object to set.

**Prototype:**

```
int GetParent(int pnum);
```

**Remarks:**

Returns the index into the NURBSSet of the curve or surface.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the NURBSId of the curve or surface.

**Parameters:**

**int pnum**

Pass 0 for the surface and 1 for the curve.

**Prototype:**

**BOOL GetTrim(int tnum);**

**Remarks:**

Returns the state of the specified trim flag. When on, the surface is trimmed against the intersection curve. When off, the surface isn't trimmed.

**Parameters:**

**int tnum**

Passing 0 trims the first parent surface, and passing 1 trims the second parent surface

**Prototype:**

**void SetTrim(int tnum, BOOL trim);**

**Remarks:**

Sets the state of the specified trim flag. When on, the surface is trimmed against the intersection curve. When off, the surface isn't trimmed.

**Parameters:**

**int tnum**

Passing 0 trims the first parent surface, and passing 1 trims the second parent surface

**BOOL trim**

TRUE to trim; FALSE to not trim.

**Prototype:**

**BOOL GetFlipTrim(int tnum);**

**Remarks:**

Returns the state of the trim flip flag. When on, the specified surface is trimmed in the opposite direction.

**Parameters:**

**int tnum**

0 for the first surface; 1 for the second surface.

**Prototype:**

**void SetFlipTrim(int tnum, BOOL flip);**

**Remarks:**

Sets the state of the trim flip flag. When on, the specified surface is trimmed in the opposite direction.

**Parameters:**

**int tnum**

0 for the first surface; 1 for the second surface.

**BOOL flip**

TRUE to flip; FALSE not to flip.

**Prototype:**

**void SetSeed(Point2& seed);**

**Remarks:**

Change the UV location of the seed value on the first surface. If there is a choice of intersections, the intersection closest to the seed point is the one used to create the curve.

**Parameters:**

**Point2& seed**

The UV location.

**Prototype:**

**Point2 GetSeed();**

**Remarks:**

Returns the UV location of the seed value on the first surface. If there is a choice of intersections, the intersection closest to the seed point is the one used to create the curve.

**Prototype:**

```
NURBSSurfSurfIntersectionCurve & operator=(const  
NURBSSurfSurfIntersectionCurve& curve);
```

**Remarks:**

Assignment operator.

**Parameters:**

```
const NURBSSurfSurfIntersectionCurve& curve  
The curve to assign.
```

## Class NURBSCurveOnSurface

See Also: [Class NURBSCVCurve](#).

```
class NURBSCurveOnSurface : public NURBSCVCurve
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the CV curve on surface parameters. These curves can be used for trimming the surface they lie on.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

#### Prototype:

```
NURBSCurveOnSurface(void);
```

#### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNCurveOnSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
```

```
mParentId = 0;
```

```
mParentIndex = -1;
```

```
mTrim = FALSE;
```

```
mFlipTrim = FALSE;
```

#### Prototype:

```
virtual ~NURBSCurveOnSurface(void);
```

#### Remarks:

Destructor.

#### Prototype:

```
void SetParent(int index);
```

#### Remarks:

Sets the index in the **NURBS**Set of the parent surface.

**Parameters:**

**int index**

The index into the **NURBS**Set of the parent surface.

**Prototype:**

**void SetParentId(NURBSId id);**

**Remarks:**

Sets the **NURBS**Id of the parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

**int GetParent();**

**Remarks:**

Returns the index in the **NURBS**Set of the specified parent object.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

Returns the **NURBS**Id of the parent. Note that a **NURBS**Id won't be valid until the object has been instantiated in the scene.

**Prototype:**

**BOOL GetTrim();**

**Remarks:**

Returns the state of the trim surface toggle.

**Prototype:**

**void SetTrim(BOOL trim);**

**Remarks:**

Sets the state of the trim surface toggle.

**Parameters:**

## **BOOL trim**

TRUE to trim the surface at the curve; FALSE to not trim.

### **Prototype:**

**BOOL GetFlipTrim();**

### **Remarks:**

Returns the state of the trim flip toggle.

### **Prototype:**

**void SetFlipTrim(BOOL flip);**

### **Remarks:**

Sets the state of the trim flip toggle. This controls which portion of the surface is trimmed.

### **Parameters:**

**BOOL flip**

TRUE for on; FALSE for off.

### **Prototype:**

**NURBSCurveOnSurface & operator=(const  
NURBSCurveOnSurface& curve);**

### **Remarks:**

Assignment operator.

### **Parameters:**

**const NURBSCurveOnSurface& curve**

The curve to assign.

## Class NURBSPointCurveOnSurface

See Also: [Class NURBSPointCurve](#).

```
class NURBSPointCurveOnSurface : public NURBSPointCurve
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the point curve on surface parameters. These curves can be used for trimming the surface they lie on.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

```
protected:
```

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

### Prototype:

```
NURBSPointCurveOnSurface(void);
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNPointCurveOnSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
```

```
mParentId = 0;
```

```
mParentIndex = -1;  
mTrim = FALSE;  
mFlipTrim = FALSE;
```

**Prototype:**

```
virtual ~NURBSPointCurveOnSurface(void);
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int index);
```

**Remarks:**

Sets the index in the **NURBSSet** of the parent surface.

**Parameters:**

**int index**

The index in the **NURBSSet** of the parent surface.

**Prototype:**

```
void SetParentId(NURBSId id);
```

**Remarks:**

Sets the **NURBSId** of the parent.

**Parameters:**

**NURBSId id**

The id to set.

**Prototype:**

```
int GetParent();
```

**Remarks:**

Returns the index in the **NURBSSet** of the parent surface.

**Prototype:**

```
NURBSId GetParentId();
```

**Remarks:**

Returns the **NURBSId** of the parent surface. Note that a **NURBSId** won't be

valid until the object has been instantiated in the scene.

**Prototype:**

**BOOL GetTrim();**

**Remarks:**

Returns the state of the trim toggle.

**Prototype:**

**BOOL GetFlipTrim();**

**Remarks:**

Returns the state of the trim flip toggle.

**Prototype:**

**void SetFlipTrim(BOOL flip);**

**Remarks:**

Sets the state of the trim flip toggle. This controls which portion of the parent surface is trimmed.

**Parameters:**

**BOOL flip**

TRUE for flipped; FALSE for not.

**Prototype:**

**NURBSPointCurveOnSurface & operator=(const  
NURBSPointCurveOnSurface& curve);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSPointCurveOnSurface& curve**

The curve to assign.

## Class NURBSUVLoftSurface

See Also: [Class NURBSSurface](#).

```
class NURBSUVLoftSurface : public NURBSSurface
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the UV Loft Surface. This surface is similar to the U Loft surface, but has a set of curves in the V dimension as well as the U dimension.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

```
protected:
```

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

### Prototype:

```
NURBSUVLoftSurface(void);
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNUVLoftSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
```

```
mUParentId.SetCount(0);  
mUParentIndex.SetCount(0);  
mVParentId.SetCount(0);  
mVParentIndex.SetCount(0);
```

**Prototype:**

```
virtual ~NURBSUVLoftSurface(void);
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetNumUCurves(int num);
```

**Remarks:**

Sets the number of curves in the U dimension.

**Parameters:**

**int num**

The number of curves to use in the U dimension.

**Prototype:**

```
int GetNumUCurves(void);
```

**Remarks:**

Returns the number of curves in the U dimension.

**Prototype:**

```
int AppendUCurve(int index);
```

**Remarks:**

Appends the specified curve (by NURBSSet index) to the list of U curves.

**Parameters:**

**int index**

The index in the **NURBSSet** of the curve to use.

**Prototype:**

```
int AppendUCurve(NURBSId id);
```

**Remarks:**

Appends the specified curve (by NURBSId) to the list of U curves.

**Parameters:**

**NURBSId id**

The **NURBSId** of the curve to use.

**Return Value:**

Returns the number of curves in the set prior to appending.

**Prototype:**

**void SetUParent(int pnum, int index);**

**Remarks:**

Sets the specified parent curve (by NURBSSet index) as the specified curve in the surface.

**Parameters:**

**int pnum**

The index into the list of U curves of the parent curve to set.

**int index**

The index into the **NURBSSet** of the parent curve to set.

**Prototype:**

**void SetUParentId(int pnum, NURBSId id);**

**Remarks:**

Sets the specified parent curve (by NURBSId) as the specified curve in the surface.

**Parameters:**

**int pnum**

The index into the list of U curves of the parent curve to set.

**NURBSId id**

The **NURBSId** of the parent curve to set.

**Prototype:**

**int GetUParent(int pnum);**

**Remarks:**

Returns the index into the **NURBSSet** of the specified parent curve.

**Parameters:**

**int pnum**

The zero based index of the parent curve.

**Prototype:**

**NURBSId GetUParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified parent curve. Note that a **NURBSId** won't be valid until the surface has been instantiated in the scene.

**Parameters:**

**int pnum**

The zero based index of the parent curve.

**Prototype:**

**void SetNumVCurves(int num);**

**Remarks:**

Sets the number of curves in the V dimension.

**Parameters:**

**int num**

The number of curves to use in the V dimension.

**Prototype:**

**int GetNumVCurves(void);**

**Remarks:**

Returns the number of curves in the V dimension.

**Prototype:**

**int AppendVCurve(int index);**

**Remarks:**

Appends the specified curve (by NURBSSet index) to the list of V curves.

**Parameters:**

**int index**

The index in the **NURBSSet** of the curve to use.

**Return Value:**

Returns the number of curves in the set prior to appending.

**Prototype:**

**int AppendVCurve(NURBSId id);**

**Remarks:**

Appends the specified curve (by NURBSId) to the list of U curves.

**Parameters:**

**NURBSId id**

The **NURBSId** of the curve to use.

**Return Value:**

Returns the number of curves in the set prior to appending.

**Prototype:**

**void SetVParent(int pnum, int index);**

**Remarks:**

Sets the specified parent curve (by **NURBSSet** index) as the specified curve in the surface.

**Parameters:**

**int pnum**

The index into the list of V curves of the parent curve to set.

**int index**

The index into the **NURBSSet** of the parent curve to set.

**Prototype:**

**void SetVParentId(int pnum, NURBSId id);**

**Remarks:**

Sets the specified parent curve (by NURBSId) as the specified curve in the surface.

**Parameters:**

**int pnum**

The index into the list of V curves of the parent curve to set.

**NURBSId id**

The **NURBSId** of the parent curve to set.

**Prototype:**

**int GetVParent(int pnum);**

**Remarks:**

Returns the index into the **NURBSSet** of the specified parent curve.

**Parameters:**

**int pnum**

The zero based index of the parent curve.

**Prototype:**

**NURBSId GetVParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified parent curve. Note that a **NURBSId** won't be valid until the surface has been instantiated in the scene.

**Parameters:**

**int pnum**

The zero based index of the parent curve.

**Prototype:**

**NURBSUVLoftSurface & operator=(const  
NURBSUVLoftSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSUVLoftSurface& surf**

The surface to assign.

## Class NURBSNBlendSurface

See Also: [Class NURBSSurface](#).

```
class NURBSNBlendSurface : public NURBSSurface
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the Multisided Blend surface. A Multisided Blend surface is a surface that "fills in" the edges defined by three or four other curve or surfaces. Unlike a regular, two-sided Blend surface, the curves or surfaces edges must form a closed loop<sup>3/4</sup>that is, they must completely surround the opening the Multisided Blend will cover.

Note: For the blend to work, the curves that define the blend must form a loop (that is, sequence head to tail, and the ends must match).

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

```
NURBSIdTab ids
```

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSNBlendSurface(void);
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNNBlendSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
for (int i = 0; i < 4; i++) {
    mParentId[i] = 0;
    mParentIndex[i] = -1;
    mParentEdge[i] = 0;
}
```

**Prototype:**

```
virtual ~NURBSNBlendSurface(void);
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the specified parent curve or surface (by NURBSSet index) as the specified edge for the surface.

**Parameters:**

**int pnum**

The index of the parent curve or surface to set.

**int index**

The index in the **NURBSSet** of the curve or surface.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the specified parent curve or surface (by NURBSId) as the specified edge for the surface.

**Parameters:**

**int pnum**

The index of the parent curve or surface to set.

**NURBSId id**

The id of the curve or surface to set.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the specified curve or surface.

**Parameters:**

**int pnum**

The index of the parent curve or surface to get.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified curve or surface.

**Parameters:**

**int pnum**

The index of the parent curve or surface to get.

**Prototype:**

**void SetEdge(int pnum, int edge);**

**Remarks:**

If using a surface for the blend surface edge, this method indicates which edge on the surface to use.

**Parameters:**

**int pnum**

The index of the parent surface.

**int edge**

The edge to use for the blend. One of the following values:

**0:** The low U edge.

**1:** The high U edge.

**2:** The low V edge.

**3:** The high V edge.

**Prototype:**

**int GetEdge(int pnum);**

**Remarks:**

Returns the edge used by the specified surface to create the blend.

**Parameters:**

**int pnum**

The index of the parent surface.

**Return Value:**

The edge used for the blend. One of the following values:

**0:** The low U edge.

**1:** The high U edge.

**2:** The low V edge.

**3:** The high V edge.

**Prototype:**

**NURBSNBlendSurface & operator=(const  
NURBSNBlendSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSNBlendSurface& surf**

The surface to assign.

## Class NURBS1RailSweepSurface

See Also: [Class NURBSSurface](#).

```
class NURBS1RailSweepSurface : public NURBSSurface
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the 1-Rail Sweep Surface. A 1-Rail Sweep Surface uses at least two curves. One curve, the "rail," defines one edge of the surface. The other curves define the surface's cross sections. The cross-section curves should intersect the rail curve. If the cross sections don't intersect the rail, the resulting surface is unpredictable.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

```
protected:
```

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

### Prototype:

```
NURBS1RailSweepSurface();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kN1RailSweepSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;  
mRailId = NULL;  
mRailIndex = -1;  
mParentId.SetCount(0);  
mParentIndex.SetCount(0);  
mFlip.SetCount(0);  
mCurveStartParam.SetCount(0);  
mSnapCrossSections = FALSE;  
mRoadlike = FALSE;  
mXForm.IdentityMatrix();
```

**Prototype:**

```
virtual ~NURBS1RailSweepSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetParentRail(int index);
```

**Remarks:**

Establishes the curve to use as the rail by specifying its index in the **NURBSSet**.

**Parameters:**

**int index**

The index in the **NURBSSet**.

**Prototype:**

```
void SetParentRailId(NURBSId id);
```

**Remarks:**

Establishes the curve to use as the rail by specifying its **NURBSId**.

**Parameters:**

**NURBSId id**

The id of the rail curve to use.

**Prototype:**

**int GetParentRail();**

**Remarks:**

Returns the index in the **NURBSSet** of the rail curve.

**Prototype:**

**NURBSId GetParentRailId();**

**Remarks:**

Returns the **NURBSId** of the rail curve.

**Prototype:**

**void SetNumCurves(int num);**

**Remarks:**

Sets the number of cross-section curves.

**Parameters:**

**int num**

The number of cross-section curves to use.

**Prototype:**

**int GetNumCurves(void);**

**Remarks:**

Returns the number of cross-section curves used.

**Prototype:**

**int AppendCurve(int index, BOOL flip);**

**Remarks:**

Adds a curve to the end of the list of cross-section curves by specifying the index in the **NURBSSet**.

**Parameters:**

**int index**

The index in the **NURBSSet** of the cross-section curve to append.

**BOOL flip**

TRUE to reverse (or flip) the direction of the curve; FALSE to use the non-reversed orientation.

**Return Value:**

The number of cross-section curves prior to appending.

**Prototype:**

**int AppendCurve(NURBSId id, BOOL flip);**

**Remarks:**

Adds a curve to the end of the list of cross-section curves by specifying a **NURBSId**.

**Parameters:**

**NURBSId id**

Specifies the cross-section curve to append.

**BOOL flip**

TRUE to reverse (or flip) the direction of the curve; FALSE to use the non-reversed orientation.

**Return Value:**

The number of cross-section curves prior to appending.

**Prototype:**

**void SetParent(int pnum, int index);**

**Remarks:**

Specifies the curve to use as a cross-section via its index in the **NURBSSet**.

**Parameters:**

**int pnum**

The zero based index of the curve to set.

**int index**

The index in the **NURBSSet** of the curve.

**Prototype:**

**void SetParentId(int pnum, NURBSId id);**

**Remarks:**

Specifies the curve to use as a cross-section via its **NURBSId**.

**Parameters:**

**int pnum**

The zero based index of the curve to set.

**NURBSId id**

The id of the curve.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the specified cross-section curve.

**Parameters:**

**int pnum**

The zero based index of the curve to get.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified cross-section curve.

**Parameters:**

**int pnum**

The zero based index of the curve to get.

**Prototype:**

**void SetFlip(int pnum, BOOL flip);**

**Remarks:**

Sets the reversed (or flipped) state of the specified cross-section curve.

**Parameters:**

**int pnum**

The zero based index of the curve.

**BOOL flip**

TRUE to reverse the direction; FALSE for the normal direction.

**Prototype:**

**BOOL GetFlip(int pnum);**

**Remarks:**

Returns the reversed (or flipped) state of the specified cross-section curve. TRUE is reversed; FALSE is not.

**Parameters:**

**int pnum**

The zero based index of the curve.

**Prototype:**

**void SetParallel(BOOL para);**

**Remarks:**

Sets the state of the parallel flag. When on, it ensures that the sweep surface's normal is parallel to the rail.

**Parameters:**

**BOOL para**

TRUE for on; FALSE for off.

**Prototype:**

**BOOL GetParallel();**

**Remarks:**

Returns the state of the parallel flag. When TRUE, 3ds max ensures that the sweep surface's normal is parallel to the rail.

**Prototype:**

**void SetCurveStartPoint(TimeValue t, int pnum, double startpoint);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the start point for the specified parent curve. Note: This is only applicable if the parent is a closed curve.

**Parameters:**

**TimeValue t**

The time at which to set the start point.

**int pnum**

The zero based index of the curve in the set of cross sections.

**double startpoint**

The start point in the range 0.0 to 1.0.

**Prototype:****double GetCurveStartPoint(TimeValue t, int pnum);****Remarks:**

This method is available in release 3.0 and later only.

Returns the start point of the specified parent curve.

**Parameters:****TimeValue t**

The time at which to get the start point.

**int pnum**

The zero based index of the curve in the set of cross sections.

**Prototype:****void SetSnapCS(BOOL snapCS);****Remarks:**

This method is available in release 3.0 and later only.

Sets the snap cross sections setting. When on, cross-section curves are translated so they intersect the rail.

**Parameters:****BOOL snapCS**

TRUE for on; FALSE for off.

**Prototype:****BOOL GetSnapCS();****Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if snap to Cross Section is on; otherwise FALSE.

**Prototype:**

**void SetRoadlike(BOOL roadlike);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the roadlike setting to on or off. When on, the sweep uses a constant up-vector so the cross sections twist uniformly as they travel along the rail. In other words, the cross sections bank like a car following a road, or a camera following a path controller in 3ds max.

**Parameters:**

**BOOL roadlike**

TRUE for on; FALSE for off.

**Prototype:**

**BOOL GetRoadlike();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if the roadlike setting is on; otherwise FALSE.

**Prototype:**

**void SetAxis(TimeValue t, Matrix3& ray);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the axis of the sweep.

**Parameters:**

**TimeValue t**

The time at which to set the axis.

**Matrix3& ray**

The axis system to set.

**Prototype:**

**Matrix3& GetAxis(TimeValue t);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the axis of the sweep.

**Parameters:**

**TimeValue t**

The time at which to get the axis.

**Operators:**

**Prototype:**

**NURBS1RailSweepSurface & operator=(const  
NURBS1RailSweepSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBS1RailSweepSurface& surf**

The surface to assign.

## Class NURBS2RailSweepSurface

See Also: [Class NURBSSurface](#).

```
class NURBS2RailSweepSurface : public NURBSSurface
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the 2-Rail Sweep Surface. A 2-Rail Sweep surface uses at least three curves. Two curves, the "rails," define the two edges of the surface. The other curves define the surface's cross sections. A 2-Rail Sweep surface is similar to a 1-Rail sweep. The additional rail gives you more control over the shape of the surface.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

```
protected:
```

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

```
NURBSIdTab ids
```

A table with the IDs of each object in the **NURBSSet**.

```
public:
```

### Prototype:

```
NURBS2RailSweepSurface();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kN2RailSweepSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;  
mParentId.SetCount(0);  
mParentIndex.SetCount(0);  
mRailParentId[0] = mRailParentId[1] = NULL;  
mRailParentIndex[0] = mRailParentIndex[1] = -1;  
mCurveStartParam.SetCount(0);  
mSnapCrossSections = FALSE;
```

**Prototype:**

```
virtual ~NURBS2RailSweepSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetNumCurves(int num);
```

**Remarks:**

Sets the number of cross-section curves.

**Parameters:**

**int num**

The number of cross-section curves to use.

**Prototype:**

```
int GetNumCurves(void);
```

**Remarks:**

Returns the number of cross-section curves used.

**Prototype:**

```
int AppendCurve(int index, BOOL flip);
```

**Remarks:**

Adds a curve to the end of the list of cross-section curves by specifying the index in the **NURBSSet**.

**Parameters:**

**int index**

The index in the **NURBSSet** of the cross-section curve to append.

**BOOL flip**

TRUE to reverse (or flip) the direction of the curve; FALSE to use the non-reversed orientation.

**Return Value:**

The number of cross-section curves prior to appending.

**Prototype:**

**int AppendCurve(NURBSId id, BOOL flip);**

**Remarks:**

Adds a curve to the end of the list of cross-section curves by specifying a **NURBSId**.

**Parameters:**

**NURBSId id**

Specifies the cross-section curve to append.

**BOOL flip**

TRUE to reverse (or flip) the direction of the curve; FALSE to use the non-reversed orientation.

**Return Value:**

The number of cross-section curves prior to appending.

**Prototype:**

**void SetParent(int pnum, int index);**

**Remarks:**

Specifies the curve to use as a cross-section via its index in the **NURBSSet**.

**Parameters:**

**int pnum**

The zero based index of the curve to set.

**int index**

The index in the **NURBSSet** of the curve.

**Prototype:**

**void SetParentId(int pnum, NURBSId id);**

**Remarks:**

Specifies the curve to use as a cross-section via its **NURBSId**.

**Parameters:**

**int pnum**

The zero based index of the curve to set.

**NURBSId id**

The id of the curve.

**Prototype:**

**int GetParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the specified cross-section curve.

**Parameters:**

**int pnum**

The zero based index of the curve to get.

**Prototype:**

**NURBSId GetParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified cross-section curve.

**Parameters:**

**int pnum**

The zero based index of the curve to get.

**Prototype:**

**void SetFlip(int pnum, BOOL flip);**

**Remarks:**

Sets the reversed (or flipped) state of the specified cross-section curve.

**Parameters:**

**int pnum**

The zero based index of the curve.

**BOOL flip**

TRUE to reverse the direction; FALSE for the normal direction.

**Prototype:**

**BOOL GetFlip(int pnum);**

**Remarks:**

Returns the reversed (or flipped) state of the specified cross-section curve. TRUE is reversed; FALSE is not.

**Parameters:**

**int pnum**

The zero based index of the curve.

**Prototype:**

**void SetParallel(BOOL para);**

**Remarks:**

Sets the state of the parallel flag. When off, it allows the cross section curve to bank to follow the rails. If on, it maintains the initial orientation of the cross section during the course of the sweep

**Parameters:**

**BOOL para**

TRUE for on; FALSE for off.

**Prototype:**

**BOOL GetParallel();**

**Remarks:**

Returns the state of the parallel flag.

**Prototype:**

**void SetScale(BOOL scale);**

**Remarks:**

To allow the cross sections to run along the rails, the cross section curves may need to be scaled. This method controls if the scaling is done uniformly or non-uniformly.

**Parameters:**

**BOOL scale**

TRUE to scale uniformly; FALSE to scale non-uniformly.

**Prototype:**

**BOOL GetScale();**

**Remarks:**

Returns TRUE if the cross section curves are scaled uniformly; FALSE for non-uniformly.

**Prototype:**

**void SetRailParent(int pnum, int index);**

**Remarks:**

Specifies the index in the **NURBSSet** of the first or second rail curve to use.

**Parameters:**

**int pnum**

Pass 0 for the first rail curve; 1 for the second rail curve.

**int index**

The index in the **NURBSSet** of the curve to use as a rail.

**Prototype:**

**void SetRailParentId(int pnum, NURBSId id);**

**Remarks:**

Specifies the **NURBSId** of the first or second rail curve to use.

**Parameters:**

**int pnum**

Pass 0 for the first rail curve; 1 for the second rail curve.

**NURBSId id**

The id of the curve to use as a rail.

**Prototype:**

**int GetRailParent(int pnum);**

**Remarks:**

Returns the index in the **NURBSSet** of the specified parent rail curve.

**Parameters:**

**int pnum**

Pass 0 for the first rail curve; 1 for the second rail curve.

**Prototype:**

**NURBSId GetRailParentId(int pnum);**

**Remarks:**

Returns the **NURBSId** of the specified parent rail curve.

**Parameters:**

**int pnum**

Pass 0 for the first rail curve; 1 for the second rail curve.

**Prototype:**

**void SetSnapCS(BOOL snapCS);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the snap cross sections setting. When on, cross-section curves are translated so they intersect the rail.

**Parameters:**

**BOOL snapCS**

TRUE for on; FALSE for off.

**Prototype:**

**BOOL GetSnapCS();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if snap cross sections is on; otherwise FALSE.

**Prototype:**

**void SetCurveStartPoint(TimeValue t, int pnum, double startpoint);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the start point for the specified parent curve at the time passed.

**Parameters:**

**TimeValue t**

The time at which to set the start point.

**int pnum**

Pass 0 for the first rail curve; 1 for the second rail curve.

**double startpoint**

The start point in the range of 0.0 to 1.0.

**Prototype:**

**double GetCurveStartPoint(TimeValue t, int pnum);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the start point for the specified parent curve at the time passed.

**Parameters:****TimeValue t**

The time at which to get the start point.

**int pnum**

Pass 0 for the first rail curve; 1 for the second rail curve.

**Operators:****Prototype:**

**NURBS2RailSweepSurface & operator=(const  
NURBS2RailSweepSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBS2RailSweepSurface& surf**

The surface to assign.

## Class NURBSCapSurface

See Also: [Class NURBSSurface](#).

```
class NURBSCapSurface : public NURBSSurface
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the Cap Surface. A Cap Surface is a surface that caps a closed curve or the edge of a closed surface. Caps are especially useful with extruded surfaces.

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSCapSurface(void);
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNCapSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
```

```
mParentId = 0;
```

```
mParentIndex = -1;
```

**mParentEdge = -1;**  
**mCurveStartParam = 0.0;**

**Prototype:**

**virtual ~NURBSCapSurface(void);**

**Remarks:**

Destructor.

**Prototype:**

**void SetParent(int index);**

**Remarks:**

Establishes the curve or surface that's capped by specifying its index in the **NURBSSet**.

**Parameters:**

**int index**

The index in the **NURBSSet** of the curve or surface to cap.

**Prototype:**

**void SetParentId(NURBSId id);**

**Remarks:**

Establishes the curve or surface that's capped by specifying its **NURBSId**.

**Parameters:**

**NURBSId id**

The id of the curve or surface to cap.

**Prototype:**

**int GetParent(void);**

**Remarks:**

Returns the index in the **NURBSSet** of the curve or surface that's capped.

**Prototype:**

**NURBSId GetParentId(void);**

**Remarks:**

Returns the **NURBSId** of the curve or surface that's capped.

**Prototype:****void SetEdge(int edge);****Remarks:**

Establishes which edge of the closed parent surface is capped.

**Parameters:****int edge**

The edge to cap. One of the following values:

**0:** The low U edge.

**1:** The high U edge.

**2:** The low V edge.

**3:** The high V edge.

**Prototype:****int GetEdge();****Remarks:**

Returns the edge of the closed parent surface is capped.

**Return Value:**

One of the following values:

**0:** The low U edge.

**1:** The high U edge.

**2:** The low V edge.

**3:** The high V edge.

**Prototype:****void SetCurveStartPoint(TimeValue t, double startpoint);****Remarks:**

This method is available in release 3.0 and later only.

Sets the start point for the curve. Note: This is only applicable if the parent is a closed curve.

**Parameters:****TimeValue t**

The time to set the start point.

## **double startpoint**

The start point to set.

### **Prototype:**

**double GetCurveStartPoint(TimeValue t);**

### **Remarks:**

This method is available in release 3.0 and later only.

Returns the start point for the curve. Note: This is only applicable if the parent is a closed curve.

### **Parameters:**

#### **TimeValue t**

The time to get the start point.

### **Prototype:**

**NURBSCapSurface & operator=(const NURBSCapSurface& surf);**

### **Remarks:**

Assignment operator.

### **Parameters:**

**const NURBSCapSurface& surf**

The surface to assign.

## Class NURBSMultiCurveTrimSurface

See Also: [Class NURBSSurface](#).

```
class NURBSMultiCurveTrimSurface : public NURBSSurface
```

### Description:

This class is available in release 2.5 and later only.

This class provides access to the Multicurve Trim Surface which is a surface that is trimmed by multiple curves forming a loop.

### Friend Classes:

```
friend class NURBSSet
```

### Methods:

protected:

### Prototype:

```
void Clean(NURBSIdTab ids);
```

### Remarks:

This method is available in release 3.0 and later only.

This methods breaks the relation between this **NURBSObject** and a **NURBSSet**.

### Parameters:

**NURBSIdTab ids**

A table with the IDs of each object in the **NURBSSet**.

public:

### Prototype:

```
NURBSMultiCurveTrimSurface();
```

### Remarks:

Constructor. The data members are initialized as follows:

```
mType = kNMultiCurveTrimSurface;
```

```
mpObject = NULL;
```

```
mpNSet = NULL;
```

```
mParentId.SetCount(0);
```

```
mParentIndex.SetCount(0);  
mSurfaceId = NULL;  
mSurfaceIndex = -1;  
mFlipTrim = FALSE;
```

**Prototype:**

```
virtual ~NURBSMultiCurveTrimSurface();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetNumCurves(int num);
```

**Remarks:**

Sets the number of curves used for the trim loop.

**Parameters:**

**int num**

The number of curves to use.

**Prototype:**

```
int GetNumCurves(void);
```

**Remarks:**

Returns the number of curves used for the trim loop.

**Prototype:**

```
int AppendCurve(int index);
```

**Remarks:**

Adds the specified curve (using a **NURBSSet** index) to the end of the list of curves comprising the trim loop.

**Parameters:**

**int index**

The index into the **NURBSSet** of the curve to add.

**Return Value:**

The number of curves in the loop prior to appending.

**Prototype:**

```
int AppendCurve(NURBSId id);
```

**Remarks:**

Adds the specified curve (using a **NURBSSet** index) to the end of the list of curves comprising the trim loop.

**Parameters:**

**NURBSId id**

The id to use.

**Return Value:**

The number of curves in the loop prior to appending.

**Prototype:**

```
void SetParent(int pnum, int index);
```

**Remarks:**

Sets the specified curve in the trim loop to the curve specified by an index into the **NURBSSet**.

**Parameters:**

**int pnum**

Zero based index of which curve to set.

**int index**

The index into the **NURBSSet** of the curve to use.

**Prototype:**

```
void SetParentId(int pnum, NURBSId id);
```

**Remarks:**

Sets the specified curve in the trim loop to the curve specified by a **NURBSId**.

**Parameters:**

**int pnum**

Zero based index of which curve to set.

**NURBSId id**

The id of the curve to use.

**Prototype:****int GetParent(int pnum);****Remarks:**

Returns the index in the **NURBSSet** of the specified parent object.

**Parameters:****int pnum**

Zero based index of which curve to get.

**Prototype:****NURBSId GetParentId(int pnum);****Remarks:**

Returns the **NURBSId** of the specified curve in the trim loop.

**Parameters:****int pnum**

Zero based index of which curve to get.

**Prototype:****void SetSurfaceParent(int index);****Remarks:**

Sets the surface that's trimmed by specifying it's index into the **NURBSSet**.

**Parameters:****int index**

The index in the **NURBSSet** of the surface.

**Prototype:****void SetSurfaceParentId(NURBSId id);****Remarks:**

Sets the surface that's trimmed by specifying it's **NURBSId**.

**Parameters:****NURBSId id**

The id of the surface.

**Prototype:**

**int GetSurfaceParent();**

**Remarks:**

Returns the index in the **NURBSSet** of the surface that is trimmed.

**Prototype:**

**NURBSId GetSurfaceParentId();**

**Remarks:**

Returns the **NURBSId** of the surface that's trimmed.

**Prototype:**

**BOOL GetFlipTrim();**

**Remarks:**

Returns the state of the flip trim flag. This controls which side of the curve is trimmed from the surface.

**Prototype:**

**void SetFlipTrim(BOOL flip);**

**Remarks:**

Sets the state of the flip trim flag.

**Parameters:**

**BOOL flip**

TRUE to flip; FALSE to not flip.

**Operators:**

**Prototype:**

**NURBSMultiCurveTrimSurface & operator=(const  
NURBSMultiCurveTrimSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSMultiCurveTrimSurface& surf**

The surface to assign.

## Class NURBSTextureSurface

See Also: [Class NURBSControlVertex](#), [Class NURBSSurface](#), [Class NURBSTexturePoint](#).

class NURBSTextureSurface

### Description:

This class is available in release 2.5 and later only.

A NURBS texture surface is a surface associated with the surface sub-object. 3ds max uses the texture surface to control how materials are mapped. In effect, changing the texture surface stretches or otherwise changes the UV coordinates for the surface, altering the mapping.

This class provides access to the NURBS Texture Surface. This is a 2D (not 3D) surface that lives in the parameter space of the corresponding **NURBSSurface** which controls the texture mapping used by the **NURBSSurface**.

### Methods:

#### Prototype:

**NURBSTextureSurface();**

#### Remarks:

Constructor. The data members are initialized as follows:

**mMapperType = kNMapDefault;**

**mpPoints = NULL;**

#### Prototype:

**~NURBSTextureSurface();**

#### Remarks:

Destructor. Any **mpPoints** are deleted.

#### Prototype:

**NURBSTexSurfType MapperType();**

#### Remarks:

This method is available in release 3.0 and later only.

Returns the NURBS Texture Surface Type. See [List of NURBS Texture](#)

## [Surface Types.](#)

### **Prototype:**

**void SetMapperType(NURBSTexSurfType type);**

### **Remarks:**

This method is available in release 3.0 and later only.

Sets the NURBS Texture Surface Type.

### **Parameters:**

**NURBSTexSurfType type**

The type to set. See [List of NURBS Texture Surface Types](#).

### **Prototype:**

**void SetParent(int index);**

### **Remarks:**

This method is available in release 3.0 and later only.

This sets the Id of the "source" surface for a **kNMapSurfaceMapper** texture surface, it should be NULL in other cases. This is only used if the **NURBSTexSurfType** is set the **kNMapSurfaceMapper**. In that case the texture surface is generated by projecting the texture of another NURBS surface sub-object in the NURBS model. The projection travels along the direction of the normals of the source surface. Projected texture surfaces are relational. This method set the parent surface which is projected.

### **Parameters:**

**int index**

The index in the **NURBSSet** of the source surface.

### **Prototype:**

**void SetParentId(NURBSId id);**

### **Remarks:**

This method is available in release 3.0 and later only.

This sets the index in the NURBSSet of the "source" surface for a **kNMapSurfaceMapper** texture surface. This is only used if the

**NURBSTexSurfType** is set the **kNMapSurfaceMapper**. In that case the texture surface is generated by projecting the texture of another NURBS surface sub-object in the NURBS model. The projection travels along the direction of the normals of the source surface. Projected texture surfaces are relational. This method set the parent surface which is projected.

**Parameters:**

**NURBSId id**

The ID of the source surface.

**Prototype:**

**int GetParent();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the index in the NURBSSet of the surface that's mapped.

**Prototype:**

**NURBSId GetParentId();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the ID of the surface that's mapped.

**Prototype:**

**void SetNumPoints(int u, int v);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the number of points in U and V for the texture surface.

**Parameters:**

**int u**

The number of points in U.

**int v**

The number of points in V.

**Prototype:**

**int GetNumUPoints();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the number of points in U.

**Prototype:**

**int GetNumVPoints();**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the number of points in V.

**Prototype:**

**void GetNumPoints(int &u, int &v);**

**Remarks:**

This method is available in release 3.0 and later only.  
Retrieves the number of points in U and V.

**Parameters:**

**int &u**

The number of points in U is returned here.

**int &v**

The number of points in V is returned here.

**Prototype:**

**NURBSTexturePoint\* GetPoint(int u, int v);**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns a pointer to the specified NURBS Texture Point.

**Parameters:**

**int u**

Specifies the U point.

**int v**

Specifies the V point.

**Prototype:**

**void SetPoint(int u, int v, NURBSTexturePoint& pnt);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the specified NURBS Texture Point.

**Parameters:**

**int u**

Specifies the U point.

**int v**

Specifies the V point.

**NURBSTexturePoint& pnt**

The point to set.

**Prototype:**

**NURBSTextureSurface & operator=(const  
NURBSTextureSurface& surf);**

**Remarks:**

Assignment operator.

**Parameters:**

**const NURBSTextureSurface& surf**

The surface to assign.

## List of NURBS Results

See Also: [Working With NURBS](#).

These results show up when one is modifying existing objects. This enum has the following choices:

### **kNOk**

The function succeeded.

### **kNInvalidObject**

The specified object was invalid. For example, if you use the **SetSurfaceApprox()** function and specify a non-NURBS object as input you'll get this error.

### **kNInvalidId**

The specified ID was invalid. For example, if you use the **Transform()** function and pass an invalid **NURBSId** you'll get this error.

### **kNInvalidParameter**

A specified parameter was invalid. For example if you tried to create a cone using **GenNURBSConeSurface()** with a radius  $\leq 0$  you'd get this error.

### **kNBad**

The function failed. This is a catch-all for any kind of unwanted result not covered above.

## Class NURBSDisplay

See Also: [Class NURBSSet](#).

class NURBSDisplay

### Description:

This class is available in release 2.5 or later.

This class provides information about the display of the **NURBSSet** in the 3D viewports. An instance of this class is maintained by each **NURBSSet**.

### Data Members:

#### **BOOL mDisplayCurves;**

TRUE if curves are displayed; otherwise FALSE.

#### **BOOL mDisplaySurfaces;**

TRUE if surfaces are displayed; otherwise FALSE.

#### **BOOL mDisplayLattices;**

TRUE if lattices are displayed; otherwise FALSE.

#### **BOOL mDisplaySurfCVLattices;**

TRUE if surface CV lattices are displayed; otherwise FALSE.

#### **BOOL mDisplayCurveCVLattices;**

TRUE if curve CV lattices are displayed; otherwise FALSE.

#### **BOOL mDisplayDependents;**

TRUE if dependent sub-objects are displayed; otherwise FALSE.

#### **BOOL mDisplayTrimming;**

TRUE if surface trimming is displayed; otherwise FALSE.

#### **BOOL mDegradeOnMove;**

TRUE if the surface may degrade while transforming it; otherwise FALSE.

### Methods:

### Prototype:

**NURBSDisplay();**

### Remarks:

Constructor. The data members are initialized as follows:

**mDisplayCurves = TRUE;**

```
mDisplaySurfaces = TRUE;  
mDisplayLattices = FALSE;  
mDisplaySurfCVLattices = TRUE;  
mDisplayCurveCVLattices = TRUE;  
mDisplayDependents = TRUE;  
mDisplayTrimming = TRUE;  
mDegradeOnMove = TRUE;
```

**Prototype:**

```
NURBSDisplay & operator=(const NURBSDisplay& disp);
```

**Remarks:**

Assignment operator.

**Parameters:**

```
const NURBSDisplay& disp
```

The object to assign.

## List of NURBSSubObjectLevel Options

See Also: [Class NURBSSurface](#), [Class NURBSSet](#).

One of the following enum values describes the sub-object level of the object.

### **kNTopLevel**

The object level.

### **kNSurfaceCVLevel**

The surface CV sub-object level.

### **kNSurfaceLevel**

The surface level.

### **kNCurveCVLevel**

The curve CV sub-object level.

### **kNPointLevel**

The point sub-object level.

### **kNCurveLevel**

The curve sub-object level.

### **kNImportsLevel**

The imports level.

## Class BezierShape

See Also: [Class ShapeObject](#), [Class Spline3D](#), [Class PatchCapInfo](#), [Class ShapeVSEL](#), [Class ShapeSSEL](#), [Class ShapePSEL](#), [Class Material](#), [Working with Shapes and Splines](#).

```
class BezierShape : public BaseInterfaceServer
```

### Description:

Defines a basic bezier shape object. The **BezierShape** is effectively a collection of Bezier Splines. For example the 3ds max Donut object has two splines in a hierarchy to make a shape. The **BezierShape** contains these splines.

### Method Groups:

The following hyperlinks jump to the start of groups of related methods within the class:

[Constructors / Destructor / Init\(\)](#)

[Get/SetVert\(\)](#)

[Render / Select / Snap / HitTest](#)

[Shape Hierarchy / Reverse\(\) / MakeFirst\(\)](#)

[MakePolyShape\(\) / ReadyCachedPolyShape\(\)](#)

[Bounding Box / Transform\(\) / InvalidateGeomCache\(\) / InvalidateCapCache\(\) / FreeAll\(\)](#)

[Data Flow Evaluation](#)

[Display Flags](#)

[Selection / UpdateSels\(\)](#)

[Load\(\) / Save\(\)](#)

[SplineCount\(\) / GetSpline\(\) / NewSpline\(\) / AddSpline\(\) / DeleteSpline\(\) /](#)

[InsertSpline\(\) / NewShape\(\)](#)

[Get/SetClosures\(\)](#)

[GetNumVerts\(\) / GetNumSegs\(\)](#)

[Capping Methods](#)

[Operators](#)

### Data Members:

public:

```
    PatchCapInfo patchCap;
```

Patch capping cache (mesh capping and hierarchy caches stored in PolyShape cache)

**BOOL patchCapCacheValid;**

Indicates if the patch cap is valid or not.

**Spline3D \*\*splines;**

A pointer to the list of splines.

**int splineCount;**

The number of splines in this shape.

**int steps;**

Number of steps (a value of -1 will use adaptive).

**BOOL optimize;**

Setting this to TRUE optimizes linear segments

**ShapeVsel vertSel;**

The selected vertices.

**ShapeSsel segSel;**

The selected segments.

**ShapePsel polySel;**

The selected polygons.

**int bezVecPoly;**

This is used internally in hit testing.

**int bezVecVert;**

This is used internally in hit testing.

**DWORD selLevel;**

Selection level.

**SHAPE\_OBJECT** - Object level selection.

**SHAPE\_SPLINE** - Spline level selection (a single polygon within the shape).

**SHAPE\_SEGMENT** - Segment level selection.

**SHAPE\_VERTEX** - Vertex level selection.

**DWORD dispFlags;**

Display attribute flags. See [List of BezierShape Display Flags](#).

**Tab<bindShape> bindList;**

The table of bind points. See [Template Class Tab](#).

**Methods:**

## **Constructors / Destructor / Init()**

**Prototype:**

**BezierShape();**

**Remarks:**

The data members are initialized as follows:

```
splines = NULL;  
splineCount = 0;  
dispFlags = 0;  
selLevel = SHAPE_OBJECT;  
bdgBox.Init();  
bezVecPoly = -1;  
bezVecVert = -1;  
vertSel.Empty();  
segSel.Empty();  
polySel.Empty();  
optimize = FALSE;  
steps = 5;
```

**Prototype:**

**BezierShape(BezierShape& fromShape);**

**Remarks:**

Constructor. The shape is initialized based on the **fromShape**.

**Prototype:**

**void Init();**

**Remarks:**

Initializes the **BezierShape**. The data members are initialized as follows:

```
splines = NULL;
```

```
splineCount = 0;  
dispFlags = 0;  
selLevel = SHAPE_OBJECT;  
bdgBox.Init();  
bezVecPoly = -1;  
bezVecVert = -1;  
vertSel.Empty();  
segSel.Empty();  
polySel.Empty();  
optimize = FALSE;  
steps = 5;
```

**Prototype:**

```
~BezierShape();
```

**Remarks:**

Destructor. All the splines in this shape are cleared.

**Get/SetVert()**

**Prototype:**

```
Point3& GetVert(int poly, int i);
```

**Remarks:**

Returns the 'i-th' vertex of the specified spline.

**Parameters:**

**int poly**

The index into the splines list where **poly**  $\geq$  **0** and **poly**  $<$  **splineCount**.

**int i**

The index of the control point in the spline.

**Prototype:**

```
void SetVert(int poly, int i, const Point3 &xyz);
```

**Remarks:**

Sets the 'i-th' vertex of the specified spline.

**Parameters:**

**int poly**

The index into the splines list where **poly**  $\geq 0$  and **poly**  $<$  **splineCount**.

**int i**

The index of the vertex in the spline.

**const Point3 &xyz**

The point to set.

**Render / Select / Snap / HitTest**

**Prototype:**

```
void Render(GraphicsWindow *gw, Material *ma, RECT *rp, int  
compFlags, int numMat);
```

**Remarks:**

This is used internally to render the shape.

**Prototype:**

```
BOOL Select(GraphicsWindow *gw, Material *ma, HitRegion *hr,  
int abortOnHit = FALSE);
```

**Remarks:**

This is used internally to hit test the shape.

**Prototype:**

```
void Snap(GraphicsWindow *gw, SnapInfo *snap, IPoint2 *p,  
Matrix3 &tm);
```

**Remarks:**

This is used internally to snap to the shape.

**Prototype:**

```
void Snap(GraphicsWindow *gw, SnapInfo *snap, IPoint2 *p,  
Matrix3 &tm, DWORD flags);
```

**Remarks:**

This is used internally to snap to the shape.

**Prototype:**

```
BOOL SubObjectHitTest(GraphicsWindow *gw, Material *ma,  
HitRegion *hr,  
DWORD flags, SubShapeHitList& hitList );
```

**Remarks:**

This method is used internally to perform sub-object hit testing of the shape.

**Prototype:**

```
float FindSegmentPoint(int poly, int segment, GraphicsWindow  
*gw,  
Material *ma, HitRegion *hr, int ptype = PARAM_SIMPLE);
```

**Remarks:**

This is used for hit testing. This method returns a value from 0.0 to 1.0 that tells you where a hit was found on a particular segment. Sample code that uses this is the edit spline modifier in `\MAXSDK\SAMPLES\MODIFIERS\EDITSPL.H`. This allows a refinement of the hit testing.

**Parameters:**

**int poly**

The index of the spline.

**int segment**

The index of the segment.

**GraphicsWindow \*gw**

The graphics window where the hit test was done.

**Material \*ma**

The list of materials.

**HitRegion \*hr**

The hit region. See [Class HitRegion](#).

**int ptype = PARAM\_SIMPLE**

This parameter is available in release 4.0 and later only.

This allows the caller to get the location on the segment in either parameter

space or normalized distance space. Both return values of 0-1. For proper backward compatibility, the default parameter type is **PARAM\_SIMPLE**. The other option is **PARAM\_NORMALIZED**.

## **Bounding Box / Transform() / InvalidateGeomCache() / InvalidateCapCache() / FreeAll()**

### **Prototype:**

```
void BuildBoundingBox(void);
```

### **Remarks:**

Computes the bounding box of the splines making up this bezier shape object. The result is stored in the **bdgBox** data member.

### **Prototype:**

```
Box3 GetBoundingBox(Matrix3 *tm=NULL);
```

### **Remarks:**

Returns the bounding box of the splines making up this bezier shape object. The optional TM allows the box to be calculated in any space.

### **Parameters:**

**Matrix3 \*tm=NULL**

The matrix to transform the points before computing the bounding box.

### **Prototype:**

```
void GetDeformBBox(TimeValue t, Box3& box, Matrix3 *tm, BOOL useSel );
```

### **Remarks:**

Computes the bounding box of this shape.

### **Parameters:**

**TimeValue t**

This parameter is not used.

**Box3& box**

The result is stored here.

**Matrix3 \*tm**

The points of each spline in this shape are deformed using this matrix.

**BOOL useSel**

If TRUE the box is computed about the selected vertices only; otherwise all points.

**Prototype:**

```
void Transform(Matrix3 &tm);
```

**Remarks:**

Transforms the points of each poly in the shape by the matrix passed.

**Parameters:****Matrix3 &tm**

The transformation matrix.

**Prototype:**

```
void InvalidateGeomCache();
```

**Remarks:**

This method should be called when a shape changes. It invalidates the caches of the shape. This resets the bounding box, and removes the hierarchy, cap and shape caches.

**Prototype:**

```
void InvalidateCapCache();
```

**Remarks:**

This method is used internally.

**Prototype:**

```
void FreeAll();
```

**Remarks:**

Frees everything from the shape.

**Data Flow Evaluation**

**Prototype:**

**void ShallowCopy(BezierShape \*ashape, ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation

**Prototype:**

**void DeepCopy(BezierShape \*ashape, ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation

**Prototype:**

**void NewAndCopyChannels(ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation

**Prototype:**

**void FreeChannels(ULONG\_PTR channels, int zeroOthers=1);**

**Remarks:**

This method is used internally in data flow evaluation

## **Display Flag Access**

**Prototype:**

**void SetDispFlag(DWORD f);**

**Remarks:**

Sets the state of the specified display flags.

**Parameters:**

**DWORD f**

The flags to set. See [List of BezierShape Display Flags](#).

**Prototype:**

**DWORD GetDispFlag(DWORD f);**

**Remarks:**

Returns the state of the specified display flags.

**Parameters:****DWORD f**

The flags to get. See [List of BezierShape Display Flags](#).

**Prototype:****void ClearDispFlag(DWORD f);****Remarks:**

Clears the specified display flags.

**Parameters:****DWORD f**

The flags to clear. See [List of BezierShape Display Flags](#).

**Selection / UpdateSels()****Prototype:****BitArray VertexTempSel(int poly, int level = -1);****Remarks:**

Constructs a vertex selection list based on the current selection level for the specified spline. For example if the selection level is at object level all the bits are set. If the selection level is at vertex level only the selected vertex bits are set. See [Class BitArray](#).

**Parameters:****int poly**

The index into the **splines** data member.

**int level = -1**

This allows a selection level to be optionally specified. For example if the selection level is at object level all the bits are set. If the selection level is at vertex level only the selected vertex bits are set. One of the following values:

**SHAPE\_OBJECT****SHAPE\_SPLINE****SHAPE\_SEGMENT**

## SHAPE\_VERTEX

### Prototype:

**BitArray VertexTempSelAll(int poly = -1, BOOL includeVecs = FALSE, int level = 0, BOOL forceSel = FALSE);**

### Remarks:

This method looks at the selection state for the selection level and sets the appropriate bits based on the selection set **unless forceSel** is TRUE. If **forceSel** is TRUE, it acts as if every item in the selection set for the specified spline was set. It's an easy way to select entire splines.

See [Class BitArray](#).

### Parameters:

**int poly = -1**

The index into the **splines** data member (-1 means use all).

**BOOL includeVecs = FALSE**

If set, this method will set the bits for vectors associated with selected knots.

**int level = 0**

This allows a selection level to be optionally specified. For example if the selection level is at object level all the bits are set. If the selection level is at vertex level only the selected vertex bits are set. One of the following values:

**SHAPE\_OBJECT**

**SHAPE\_SPLINE**

**SHAPE\_SEGMENT**

**SHAPE\_VERTEX**

**BOOL forceSel = FALSE**

If TRUE, it acts as if every item in the selection set for the specified spline was set. It's an easy way to select entire splines. For example, selecting spline 2 regardless of the spline-level selection state:

**VertexTempSelAll(2, FALSE, SHAPE\_SPLINE, TRUE);**

This just builds the selection set as if spline 2 was completely selected.

Otherwise, the spline bits would have only been selected if `BezierShape::polySel.sel[2]` was set.

Incidentally, the following calls will do exactly the same thing:

**VertexTempSelAll(2, FALSE, SHAPE\_VERTEX, TRUE);**  
**VertexTempSelAll(2, FALSE, SHAPE\_SEGMENT, TRUE);**

For what it's worth, the SHAPE\_SPLINE version is the most efficient.

**Prototype:**

**void UpdateSels(BOOL save = FALSE);**

**Remarks:**

This is a very important call to make. When you are done adding polygons to the shape you should call this method. This method updates a set of embedded selection set data within the shape. This selection set data tells what polygons are selected, what segments are selected, and what control points are selected. This resets the sizes of the selection bit arrays for this shape.

**Parameters:**

**BOOL save = FALSE**

This parameter is available in release 4.0 and later only.

This optional parameter preserves the selection set information when set to TRUE, and erases it when set to FALSE. The default, FALSE, is present for backward compatibility, where there was no argument.

**Load() / Save()**

**Prototype:**

**IOResult Save(ISave\* isave);**

**Remarks:**

Saves the shape data to the .MAX file.

**Prototype:**

**IOResult Load(ILoad\* iload);**

**Remarks:**

Loads the shape data from the .MAX file.

**SplineCount() / GetSpline() / NewSpline() / AddSpline() /  
DeleteSpline() / InsertSpline() / NewShape()**

**Prototype:**

**inline int SplineCount();**

**Remarks:**

Returns the number of splines in this shape.

**Prototype:**

**Spline3D\* GetSpline(int index);**

**Remarks:**

Returns a pointer to the spline specified by the index passed.

**Parameters:**

**int index**

Specifies which spline to return. This is an index into the **splines** data member.

**Prototype:**

**Spline3D\* NewSpline(int itype = KTYPE\_CORNER,int dtype = KTYPE\_BEZIER,  
int ptype = PARM\_UNIFORM);**

**Remarks:**

Creates and adds an empty spline to the shape.

**Parameters:**

**int itype = KTYPE\_CORNER**

The initial knot type you get when you click and release the mouse during spline creation. See [List of Knot Types](#).

**int dtype = KTYPE\_BEZIER**

The drag knot type used when you click and drag to create a vertex during spline creation. See [List of Knot Types](#).

**int ptype = PARM\_UNIFORM**

This parameter is not used. Let it default to **PARM\_UNIFORM**.

**Return Value:**

A pointer to the newly created spline.

**Prototype:**

**Spline3D\* AddSpline(Spline3D\* spline);**

**Remarks:**

Add an existing spline to this shape as the last one in the list. Note that this copies only the pointer, it **does not** copy the entire spline, so **do not** delete the spline anywhere else. This will do it when it's done with it.

**Parameters:**

**Spline3D\* spline**

The spline to add.

**Return Value:**

A pointer to the spline passed or NULL if the call failed.

**Prototype:**

**int DeleteSpline(int index);**

**Remarks:**

Deletes the specified spline.

**Parameters:**

**int index**

The index into the splines data member.

**Return Value:**

Nonzero on success; otherwise zero.

**Prototype:**

**int InsertSpline(Spline3D\* spline, int index);**

**Remarks:**

Inserts the specified spline into the spline list at the location passed.

**Parameters:**

**Spline3D\* spline**

The spline to add.

**int index**

The index in to the splines data member indicating where to insert the spline.

**Return Value:**

Nonzero on success; otherwise zero.

**Prototype:**

**void NewShape();**

**Remarks:**

This method deletes every spline in the shape.

**Prototype:**

**void AddAndWeld(BezierShape &from, float weldThreshold);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is used for adding the splines from one BezierShape to another, with a weld threshold that will weld endpoints of the new splines onto endpoints of existing splines. Calling this method will cause the splines of the "from" shape to be added to those of the shape. If any endpoints in the "from" shape are within the specified weld threshold, they will be automatically welded.

**Parameters:**

**BezierShape &from**

The shape whose splines are added.

**float weldThreshold**

The endpoint weld threshold.

## **GetNumVerts() / GetNumSegs()**

**Prototype:**

**int GetNumVerts();**

**Remarks:**

Returns the total number of vertices in the entire shape.

**Prototype:**

**int GetNumSegs();**

**Remarks:**

Returns the total number of segments in the entire shape.

## **Get/SetClosures()**

### **Prototype:**

**void GetClosures(BitArray& array);**

### **Remarks:**

For each spline in this shape, this method sets the corresponding bit if the shape is closed and clears the bit if the shape is open.

### **Parameters:**

**BitArray& array**

The BitArray to update.

### **Prototype:**

**void SetClosures(BitArray& array);**

### **Remarks:**

Sets the closed state of each spline in this shape based on the BitArray passed.

### **Parameters:**

**BitArray& array**

Indicates which shapes should be closed: 1 = closed; 0 = open.

## **Shape Hierarchy / Reverse() / MakeFirst()**

### **Prototype:**

**ShapeHierarchy &OrganizeCurves(TimeValue t, ShapeHierarchy \*hier = NULL);**

### **Remarks:**

This method looks at the shape organization, and puts together a shape hierarchy. This provides information on how the shapes are nested. For example on a donut object with two circles, this method determines which circle is inside the other one.

### **Parameters:**

**TimeValue t**

This parameter is not used.

**ShapeHierarchy \*hier = NULL**

If non-NULL the result is store here (in addition to being returned). See [Class ShapeHierarchy](#).

**Return Value:**

The result is returned here.

**Prototype:**

**void Reverse(int poly, BOOL keepZero = FALSE);**

**Remarks:**

Reverses the spline whose index is passed.

**Parameters:**

**int poly**

The spline to reverse.

**BOOL keepZero = FALSE**

This optional parameter is available in release 2.0 and later only.

This parameter defaults to FALSE in order to retain backwards compatibility. Setting it to TRUE insures that a closed spline will have the same vertex as its first point when it is reversed. The parameter is ignored on open splines.

**Prototype:**

**void Reverse(BitArray &reverse, BOOL keepZero = FALSE);**

**Remarks:**

Reverses the splines of this shape if the corresponding bit in **reverse** is set.

**Parameters:**

**BitArray &reverse**

If the bit is set the spline is reversed; otherwise it is left alone.

**BOOL keepZero = FALSE**

This optional parameter is available in release 2.0 and later only.

This parameter defaults to FALSE in order to retain backwards compatibility. Setting it to TRUE insures that a closed spline will have the same vertex as its first point when it is reversed. The parameter is ignored on open splines.

**Prototype:**

```
void MakeFirst(int poly, int vertex);
```

**Remarks:**

Sets the specified vertex of the specified poly as the first vertex. On an open polygon this has to be one of the end control points. On a closed shape it doesn't matter.

**Parameters:**

**int poly**

The poly to update.

**int vertex**

The vertex to make first.

**MakePolyShape() / ReadyCachedPolyShape()****Prototype:**

```
void MakePolyShape(PolyShape &pshp, int steps = -1, BOOL  
optimize = FALSE);
```

**Remarks:**

Makes a **PolyShape** from this shape.

**Parameters:**

**PolyShape &pshp**

The results are stored here.

**int steps = -1**

The number of steps between knots in the spline.

**BOOL optimize = FALSE**

If TRUE, linear segments between control points in the spline will not generate steps in between. It will just be one line segment.

**Prototype:**

```
void ReadyCachedPolyShape();
```

**Remarks:**

This method is used internally.

## Capping Methods

### Prototype:

**int MakeCap(TimeValue t, MeshCapInfo &capInfo, int capType);**

### Remarks:

This method is passed a capping information structure and it will compute the information it needs to make a mesh cap for this shape.

### Parameters:

#### **TimeValue t**

This parameter is not used.

#### **MeshCapInfo &capInfo**

The capping information. See [Class MeshCapInfo](#).

#### **int capType**

See [List of Shape Capping Types](#).

### Return Value:

Nonzero if the method succeeded; otherwise zero.

### Prototype:

**int MakeCap(TimeValue t, PatchCapInfo &capInfo);**

### Remarks:

This method is passed a capping information structure and it will compute the information it needs to make a patch cap for this shape.

### Parameters:

#### **TimeValue t**

This parameter is not used.

#### **PatchCapInfo &capInfo**

The capping information. See [Class PatchCapInfo](#).

### Return Value:

Nonzero if the method succeeded; otherwise zero.

### Prototype:

**int ReadyPatchCap();**

**Remarks:**

This method is used internally by the BezierShape code. When you call **BezierShape::MakeCap(TimeValue t, PatchCapInfo &capInfo)**, if the patch cap data isn't cached, this method is called to build it. Calling it is not normally necessary, or a good idea -- it builds the PatchCapInfo data structure regardless of whether it's cached or not. Just call the **MakeCap()** method to get the cap information and the caching is done automatically.

**Prototype:**

**void CopyShapeDataFrom(BezierShape &fromShape);**

**Remarks:**

This method copies the shapes, selection sets and any caches from the source object. It does **not** copy selection level, or display information.

**Parameters:**

**BezierShape &fromShape**

The shape to copy from.

**Prototype:**

**void PrepVertBaseIndex();**

**Remarks:**

This method is used internally.

**Prototype:**

**int GetVertIndex(int poly, int vert);**

**Remarks:**

This method provides an easy way to derive a simple index for any vertex in any spline in the shape.

**Parameters:**

**int poly**

The zero based index of the spline.

**int vert**

The zero based index of the vertex.

**Return Value:**

A zero based index for vertex.

**Prototype:**

```
void GetPolyAndVert(int index, int &polyOut, int &vertOut);
```

**Remarks:**

This method takes a vertex index and turns it back into a poly / vertex pair (see **GetVertIndex()** above).

**Parameters:**

**int index**

The input index.

**int &polyOut**

The output poly.

**int &vertOut**

The output vertex.

**Prototype:**

```
int GetTotalVerts();
```

**Remarks:**

Returns the total number of vertices in the shape.

**Prototype:**

```
void PrepKnotBaseIndex();
```

**Remarks:**

This method is used internally, automatically.

**Prototype:**

```
int GetKnotIndex(int poly, int knot);
```

**Remarks:**

This method returns an index for any knot in any spline in the shape.

**Parameters:**

**int poly**

The input poly number.

**int knot**

The input knot number.

**Return Value:**

The zero based index of the knot.

**Prototype:**

**void GetPolyAndKnot(int index, int &polyOut, int &knotOut);**

**Remarks:**

This method computes a poly / knot pair from an index (see **GetKnotIndex()** above).

**Parameters:**

**int index**

The input knot index.

**int &polyOut**

The output index of the poly it is a part of.

**int &knotOut**

The output knot number.

**Prototype:**

**int GetTotalKnots();**

**Remarks:**

Returns the total number of knots in the shape.

**Prototype:**

**BOOL DeleteSelVerts(int poly);**

**Remarks:**

Deletes the selected vertices for the specified poly in the shape.

**Parameters:**

**int poly**

The zero based index of the polygon.

**Return Value:**

TRUE if any were deleted; FALSE if none were deleted.

**Prototype:**

**BOOL DeleteSelSegs(int poly);**

**Remarks:**

Deletes the selected polygons for the specified poly in the shape.

**Parameters:**

**int poly**

The zero based index of the polygon.

**Return Value:**

TRUE if any were deleted; FALSE if none were deleted.

**Prototype:**

**BOOL DeleteSelectedVerts();**

**Remarks:**

Deletes the selected vertices for all polys in the shape.

**Return Value:**

TRUE if any were deleted; FALSE if none were deleted.

**Prototype:**

**BOOL DeleteSelectedSegs();**

**Remarks:**

Deletes the selected segments for all polys in the shape.

**Return Value:**

TRUE if any were deleted; FALSE if none were deleted.

**Prototype:**

**BOOL DeleteSelectedPolys();**

**Remarks:**

Deletes the selected polygons for all polys in the shape.

**Return Value:**

TRUE if any were deleted; FALSE if none were deleted.

**Prototype:**

**BOOL CloneSelectedParts(BOOL reverse=FALSE);**

**Remarks:**

Copies the selected geometry (segments or polys), reversing if needed.

**Parameters:**

**BOOL reverse=FALSE**

TRUE to reverse; FALSE to leave alone.

**Return Value:**

Returns TRUE if anything was copied.

**Prototype:**

**BOOL RecordTopologyTags(int channel=0);**

**Remarks:**

Tags the points in the spline components to record the topology of the shape. (This stores identifying values in the Spline3D's Knot::aux fields for each control point). This info can be used after topology-changing operations to remap information tied to control points.

**Parameters:**

**int channel=0**

This parameter is available in release 3.0 and later only.

Specifies which auxiliary channel. One of the following values:

**0**=aux2

**1**=aux3

**Return Value:**

Returns FALSE if > 32767 knots or polys (it can't record that many).

**Prototype:**

**Point3 InterpCurve3D(int poly, float param, int ptype=PARAM\_SIMPLE);**

**Remarks:**

This method returns a point interpolated on the specified spline on the entire curve. This method returns the point but you don't know which segment the point falls on. See method **InterpPiece3D()**.

**Parameters:**

**int poly**

The zero based index of the spline.

**float param**

The position along the curve to return where 0 is the start and 1 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The interpolated point on the curve.

**Prototype:**

**Point3 TangentCurve3D(int poly, float param, int ptype=PARAM\_SIMPLE);**

**Remarks:**

This method returns a tangent vector interpolated on the entire curve of the specified spline. Also see method **TangentPiece3D()**.

**Parameters:**

**int poly**

The zero based index of the spline.

**float param**

The position along the curve to return where 0 is the start and 1 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The tangent vector

**Prototype:**

**Point3 InterpPiece3D(int poly, int piece, float param, int ptype=PARAM\_SIMPLE);**

**Remarks:**

This method returns the interpolated point along the specified sub-curve (segment) for the specified spline. For example consider a shape that is a single circle with four knots. If you called this method with curve=0 and piece=0 and param=0.0 you'd get back the point at knot 0. If you passed the same parameters except param=1.0 you'd get back the point at knot 1.

**Parameters:**

**int poly**

The zero based index of the spline.

**int piece**

The sub-curve (segment) to evaluate.

**float param**

The position along the curve to return where 0 is the start and 1 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Return Value:**

The point in world space.

**Prototype:**

**Point3 TangentPiece3D(int poly, int piece, float param, int ptype=PARAM\_SIMPLE);**

**Remarks:**

Returns the tangent vector of the specified spline on a sub-curve at the specified 'distance' along the curve.

**Parameters:**

**int poly**

The zero based index of the spline.

**int piece**

The sub-curve (segment) to evaluate.

**float param**

The position along the curve to return where 0 is the start and 1 is the end.

**int ptype=PARAM\_SIMPLE**

The parameter type for spline interpolation. See [List of Parameter Types for Shape Interpolation](#).

**Prototype:**

**float LengthOfCurve(int poly);**

**Remarks:**

Returns the length of the specified spline.

**Parameters:**

**int poly**

The index of the spline to check.

**Prototype:**

**MtlID GetMatID(int poly, int piece);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the material ID for the specified spline and segment of this shape.

**Parameters:**

**int poly**

The zero based index of the spline.

**int piece**

The zero based index of the segment of the spline.

**Prototype:**

**void GetTopology(BezierShapeTopology &topo);**

**Remarks:**

This method is available in release 3.0 and later only.

Retrieves information on the shape topology.

**Parameters:**

**BezierShapeTopology &topo**

The object which is updated with the shape data. See [Class](#)

[BezierShapeTopology](#).

**Prototype:**

**BOOL PerformTrimOrExtend(IObjParam \*ip, ViewExp \*vpt, ShapeHitData \*hit, IPoint2 &m, ShapeContextCallback &cb, int trimType, int trimInfinite);**

**Remarks:**

This method is available in release 3.0 and later only.

This method provides a way for a BezierShape to trim and extend splines.

This method is meant to perform as part of a mouse-centered operation -- Click on the part of the spline and pass the hitrecord, viewport and mouse point to the trim function.

**Parameters:**

**IObjParam \*ip**

The interface pointer. See [Class Interface](#).

**ViewExp \*vpt**

The viewport the user clicked in. See [Class ViewExp](#).

**ShapeHitData \*hit**

The hit record for the selection. See [Class ShapeHitData](#).

**IPoint2 &m**

The point the user clicked on in the viewport. See [Class IPoint2](#).

**ShapeContextCallback &cb**

The callback object. See [Class ShapeContextCallback](#).

**int trimType**

Specifies if the operation is a Trim or an Extend. One of the following values:

**SHAPE\_TRIM**

**SHAPE\_EXTEND**

**int trimInfinite**

This is set to TRUE for infinite projections.

**Return Value:**

This returns TRUE if the trim or extend was performed.

**Prototype:**

**void BindKnot(BOOL isEnd, int segIndex, int splineSegID, int splinePointID);**

**Remarks:**

This method is available in release 3.0 and later only.

This method binds a knot to a segment. A bind acts as a constraint, it constrains the first point or the end point of a spline to the mid point of a segment.

**Parameters:**

**BOOL isEnd**

Specifies whether the first or last point is bound. TRUE for the end; FALSE for the start.

**int segIndex**

The index of the segment to be bound to.

**int splineSegID**

The index of the spline that contains the segment.

**int splinePointID**

The index of spline that is being bound.

**Prototype:**

**void UnbindKnot(int splineID, BOOL isEnd);**

**Remarks:**

This method is available in release 3.0 and later only.

This method unbinds the specified spline.

**Parameters:**

**int splineID**

The index of spline that is being bound.

**BOOL isEnd**

Specifies whether the first or last point is unbound. TRUE for the end; FALSE for the start.

**Prototype:**

**void UpdateBindList(BOOL useGeomTracking=FALSE);**

**Remarks:**

This method is available in release 3.0 and later only.

This method needs to be called when the topology changes to update the bind list.

**Parameters:****BOOL useGeomTracking=FALSE**

This parameter is available in release 4.0 and later only.

This allows the update to be dependant on previous recorded topology or geometry. If this flag is set to TRUE it uses the old method of using the geometry to rebuild the binds, else it uses the new method which uses the aux flags of the splines which store the old topology indices. Normally this will be FALSE since most of the times the topology tracking is more accurate, it should be set to TRUE when there is no initial topology to work from for instance when attaching or detaching geometry.

**Prototype:****BOOL HideSelectedSegs();****Remarks:**

This method is available in release 3.0 and later only.

Hides the selected segments.

**Return Value:**

TRUE if any were hidden. FALSE if none were hidden.

**Prototype:****BOOL HideSelectedVerts();****Remarks:**

This method is available in release 3.0 and later only.

Hides the segments attached to the selected vertices.

**Return Value:**

TRUE if any were hidden. FALSE if none were hidden.

**Prototype:****BOOL HideSelectedSplines();**

**Remarks:**

This method is available in release 3.0 and later only.

Hides the segments attached to the selected splines.

**Return Value:**

TRUE if any were hidden. FALSE if none were hidden.

**Prototype:**

**BOOL UnhideSegs();**

**Remarks:**

This method is available in release 3.0 and later only.

Unhides all the segments in the shape.

**Return Value:**

TRUE if any were unhidden. FALSE if none were unhidden.

**Operators:****Prototype:**

**BezierShape& operator+=(BezierShape& from);**

**Remarks:**

This lets you add another shape to this one.

**Prototype:**

**BezierShape& operator=(BezierShape& fromShape);**

**Remarks:**

Assignment operator.

**Prototype:**

**BezierShape& operator=(PolyShape& fromShape);**

**Remarks:**

Assignment operator.

## Class PolyShape

See Also: [Class ShapeObject](#), [Class PolyLine](#), [Class ShapeVSel](#), [Class ShapeSSel](#), [Class ShapePSel](#), [Class ShapeHierarchy](#).

class PolyShape

### Description:

A multi-polygon shape class. This class is used in the caching of bezier shapes. This is used for doing a one time interpolation of a bezier shape into a form that is the same shape but doesn't require any further interpolation. In this way the system can do the complex calculations once, store the shape into this **PolyShape** representation, and not have to go through the cubic spline calculations to figure out where the points are in the future. This class maintains an array of **PolyLines**.

This is used for example in the Extrude modifier. The first thing it does is generate a **PolyShape** from the bezier shape it is extruding. This **PolyShape** is then used to generate the mesh.

### Method Groups:

The following hyperlinks take you to the start of groups of related methods within the class:

[Constructors / Destructor / Init\(\) / GetMatID\(\)](#)

[SetNumLines\(\) / NewShape\(\) / NewLine\(\)](#)

[Append / Insert / Delete / Reverse\(\)](#)

[Bounding Box / InvalidateGeomCache\(\) / InvalidateCapCache\(\)](#)

[Render / Select / Snap / Transform](#)

[Capping Methods](#)

[Selection / UpdateSels\(\)](#)

[Shape Hierarchy](#)

[Data Flow Evaluation](#)

[Dump\(\) / Load\(\) / Save\(\)](#)

[Operators](#)

### Data Members:

public:

**int numLines;**

The number of lines in the polyshape.

**PolyLine \*lines;**

The array of lines.

**DWORD flags;**

These are not currently used.

**Box3 bdgBox;**

Stores the bounding box surrounding each **PolyLine** in the shape.

**ShapeVSel vertSel;**

The selected vertices.

**ShapeSSel segSel;**

The selected segments.

**ShapePSel polySel;**

The selected polygons.

**DWORD selLevel;**

The selection level. One of the following values:

**SHAPE\_OBJECT** - Object level selection.

**SHAPE\_SPLINE** - Spline level selection.

**SHAPE\_SEGMENT** - Segment level selection.

**SHAPE\_VERTEX** - Vertex level selection.

**DWORD dispFlags;**

Display attribute flags.

**DISP\_VERTTICKS** - Display vertices as ticks.

**DISP\_SELVERTS** - Display selected vertices.

**DISP\_SELSEGMENTS** - Display selected segments.

**DISP\_SELPOLYS** - Display selected polygons.

The following capping and cache information is used internally:

**MeshCapInfo morphCap;**

The morph cap information.

**BOOL morphCapCacheValid;**

Determines if the morph cap is valid.

**MeshCapInfo gridCap;**

The grid cap information.

**BOOL gridCapCacheValid;**

Determines if the grid cap is valid.

**PatchCapInfo patchCap;**

The patch cap information.

**BOOL patchCapCacheValid;**

Determines if the patch cap is valid.

**ShapeHierarchy cachedHier;**

Hierarchy cache.

**BOOL hierCacheValid;**

Determines if the hierarchy cache is valid.

**Methods:**

**Constructors / Destructor / Init() / GetMatID**

**Prototype:**

**PolyShape();**

**Remarks:**

Constructor. The number of lines is set to 0, the lines array is set to NULL, the flags and display flags are set to 0, the selection level is set to SHAPE\_OBJECT, the geometry cache is invalidated, and the masterObject is set to NULL.

**Prototype:**

**PolyShape(PolyShape& from);**

**Remarks:**

Constructor. The **PolyShape** is initialized from the specified **PolyShape**.

**Prototype:**

**~PolyShape();**

**Remarks:**

Destructor.

**Prototype:**

**void Init();**

**Remarks:**

**PolyShape::Init()** is a special version used by the constructors and should not be called by plug-in developers. If you need to clear out a **PolyShape** use **NewShape()** described below.

**Prototype:**

**MtlID GetMatID(int poly, int piece);**

**Remarks:**

This method is available in release 3.0 and later only.

Returns the material ID for the specified segment of the specified poly.

**Parameters:**

**int poly**

The zero based index of the poly.

**int piece**

The zero based index of the segment.

**SetNumLines() / NewShape() / NewLine()****Prototype:**

**BOOL SetNumLines(int count, BOOL keep = TRUE);**

**Remarks:**

Sets the number of polygons used by the poly shape.

**Parameters:**

**int count**

The number of lines.

**BOOL keep = TRUE**

If TRUE any old lines are copied to the new storage; otherwise they are freed.

**Return Value:**

TRUE if the number of lines were set; otherwise FALSE.

**Prototype:**

**void NewShape();**

**Remarks:**

This deletes all the lines from the **PolyShape** and clears the shape out. Use this method instead of **Init()** above.

**Prototype:**

**PolyLine\* NewLine();**

**Remarks:**

Creates a new **PolyLine** and appends it to the end of the list of lines maintained by this **PolyShape**.

**Return Value:**

The address of the newly added line.

**Append / Insert / Delete / Reverse()****Prototype:**

**void Append(PolyLine &l);**

**Remarks:**

Appends the specified **PolyLine** to the end of the lines list.

**Parameters:**

**PolyLine &l**

The **PolyLine** to append.

**Prototype:**

**void Insert(int where, PolyLine& l);**

**Remarks:**

Inserts the specified **PolyLine** at the location passed.

**Parameters:**

**int where**

The index into the lines list specifying where to insert the **PolyLine**.

**PolyLine& l**

The **PolyLine** to insert.

**Prototype:**

**void Delete(int where);**

**Remarks:**

Deletes the specified **PolyLine** from the lines list.

**Parameters:**

**int where**

The index into the lines list specifying which line to delete.

**Prototype:**

**void Reverse(int poly, BOOL keepZero=FALSE);**

**Remarks:**

Reverses the **PolyLine** whose index is passed.

**Parameters:**

**int poly**

The spline to reverse.

**BOOL keepZero=FALSE**

This should normally be passed as TRUE. If TRUE, and the polyline is closed, this method will make sure that vertex zero is the same on the reversed version as on the non-reversed version. Otherwise if passed as FALSE the last vertex becomes the first vertex, and the first vertex becomes the last. This is an important distinction for the loftter because it always wants vertex zero to be in the same place.

**Prototype:**

**void Reverse(BitArray &reverse, BOOL keepZero=FALSE);**

**Remarks:**

Reverses the splines of this shape if the corresponding bit in **reverse** is set. The BitArray has one bit for each polyline.

**Parameters:**

**BitArray &reverse**

If the bit is set the spline is reversed; otherwise it is left alone.

**BOOL keepZero=FALSE**

This should normally be passed as TRUE. If TRUE, and the polyline is closed, this method will make sure that vertex zero is the same on the reversed version as on the non-reversed version. Otherwise if passed as FALSE the last vertex becomes the first vertex, and the first vertex becomes the last. This is an important distinction for the lofter because it always wants vertex zero to be in the same place.

## **Bounding Box / InvalidateGeomCache() / InvalidateCapCache()**

### **Prototype:**

```
void BuildBoundingBox();
```

### **Remarks:**

Builds a bounding box surrounding every line in the lines list. The bounding box is returned through the **bdgBox** data member.

### **Prototype:**

```
Box3 GetBoundingBox(Matrix3 *tm=NULL);
```

### **Remarks:**

Returns the bounding box of the **PolyLines** in this **PolyShape**. The optional TM allows the box to be calculated in any space.

### **Parameters:**

**Matrix3 \*tm=NULL**

The points of the **PolyLines** in this **PolyShape** are transformed by this matrix prior to the bounding box computations.

### **Prototype:**

```
void GetDeformBBox(TimeValue t, Box3& box, Matrix3 *tm,  
BOOL useSel);
```

### **Remarks:**

Computes the bounding box of this **PolyShape**.

### **Parameters:**

**TimeValue t**

The time at which to evaluate the bounding box.

**Box3& box**

The result is stored here.

**Matrix3 \*tm**

The points of each **PolyLine** in this **PolyShape** are deformed using this matrix.

**BOOL useSel**

If TRUE the box is computed about the selected vertices only; otherwise all points.

**Prototype:**

```
void InvalidateGeomCache(BOOL unused);
```

**Remarks:**

Invalidates the cache of each line in the **PolyShape**. The bounding box is set to empty. This method also invalidates the capping caches.

**Parameters:****BOOL unused**

This parameter is not used.

**Prototype:**

```
void InvalidateCapCache();
```

**Remarks:**

Invalidates the morph, and grid cap caches.

**Render / Select / Snap / Transform****Prototype:**

```
void Render(GraphicsWindow *gw, Material *ma, RECT *rp, int compFlags);
```

**Remarks:**

This method is used internally.

**Prototype:**

**BOOL Select(GraphicsWindow \*gw, Material \*ma, HitRegion \*hr,  
int abortOnHit = FALSE);**

**Remarks:**

This method is used internally.

**Prototype:**

**void Snap(GraphicsWindow \*gw, SnapInfo \*snap, IPoint2 \*p,  
Matrix3 &tm);**

**Remarks:**

This method is used internally.

**Prototype:**

**void Snap(GraphicsWindow \*gw, SnapInfo \*snap, IPoint2 \*p,  
Matrix3 &tm, DWORD flags);**

**Remarks:**

This method is used internally.

**Prototype:**

**void Transform(Matrix3 &tm);**

**Remarks:**

Transforms the vertices of each PolyLine in this PolyShape by the specified matrix.

**Parameters:**

**Matrix3 &tm**

The transformation matrix.

## **Capping Methods**

**Prototype:**

**int MakeCap(TimeValue t, MeshCapInfo &capInfo, int capType);**

**Remarks:**

This method may be called to fill in the MeshCapInfo passed with the

appropriate capping information. See [Working with Patches](#) for more details on capping. This method is used for meshes. The method below is used for patches.

**Parameters:**

**TimeValue t**

This should be passed as the current time. You may retrieve this using **Interface::GetTime()**. See [Class Interface](#).

**MeshCapInfo &capInfo**

This information is filled in by this method. See [Class MeshCapInfo](#).

**int capType**

The cap type. See [List of Shape Capping Types](#).

**Return Value:**

Nonzero if the cap info was set up successfully; otherwise zero.

**Prototype:**

**int MakeCap(TimeValue t, PatchCapInfo &capInfo);**

**Remarks:**

This method may be called to fill in the **PatchCapInfo** passed with the appropriate capping information. This method is used for patches. Note that it is generally not recommended to try to create patch caps from **PolyShapes**.

The patch cap uses bezier information, so it is much better to use a **BezierShape** to make a patch cap. It is very inefficient to do this with a **PolyShape**.

**Parameters:**

**TimeValue t**

This should be passed as the current time. You may retrieve this using **Interface::GetTime()**. See [Class Interface](#).

**PatchCapInfo &capInfo**

This information is filled in by this method. See [Class PatchCapInfo](#).

**Return Value:**

Nonzero if the cap info was set up successfully; otherwise zero.

**Prototype:**

**int Make3DSCap(MeshCapInfo &capInfo, DWORD options = 0);**

**Remarks:**

This method is used internally.

**Prototype:**

**int MakeGridCap(MeshCapInfo &capInfo);**

**Remarks:**

This method is used internally.

**Dump() / Load() / Save()**

**Prototype:**

**void Dump(TCHAR \*title = NULL);**

**Remarks:**

This may be called to display information about the PolyShape to the debug window via **DebugPrint()**. See [Debugging](#).

**Parameters:**

**TCHAR \*title = NULL**

A string to be display before the **PolyShape** data is displayed.

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

This method is used internally in saving to the MAX file.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

This method is used internally in loading from the MAX file.

## **Shape Hierarchy**

**Prototype:**

```
void UpdateCachedHierarchy();
```

**Remarks:**

This method is used internally.

**Prototype:**

```
ShapeHierarchy &OrganizeCurves(TimeValue t, ShapeHierarchy  
*hier = NULL);
```

**Remarks:**

This methods looks at the shape organization, and puts together a shape hierarchy. This provides information on how the shapes are nested. For example on a donut object with two circles, this method determines which circle is inside the other one.

**Parameters:**

**TimeValue t**

This parameter is not used.

**ShapeHierarchy \*hier = NULL**

If non-NULL the result is store here (in addition to being returned). See [Class ShapeHierarchy](#).

**Return Value:**

The result is store here.

**Selection / UpdateSels()****Prototype:**

```
BitArray VertexTempSel(int poly);
```

**Remarks:**

Constructs a vertex selection list based on the current selection level of the specified poly. For example if the selection level is at object level all the bits are set. If the selection level is at vertex level only the selected vertex bits are set. See [Class BitArray](#).

**Parameters:**

**int poly**

The poly whose selection level is constructed.

**Prototype:**

**void UpdateSels();**

**Remarks:**

This is a very important call to make. Whenever you have changed the **PolyShape**, for example after you are done adding polygons to the shape or have changed the number of points in a shape, you should call this method. This method updates a set of embedded selection set data within the shape. This selection set data tells what polygons are selected, what segments are selected, and what control points are selected. This resets the sizes of the selection bit arrays for this shape.

## **Data Flow Evaluation**

**Prototype:**

**void ShallowCopy(PolyShape \*ashape, ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation in the pipeline.

**Prototype:**

**void DeepCopy(PolyShape \*ashape, ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation in the pipeline.

**Prototype:**

**void NewAndCopyChannels(ULONG\_PTR channels);**

**Remarks:**

This method is used internally in data flow evaluation in the pipeline.

**Prototype:**

**void FreeChannels(ULONG\_PTR channels, int zeroOthers=1);**

**Remarks:**

This method is used internally in data flow evaluation in the pipeline.

## **Operators:**

### **Prototype:**

**PolyShape& operator=(PolyShape& from);**

### **Remarks:**

Assignment operator.

### **Prototype:**

**PolyShape& operator=(BezierShape& from);**

### **Remarks:**

Assignment operator. Note that this operator does not offer as much control as calling the method on the **BezierShape** itself named **MakePolyShape()**. That version allows you to specify the number of steps and an optimize parameter. These options are not available on this simple assignment operator.

# Class PolyLine

See Also: [Class PolyPt](#), [Class PolyShape](#).

class PolyLine

## Description:

This class describes a single polygon in a **PolyShape** using linear segments. All methods of this class are implemented by the system.

## Method Groups:

The following hyperlinks take you to the start of groups of related methods within the class:

[Constructors / Destructor / Init\(\) / GetMatID\(\)](#)  
[Opened/Closed/Self Intersects/Clockwise/Surrounds](#)  
[Point/SetNumPts\(\)/CurveLength\(\)](#)  
[SpliceLine\(\) / HitsSegment\(\) / HitsPolyLine\(\)](#)  
[Vertex / Segment counts](#)  
[Append / Insert / Delete / Reverse\(\)](#)  
[Bounding Box / InvalidateGeomCache\(\)](#)  
[Render / Select / Snap / Transform](#)  
[Interpolated Points / Tangent Vectors](#)  
[Capping Methods](#)  
[Dump\(\)](#)  
[Operators](#)

## Data Members:

public:

**int numPts;**

The number of points in the polyline.

**PolyPt \*pts;**

The points themselves.

**DWORD flags;**

PolyLine flags. One or more of the following values:

**POLYLINE\_CLOSED**

Indicates the polyline is closed.

**POLYLINE\_NO\_SELF\_INT**

Ignore self-intersections. This is used internally in the patch capping to indicate that this line always returns FALSE from the self intersects test. Normally this should not be set.

**Box3 bdgBox;**

The bounding box of the polyline.

**float cachedLength;**

The length of the polyline.

**float \*lengths;**

Cached lengths for each point

**float \*percents;**

Cached percentages for each point

**BOOL cacheValid;**

Indicates if the cache is valid.

**Methods:**

**Constructors / Destructor / Init / GetMatID()**

**Prototype:**

**PolyLine();**

**Remarks:**

Constructor. The number of points is set to zero, the points array is set to NULL, the flags are set to 0, the cache validity is set to FALSE and the bounding box is set to empty.

**Prototype:**

**PolyLine(PolyLine& from);**

**Remarks:**

Constructor. The **PolyLine** is initialized using **from**.

**Prototype:**

**~PolyLine();**

**Remarks:**

Destructor. The array of points is freed.

**Prototype:**

**void Init();**

**Remarks:**

Initializes the PolyLine. The **pts** array is freed, the number of points is set to 0, and the cache validity is set to FALSE.

**Prototype:**

**MtlID GetMatID(int segment);**

**Remarks:**

This method is available in release 3.0 and later only.  
Returns the material ID for the specified segment.

**Parameters:**

**int segment**

The zero based index of the segment.

**Opened / Closed / Self Intersects / Clockwise / Surrounds Point / SetNumPts()**

**Prototype:**

**void Close();**

**Remarks:**

Call this method to indicate the polyline is closed.

**Prototype:**

**BOOL IsClosed();**

**Remarks:**

Returns TRUE if the polyline is closed; otherwise FALSE.

**Prototype:**

**void Open();**

**Remarks:**

Call this method to indicate the polyline is open.

**Prototype:**

**BOOL IsOpen();**

**Remarks:**

Returns TRUE if the polyline is open; otherwise FALSE.

**Prototype:**

**void SetNoSelfInt();**

**Remarks:**

Call this method to set the self intersects flag. This is used in the patch capping to indicate that this line always returns FALSE from the self intersects test. Normally this should not be set.

**Prototype:**

**BOOL IsNoSelfInt()**

**Remarks:**

Returns TRUE if the no self intersect flag is set; otherwise FALSE.

**Prototype:**

**BOOL IsClockWise();**

**Remarks:**

Returns TRUE if the polyline is clockwise in the XY plane (it ignores Z); otherwise FALSE. If the **PolyLine** self intersects, the results from this method are meaningless.

**Prototype:**

**BOOL SelfIntersects(BOOL findAll = FALSE,  
IntersectionCallback3D \*cb = NULL);**

**Remarks:**

Returns TRUE if the polyline intersects itself in the XY plane (it ignores Z);

otherwise FALSE.

**Parameters:**

**BOOL findAll = FALSE**

TRUE to find all self intersections. FALSE to find only the first self intersection.

**IntersectionCallback3D \*cb = NULL**

A pointer to an IntersectionCallback3D class.

**Prototype:**

**float CurveLength();**

**Remarks:**

Returns the length of the **PolyLine**.

**Prototype:**

**BOOL SurroundsPoint(Point2 &point);**

**Remarks:**

Returns TRUE if the specified point is surrounded (contained within) this spline. This method should only be called on closed **PolyLines**.

**Parameters:**

**Point2 &point**

The point to check.

## **Vertex / Segment counts**

**Prototype:**

**int Verts()**

**Remarks:**

Returns the number of vertices (points) in the polyline.

**Prototype:**

**int Segments();**

**Remarks:**

Returns the number of segments (edges between vertices) of the polyline.

**Prototype:**

**BOOL SetNumPts(int count, BOOL keep = TRUE);**

**Remarks:**

Sets the number of points in the polyline.

**Parameters:**

**int count**

The number to set.

**BOOL keep = TRUE**

If TRUE any existing points are copied to the new array; otherwise they are freed.

**Return Value:**

TRUE if the number of points was set; otherwise FALSE.

**Append() / Insert() / Delete() / Reverse()**

**Prototype:**

**void Append(PolyPt& p);**

**Remarks:**

Appends the specified point to the polyline. This adds the point to the end of the points list.

**Parameters:**

**PolyPt& p**

The point to append.

**Prototype:**

**void Insert(int where, PolyPt& p);**

**Remarks:**

Inserts the specified point at the location passed.

**Parameters:**

**int where**

The **pts** array index indicating where to insert the point.

**PolyPt& p**

The point to insert.

**Prototype:**

**void Delete(int where);**

**Remarks:**

Deletes the specified point.

**Parameters:**

**int where**

The **pts** array index indicating which point to delete.

**Prototype:**

**void Reverse(BOOL keepZero=FALSE);**

**Remarks:**

Reverses the order of the points in the polyline.

**Parameters:**

**BOOL keepZero=FALSE**

This should normally be passed as TRUE. If TRUE, and the polyline is closed, this method will make sure that vertex zero is the same on the reversed version as on the non-reversed version. Otherwise if passed as FALSE the last vertex becomes the first vertex, and the first vertex becomes the last. This is an important distinction for the lofter because it always wants vertex zero to be in the same place.

## **Bounding Box / InvalidateGeomCache()**

**Prototype:**

**void BuildBoundingBox();**

**Remarks:**

Computes the bounding box of the polyline. The result is stored in the **bdgBox** data member.

**Prototype:**

**Box3 GetBoundingBox(Matrix3 \*tm=NULL);**

**Remarks:**

Returns the bound box of the polyline. The optional TM allows the box to be calculated in any space.

**Parameters:**

**Matrix3 \*tm=NULL**

The points of the polyline are multiplied by this matrix before the box computation.

**Prototype:**

**void InvalidateGeomCache();**

**Remarks:**

This method makes sure the **PolyLine** has flushed out any cached data it may have had. This resets the bounding box size and sets the cache validity to FALSE. This should be called when any points have been changed in the polyline.

## **Render / Select / Snap / Transform**

**Prototype:**

**void Render(GraphicsWindow \*gw, Material \*ma, RECT \*rp, int compFlags, int numMat);**

**Remarks:**

This method is used internally.

**Prototype:**

**void Render(GraphicsWindow \*gw, Material \*ma, in numMat, BOOL colorSegs, BitArray &segSel);**

**Remarks:**

This method is used internally.

**Prototype:**

**BOOL Select(GraphicsWindow \*gw, Material \*ma, HitRegion \*hr,  
int abortOnHit = FALSE);**

**Remarks:**

This method is used internally.

**Prototype:**

**void Snap(GraphicsWindow \*gw, SnapInfo \*snap, IPoint2 \*p,  
Matrix3 &tm, DWORD flags);**

**Remarks:**

This method is used internally.

**Prototype:**

**void Transform(Matrix3 &tm);**

**Remarks:**

Transforms the points of the polyline by the specified matrix.

**Parameters:**

**Matrix3 &tm**

The matrix to transform the points.

## **Dump()**

**Prototype:**

**void Dump(TCHAR \*title = NULL);**

**Remarks:**

You may call this method to dump the polyline structure via **DebugPrint()**.

See [Debugging](#).

**Parameters:**

**TCHAR \*title = NULL**

This title string is displayed using a **DebugPrint()** before the rest of the data.

## **Splice() / HitsSegment() / HitsPolyLine()**

**Prototype:**

**void SpliceLine(int where, PolyLine &source, int splicePoint);**

**Remarks:**

This method is used internally as part of the capping mechanism and should not be used.

**Prototype:**

**BOOL HitsSegment(Point2 p1, Point2 p2, BOOL findAll=FALSE, IntersectionCallback3D \*cb = NULL);**

**Remarks:**

Returns TRUE if the line segment defined between points **p1** and **p2** intersects this **PolyLine**; otherwise FALSE.

**Parameters:**

**Point2 p1, Point2 p2**

The endpoints of the line to check.

**BOOL findAll=FALSE**

TRUE to find all intersections. FALSE to find only the first intersection.

**IntersectionCallback3D \*cb = NULL**

A pointer to an IntersectionCallback3D class.

**Prototype:**

**BOOL HitsPolyLine(PolyLine &line, BOOL findAll=FALSE, IntersectionCallback3D \*cb = NULL);**

**Remarks:**

Returns TRUE if the specified **PolyLine** intersects this PolyLine; otherwise FALSE.

**Parameters:**

**PolyLine &line**

The PolyLine to check

**BOOL findAll=FALSE**

TRUE to find all intersections. FALSE to find only the first intersection.

**IntersectionCallback3D \*cb = NULL**

A pointer to an IntersectionCallback3D class.

## Capping Methods

### Prototype:

```
int Cap3DS(CapVert *capverts, MeshCapInfo &capInfo, DWORD options = 0);
```

### Remarks:

This method is used internally.

## Interpolated Points / Tangent Vectors

### Prototype:

```
Point3 InterpPiece3D(int segment, float t);
```

### Remarks:

This method returns a point interpolated on a segment between two points.

### Parameters:

**int segment**

The index of the segment to interpolate.

**float t**

A value in the range of 0.0 to 1.0. 0 is the first point and 1 is the second point.

### Return Value:

The interpolated point.

### Prototype:

```
Point3 InterpCurve3D(float u, int ptype=POLYSHP_INTERP_SIMPLE);
```

### Remarks:

This method returns a point interpolated on the entire curve. This method returns a point but you don't know which segment the point falls on.

### Parameters:

**float u**

A value in the range of 0.0 to 1.0 for the entire curve.

**int ptype=POLYSHP\_INTERP\_SIMPLE**

This parameter is available in release 2.0 and later only.

The parameter type for interpolation. The default value is based on segments (rather than the entire curve). See [List of Parameter Types for PolyLine Interpolation](#).

**Return Value:**

The interpolated point.

**Prototype:**

**Point3 TangentPiece3D(int segment, float t);**

**Remarks:**

This method returns a tangent vector interpolated on a segment between two points.

**Parameters:**

**int segment**

The index of the segment.

**float t**

A value in the range of 0.0 to 1.0. 0 is the first point and 1 is the second point.

**Return Value:**

The tangent vector.

**Prototype:**

**Point3 TangentCurve3D(float u, int ptype=POLYSHP\_INTERP\_SIMPLE);**

**Remarks:**

This method returns a tangent vector interpolated on the entire curve.

**Parameters:**

**float u**

A value in the range of 0.0 to 1.0 for the entire curve.

**int ptype=POLYSHP\_INTERP\_SIMPLE**

This parameter is available in release 2.0 and later only.

The parameter type for interpolation. The default value is based on segments

(rather than the entire curve). See [List of Parameter Types for PolyLine Interpolation](#).

**Return Value:**

The tangent vector.

**Operators:**

**Prototype:**

**PolyLine& operator=(PolyLine& from);**

**Remarks:**

Assignment operator.

**Prototype:**

**PolyLine& operator=(Spline3D& from);**

**Remarks:**

Assignment operator. This generates a polyline from the spline.

**Prototype:**

**PolyPt& operator[](int index);**

**Remarks:**

Array access operator. Returns the specified point in the **pts** array.

## Class INodeTab

See Also: [Class Interface](#), [Template Class Tab](#), [Class INode](#).

```
class INodeTab : public Tab<INode*>;
```

### **Description:**

This class is used to hold a list of **INode** pointers. Methods of the template class **Tab** are used to manipulate the table.

## Class NameEnumCallback

See Also: [Class Animatable](#), [Class NameTab](#).

class NameEnumCallback

### Description:

This class is the callback object passed to **Interface::EnumAuxFiles()** and to **Animatable::EnumAuxFiles()**. A developer derives a class from this class and implements the **RecordName()** method to store each name as it's passed to it. At the end of the **EnumAuxFiles()** processing, the table of names may be used. See [Class NameTab](#) for help storing the names.

### Methods:

### Prototype:

```
virtual void RecordName(TCHAR *name)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is used to record the name passed.

### Parameters:

**TCHAR \*name**

The name to store.

## List of EnumAuxFiles Flags

See Also: [Class Interface](#), [Class Animatable](#).

One or more of the following values:

### **FILE\_ENUM\_INACTIVE**

Enumerate inactive files. Inactive files are those that aren't being used currently. For instance, a texture map file that is present, but not activated in the materials editor user interface, is considered inactive.

### **FILE\_ENUM\_VP**

Enumerate video post files.

### **FILE\_ENUM\_RENDER**

Enumerate render files.

### **FILE\_ENUM\_ALL**

Enumerate ALL files (this is the same as:

**FILE\_ENUM\_INACTIVE|FILE\_ENUM\_VP|FILE\_ENUM\_RENDER|FILE\_ENUM\_MISSING\_ONLY**

Enumerate missing files only.

### **FILE\_ENUM\_1STSUB\_MISSING**

Just enumerate the first file named by an IFL (Image File List) if missing.

### **FILE\_ENUM\_DONT\_RECURSE**

Don't enumerate references.

### **FILE\_ENUM\_CHECK\_AWORK1**

Don't enumerate things with flag **A\_WORK1** set.

## Class TimeChangeCallback

See Also: [Class Interface](#).

class TimeChangeCallback

### Description:

A callback object passed to **Interface::RegisterTimeChangeCallback()**. The method **TimeChange()** is called every time the user changes the frame slider allowing the plug-in to respond.

### Methods:

### Prototype:

**virtual void TimeChanged(TimeValue t)=0;**

### Remarks:

Implemented by the Plug-In.

This method is called every time the user changes the frame slider.

### Parameters:

#### **TimeValue t**

The time the user has moved the frame slider to.

See Also: Methods **RegisterTimeChangeCallback()** and **UnRegisterTimeChangeCallback()** in [Class Interface](#).

# Class ViewportDisplayCallback

See Also: [Class Interface](#), [Class ViewExp](#), [Data Types](#).

```
class ViewportDisplayCallback
```

## Description:

This class is available in release 2.0 and later only.

This class is a callback object that enables plug-ins that aren't actually objects (such as utility plug-ins) to draw in the 3ds max viewports. See the methods in class **Interface** that register and unregister this callback object:

**Interface::RegisterViewportDisplayCallback()**, and  
**Interface::UnRegisterViewportDisplayCallback()**

Also see the method

**Interface::NotifyViewportDisplayCallbackChanged()**.

## Methods:

### Prototype:

```
virtual void Display(TimeValue t, ViewExp *vpt, int flags)=0;
```

### Remarks:

This method is called to allow the plug-in to draw in the viewports.

### Parameters:

**TimeValue t**

The current time when this method is called.

**ViewExp \*vpt**

An interface into the viewport.

**int flags**

These flags are used internally.

### Prototype:

```
virtual void GetViewportRect(TimeValue t, ViewExp *vpt, Rect *rect)=0;
```

### Remarks:

Retrieves the dimensions of the specified viewport given an **ViewExp**

interface to it.

**Parameters:**

**TimeValue t**

The time to get the viewport rectangle.

**ViewExp \*vpt**

Specifies which viewport

**Rect \*rect**

The rectangle is returned here.

**Prototype:**

**virtual BOOL Foreground()=0;**

**Remarks:**

This method should return TRUE if the object changes a lot or FALSE if it doesn't change very much. This method relates to the foreground/background display system used by 3ds max. Basically, items that change a lot are placed in the foreground buffer. Items that don't change much are placed in the background buffer and simply biltted to the dispaly. See the Advanced Topics section on [Foreground / Background Planes](#) for more details. Most plug-ins can simply return TRUE because they are not likely to be very heavyweight objects (they are usually just a gizmo or apparatus image) and can simply go into the foreground. On the other hande, some items, for instance the 3ds max home grid, don't change and can always go into the background. The home grid only changes when the view direction is changed in which case everything is redrawn.

# Class ExitMAXCallback

See Also: [Class Interface](#), [Structure NotifyInfo](#).

class ExitMAXCallback

## Description:

This class is available in release 2.0 and later only.

This class is a callback object that will get called before the program exits. All methods of this class are implemented by the plug-in.

## Methods:

### Prototype:

**virtual BOOL Exit(HWND hWnd)=0;**

### Remarks:

This method is called when 3ds max is about to exit.

### Parameters:

**HWND hWnd**

The main 3ds max window handle.

### Return Value:

TRUE to exit; FALSE to abort the exit and remain in 3ds max.

## Class AxisChangeCallback

See Also: [Class Interface](#).

class AxisChangeCallback

### Description:

This is the callback used with **Interface::RegisterAxisChangeCallback()**.

### Methods:

### Prototype:

```
virtual void proc(Interface *ip)=0;
```

### Remarks:

This callback object will be notified any time the user changes the reference coordinate system by:

- Changing the transform coordinate system drop-down menu.
- Changing the state of the transform center fly-off.
- Changing X, Y, Z, XY, YZ, ZX constraint buttons/fly-off.
- Using an accelerator or anything else that changes the above.

### Parameters:

**Interface \*ip**

A pointer available for calling methods provided by 3ds max.

# Class RedrawViewsCallback

See Also: [Class Interface](#).

class RedrawViewsCallback

## Description:

This is the callback used with

**Interface::RegisterRedrawViewsCallback()**.

## Methods:

## Prototype:

```
virtual void proc(Interface *ip)=0;
```

## Remarks:

Implemented by the Plug-In.

This method will be called after all the viewports have completed drawing.

## Parameters:

**Interface \*ip**

A pointer for calling functions available in 3ds max.

## Class SelectFilterCallback

See Also: [Class InterfaceServer](#), [Class Interface](#).

```
class SelectFilterCallback : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This is a call-back class for the selection filter drop down in the tab panel. This allows plug-ins to add additional filters to this list.

### Methods:

public:

### Prototype:

```
virtual TCHAR* GetName()=0;
```

### Remarks:

Returns the name of the filter that will appear in the drop down list in the tab panel.

### Prototype:

```
virtual BOOL IsFiltered(SClass_ID sid, Class_ID cid, INode  
*node) = 0;
```

### Remarks:

This is the method that does the filtering of the node. It returns TRUE if the node may be selected; FALSE if it is not selectable.

### Parameters:

**SClass\_ID sid**

The Super Class ID of the node.

**Class\_ID cid**

The Class ID of the node.

**INode \*node**

Points to the node to check.

# Class DisplayFilterCallback

See Also: [Class Interface](#).

```
class DisplayFilterCallback : public InterfaceServer
```

## Description:

This class is available in release 4.0 and later only.

This is a call-back class for the display filter list in the Hide by Category rollup of the Display command panel. This allows plug-ins to add additional filters to this list.

## Data Members:

public:

### **BOOL on;**

Determines if the callback is on or off. If a callback is selected in the list in the Display Panel list it is on; else it's off.

## Methods:

public:

## Prototype:

```
virtual TCHAR* GetName()=0;
```

## Remarks:

Returns the name that will appear in the drop down list in the display panel when the callback is registered.

## Prototype:

```
virtual BOOL IsVisible(SClass_ID sid, Class_ID cid, INode  
*node)=0;
```

## Remarks:

This is the method that does the filtering of the node. It returns TRUE if the node is visible; FALSE if it is not visible.

## Parameters:

**SClass\_ID sid**

The Super Class ID of the node.

**Class\_ID cid**

The Class ID of the node.

**Inode \*node**

Points to the node to check.

## List of Standard Command Modes

See Also: [Class Interface](#), [Command Modes and Mouse Procs](#), [Class Command Mode](#).

One of the following values:

## **XForm Command Modes**

**CID\_OBJMOVE**

**CID\_OBJROTATE**

**CID\_OBJSCALE**

**CID\_OBJUSCALE**

**CID\_OBJSQUASH**

**CID\_OBJSELECT**

## **Hierarchy / Space warp Command Modes**

**CID\_LINK**

**CID\_BINDWSM**

## **Viewport Command Modes**

**CID\_ZOOMVIEW**

**CID\_ZOOMREGION**

**CID\_PANVIEW**

**CID\_ROTATEVIEW**

**CID\_ZOOMALL**

**CID\_RNDREGION**

## **Camera Command Modes**

**CID\_CAMFOV**

**CID\_CAMDOLLY**

**CID\_CAMPERSP**

**CID\_CAMTRUCK**

**CID\_CAMROTATE**

**CID\_CAMROLL**

# Class EventUser

See Also: [Class EventRouter](#), [Class Interface](#).

class EventUser

## Description:

This is a generic event notification system. The only two places this is currently used are for when the Delete key is pressed or the Delete menu item is selected, and when the Backspace key is pressed. The usage can be seen in `\MAXSDK\SAMPLES\MODIFIERS\EDITPAT.CPP`, where the Edit Patch modifier sets up for notification of Delete operations. It is also used in `EDITSP.L.CPP` where it deals with deletion of selected items.

To use it:

- 1 Create an EventUser object.
- 2 Register the EventUser object with the appropriate router.
- 3 The EventRouter will call the EventUser's **Notify()** method when the event occurs.
- 4 When you're done with the EventUser object, call the EventRouter's **UnRegister()** method. This will delete the EventUser from the router's notification system.
- 5 If your code is part of a window proc, call the router's Register and UnRegister methods when the window receives WM\_ACTIVATE messages. This will properly uncouple the notification system when the window is deactivated.

## Methods:

### Prototype:

```
virtual void Notify()=0;
```

### Remarks:

Implemented by the Plug-In.

This is the proc called by the **EventRouter** when the event occurs.

## Class ArcballDialog

See Also: [Class ArcballCallback](#).

class ArcballDialog

### Description:

This class is available in release 2.0 and later only.

An instance of this class is created when the global function **ArcballDialog \*CreateArcballDialog()** is called. The **DeleteThis()** method of this class is used to free the memory allocated by **CreateArcballDialog()**. Call it when you are done using the dialog box.

### Methods:

#### Prototype:

**virtual void DeleteThis()=0;**

### Remarks:

This method is called to delete the instance of the class.

## Class TrackViewPick

See Also: [Class ReferenceTarget](#), [Class Interface](#) (see the method **TrackViewPickDlg**).

### Description:

This class stores the result of a selection from the Track View Pick dialog.

### Data Members:

**ReferenceTarget \*anim;**

The item the user picked.

**ReferenceTarget \*client;**

The owner of the **anim**.

**int subNum;**

Sub-animatable number of the **anim**.

# Class TrackViewFilter

See Also: [Class Animatable](#).

```
class TrackViewFilter : public InterfaceServer
```

## Description:

This is the callback object used to filter selections in the track view.

## Methods:

### Prototype:

```
virtual BOOL proc(Animatable *anim, Animatable *client,int  
subNum)=0;
```

### Remarks:

Implemented by the Plug-In.

This is the callback object proc used to filter selections in the track view.

### Parameters:

**Animatable \*anim**

The item the user picked.

**Animatable \*client**

The owner of the **anim**.

**int subNum**

The sub-animatable number of the **anim**.

### Return Value:

Return TRUE to accept the anim as selectable; otherwise FALSE.

### Prototype:

```
virtual BOOL TextColor(Animatable *anim, Animatable *client,  
int subNum, COLORREF& color);
```

### Remarks:

This method is available in release 4.0 and later only.

This method allows the filter to control the color of the label text used for the anim.

### Parameters:

**Animatable \*anim**

The item the user picked.

**Animatable \*client**

The owner of the **anim**.

**int subNum**

The sub-animatable number of the **anim**.

**COLORREF& color**

The color for the label text. See [COLORREF-DWORD format](#).

**Return Value:**

TRUE for the Treeview to use the color in the **color** argument, FALSE to ignore that color and use the system default.

**Default Implementation:**

```
{ return FALSE; }
```

## Class ExclList

See Also: [Class INode](#), [Class ILoad](#), [Class ISave](#), [Class IMergeManager](#)

class ExclList : public InterfaceServer

### Description:

This class is available in release 4.0 and later only.

This class represents an exclusion list and is a direct parallel to the NameTab, and converting from using one to the other is fairly simple.

### Methods:

public:

### Prototype:

**ExclList();**

### Remarks:

Constructor.

### Prototype:

**void SetFlag(ULONG f, BOOL b=1);**

### Remarks:

Sets the specified flag to the specified value.

### Parameters:

**ULONG f**

The flag(s) to set. One or more of the following values:

**NT\_INCLUDE**

This bit is used to indicate "Include" mode.

**NT\_AFFECT\_ILLUM**

This bit is used to indicate the "Illumination" check box in the exclusion list dialog.

**NT\_AFFECT\_SHADOWCAST**

This bit is used to indicate the "Shadow Casting" check box in the exclusion list dialog.

**BOOL b=1**

The value to set.

**Prototype:**

**BOOL TestFlag(ULONG f);**

**Remarks:**

Returns TRUE if the specified flag(s) are set; otherwise FALSE.

**Parameters:**

**ULONG f**

The flag(s) to set. One or more of the following values:

**NT\_INCLUDE**

This bit is used to indicate "Include" mode.

**NT\_AFFECT\_ILLUM**

This bit is used to indicate the "Illumination" check box in the exclusion list dialog.

**NT\_AFFECT\_SHADOWCAST**

This bit is used to indicate the "Shadow Casting" check box in the exclusion list dialog.

**Prototype:**

**int Count();**

**Remarks:**

This method returns the number of handles in the table.

**Prototype:**

**void Set(int i, INode \*node);**

**Remarks:**

This method allows you to set a specified entry in the table to the specified node.

**Parameters:**

**int i**

The index in the table.

**INode \*node**

The node to set.

**Prototype:**

**int FindNode(INode \*node);**

**Remarks:**

Returns the index of the node passed; otherwise returns -1.

**Parameters:**

**INode \*node**

The node to find.

**Prototype:**

**int AddNode(INode \*node);**

**Remarks:**

Appends the specified node to the end of the list.

**Parameters:**

**INode \*node**

The node to add.

**Return Value:**

Returns the number of items in the list prior to appending.

**Prototype:**

**void RemoveNode(INode \*node);**

**Remarks:**

Removes this node from the list.

**Parameters:**

**INode \*node**

The node to remove.

**Prototype:**

**void RemoveNode(int i);**

**Remarks:**

Removes the 'i-th' name from the list.

**Parameters:**

**int i**

Specifies the index of the node to remove.

**Prototype:**

**void SetSize(int num);**

**Remarks:**

Sets the size of the list. If the new size is smaller than the current size entries are deleted.

**Parameters:**

**int num**

Specifies the size of the list.

**Prototype:**

**SetCount(int num);**

**Remarks:**

Sets the size of the list. If the new size is smaller than the current size entries are deleted.

**Parameters:**

**int num**

Specifies the size of the list.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

Loads this ExclList from disk.

**Parameters:**

**ILoad \*iload**

This class provides methods to load data from disk.

**Return Value:**

See Also: [List of IO Results](#).

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

Saves this ExclList to disk.

**Parameters:**

**ISave \*isave**

This class provides methods to save data to disk.

**Return Value:**

See Also: [List of IO Results](#).

**Prototype:**

**void OnMerge(IMergeManager\* imm);**

**Remarks:**

This method takes care of setting the merge manager interface.

**Parameters:**

**IMergeManager\* imm**

A pointer to the merge manager interface.

**Prototype:**

**INode\* operator[] (const int i);**

**Remarks:**

Index operator.

**Prototype:**

**ExclList& operator=(const ExclList& e);**

**Remarks:**

Assignment operator.

**Prototype:**

**ExclList& operator=(const NameTab& n);**

**Remarks:**

Assignment operator.

## List of Material Browser Flags

One or more of the following values:

**BROWSE\_MATSONLY**

Materials only.

**BROWSE\_MAPSONLY**

Maps only.

**BROWSE\_INCNONE**

Include 'None' as an option.

**BROWSE\_INSTANCEONLY**

Only allow instances, not copies.

## Class HitByNameDlgCallback

See Also: [Class Interface](#).

class HitByNameDlgCallback

### Description:

This is the callback object used with **Interface::DoHitByNameDialog()**.

### Methods:

#### Prototype:

**virtual TCHAR \*dialogTitle()**

#### Remarks:

Implemented by the Plug-In.

Returns the title string displayed in the dialog.

#### Default Implementation:

```
{ return _T(""); }
```

#### Prototype:

**virtual TCHAR \*buttonText()**

#### Remarks:

Implemented by the Plug-In.

Returns the text displayed in the 'Select' or 'Pick' button.

#### Default Implementation:

```
{ return _T(""); }
```

#### Prototype:

**virtual BOOL singleSelect()**

#### Remarks:

Implemented by the Plug-In.

Returns TRUE if the user may only make a single selection in the list at one time; otherwise FALSE.

#### Default Implementation:

```
{ return FALSE; }
```

**Prototype:**

```
virtual BOOL useFilter()
```

**Remarks:**

Implemented by the Plug-In.

This gives the callback the opportunity to filter out items from the list. This is called before the dialog is presented. It returns TRUE if the **filter()** method (below) should be called; otherwise FALSE.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual int filter(INode *node)
```

**Remarks:**

Implemented by the Plug-In.

This method will be called if **useFilter()** above returned TRUE. This gives the callback the chance to filter out items from the list before they are presented to the user in the dialog. This is called once for each node that would otherwise be presented. Return nonzero to accept the item and zero to skip it.

**Parameters:**

**INode \*node**

The node to check for inclusion in the list.

**Return Value:**

Nonzero to accept the item and zero to skip it.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual BOOL useProc()
```

**Remarks:**

Implemented by the Plug-In.

Normally, when the user selects OK from the dialog, the system selects all the chosen nodes in the scene. At times a developer may want to do something other than select the chosen nodes. If this method returns TRUE then the nodes in the list will not be selected, but the **proc()** method is called instead (see below). If this method returns FALSE, then the nodes are selected in the scene and **proc()** is not called.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual void proc(INodeTab &nodeTab)
```

**Remarks:**

Implemented by the Plug-In.

This allows the plug-in to process the nodes chosen from the dialog in any manner. For example if the developer wanted to delete the nodes chosen using this dialog, they would implement **useProc()** to return TRUE and this method to delete all the nodes in the table passed.

**Parameters:**

**INodeTab &nodeTab**

A table of those nodes selected by the user. See [Template Class Tab](#).

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BOOL doCustomHilite()
```

**Remarks:**

Implemented by the Plug-In.

Normally, when the dialog is entered, the nodes in the scene that are selected are highlighted in the list. If this method returns TRUE, the developer may control which items are highlighted by implementing **doHilite()** (see below). If this method returns FALSE the selected nodes will have their names highlighted in the list.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual BOOL doHilite(INode *node)
```

**Remarks:**

Implemented by the Plug-In.

This method is called for each item in the list if **doCustomHilite()** returns TRUE. This method returns TRUE or FALSE to control if each item is highlighted.

**Parameters:**

**INode \*node**

The node to check.

**Return Value:**

TRUE to highlight the node in the list; FALSE to not highlight it.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual BOOL showHiddenAndFrozen();
```

**Remarks:**

This method is available in release 2.0 and later only.

This defaults to returning FALSE, which means that hidden and frozen objects will not be included in the select by name dialog list. If this method is overridden to return TRUE, then hidden and frozen nodes will be sent through the user-supplied filter as in version 1.x. (Note that, apart from Unhide by Name and Unfreeze by Name, the new default behavior is likely to be correct for all uses of this class.)

**Default Implementation:**

```
{ return FALSE; }
```

## Class SpaceArrayCallback

See Also: [Class Interface](#), [Class ShapeObject](#), [Class Spline3D](#), [Class Point3](#).

class SpaceArrayCallback

### Description:

This class is available in release 3.0 and later only.

This is the callback object for the method

**Interface::DoSpaceArrayDialog(SpaceArrayCallback \*sacb=NULL)=0**

This is the method which brings up the Spacing tool. This lets the user distribute objects based on the current selection along a path defined by a spline or a pair of points.

The callback is mostly for use for plug-ins that need to use the spacing tool as a way of generating spacing information. If one wants to customize the default behaviour of the dialog, they derive a class from this one and implement the virtual methods. The non-virtuals simply provide access to the spacing information. This is both for setting the defaults before calling the spacing tool as well as getting out the information after using the spacing tool.

### Methods:

public:

### Prototype:

**SpaceArrayCallback();**

### Remarks:

Constructor. The data members are initialized as follows:

**path = NULL; start = end = space = 0.0f; count = 1; oType = NODE\_COPY; sType = SPACE\_CENTER; follow = false; context = CTXT\_FREE; width = 0.0f; countLimit = INT\_MAX;**

### Prototype:

**virtual ~SpaceArrayCallback();**

### Remarks:

Destructor.

**Prototype:**

**virtual bool isModal();**

**Remarks:**

Implemented by the Plug-in.

If this method returns false, the dialog is presented as a modeless dialog.

**Default Implementation:**

**{ return true; }**

**Prototype:**

**virtual bool doPickPath();**

**Remarks:**

Implemented by the Plug-in.

If this method returns true, the path picking buttons are turned on.

**Default Implementation:**

**{ return false; }**

**Prototype:**

**virtual TCHAR \*dialogTitle();**

**Remarks:**

Implemented by the Plug-in.

Returns the title for the dialog (e.g. Space Array Tool).

**Default Implementation:**

**{ return \_T(""); }**

**Prototype:**

**virtual TCHAR \*startMessage();**

**Remarks:**

Implemented by the Plug-in.

Returns the message to be displayed in the static display right after the dialog comes up.

**Default Implementation:**

```
{ return _T(""); }
```

**Prototype:**

```
virtual TCHAR *buttonText();
```

**Remarks:**

Implemented by the Plug-in.

Returns the button text.

**Default Implementation:**

```
{ return _T("OK"); }
```

**Prototype:**

```
virtual bool isSilent();
```

**Remarks:**

Implemented by the Plug-in.

If this method returns true, the spacing tool generates points, but the actual dialog is not presented to the user. This is currently used, for example, by the Stairs in VIZ to generate spacing information for the mesh.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual bool doObjectType();
```

**Remarks:**

Implemented by the Plug-in.

If this method returns false, then the Instance, Copy, Reference radio buttons are disabled (greyed out).

**Default Implementation:**

```
{ return true; }
```

**Prototype:**

```
virtual void proc();
```

**Remarks:**

Implemented by the Plug-in.

This method is called after the spacing information is generated. It is here that a developer using this callback could get the updated values, perform some calculation of their own, etc.

**Default Implementation:**

```
{}
```

**Prototype:**

```
ShapeObject *getPath();
```

**Remarks:**

Implemented by the System.

Returns a pointer to the path ShapeObject.

**Prototype:**

```
void setPath(ShapeObject *p);
```

**Remarks:**

Implemented by the System.

Sets the path used (as a ShapeObject).

**Parameters:**

```
ShapeObject *p
```

Points to the shape object to use as a path.

**Prototype:**

```
void setPath(Point3 pt1, Point3 pt2);
```

**Remarks:**

Implemented by the System.

Sets the points for the path.

**Parameters:**

```
Point3 pt1
```

One of the endpoints.

```
Point3 pt2
```

The other endpoint.

**Prototype:**

**void setPath(Spline3D \*s);**

**Remarks:**

Implemented by the System.

Sets the path used.

**Parameters:**

**Spline3D \*s**

Points to the path to use.

**Prototype:**

**float getStart();**

**Remarks:**

Implemented by the System.

Returns the start offset.

**Prototype:**

**float getEnd();**

**Remarks:**

Implemented by the System.

Returns the end offset.

**Prototype:**

**float getSpace();**

**Remarks:**

Implemented by the System.

Returns the spacing.

**Prototype:**

**int getCount();**

**Remarks:**

Implemented by the System.

Returns the object count.

**Prototype:**

```
void setContext(CTYPE c);
```

**Remarks:**

Implemented by the System.

Sets the context.

**Parameters:**

**CTYPE c**

One of the following values:

**CTXT\_FREE** - Free Center

**CTXT\_CNTRCOUNT** - Divide Evenly, Objects at Ends

**CTXT\_CNTRSPACE** - Centered, Specify Spacing

**CTXT\_END** - End Offset

**CTXT\_ENDCOUNT** - End Offset, Divide Evenly

**CTXT\_ENDSPACE** - End Offset, Specify Spacing

**CTXT\_START** - Start Offset

**CTXT\_STARTCOUNT** - Start Offset, Divide Evenly

**CTXT\_STARTSPACE** - Start Offset, Specify Spacing

**CTXT\_FULLSPACE** - Specify Offset and Spacing

**CTXT\_FULLCOUNT** - Specify Offsets, Divide Evenly

**CTXT\_ENDLOCK** - Space from End, Unbounded

**CTXT\_ENDLOCKCOUNT** - Space from End, Specify Number

**CTXT\_ENDLOCKSPACE** - Space from End, Specify Spacing

**CTXT\_STARTLOCK** - Space from Start, Unbounded

**CTXT\_STARTLOCKCOUNT** - Space from Start, Specify Number

**CTXT\_STARTLOCKSPACE** - Space from Start, Specify Spacing

**CTXT\_FULLLOCKSPACE** - Specify Spacing, Matching Offsets

## CTXT\_FULLLOCKCOUNT - Divide Evenly, No Objects at Ends

### Prototype:

**int getContext();**

### Remarks:

Implemented by the System.

Returns the context. See **setContext()** above.

### Prototype:

**float getWidth();**

### Remarks:

Implemented by the System.

Returns the width. This is the width of the object to be arrayed. It is a single value so can be calculated any way the user wishes. In the default spacing tool this is calculated based on the x size of the bounding box.

### Prototype:

**void setWidth(float nWidth);**

### Remarks:

Implemented by the System.

Sets the width. See **getWidth()** above.

### Parameters:

**float nWidth**

The width to set.

### Prototype:

**void setStart(float f);**

### Remarks:

Implemented by the System.

Sets the start offset.

### Parameters:

**float f**

The start offset to set.

**Prototype:**

**void setEnd(float f);**

**Remarks:**

Implemented by the System.

Sets the end offset.

**Parameters:**

**float f**

The end offset to set.

**Prototype:**

**void setSpace(float f);**

**Remarks:**

Implemented by the System.

Sets the spacing.

**Parameters:**

**float f**

The spacing to set.

**Prototype:**

**void setCount(int n);**

**Remarks:**

Implemented by the System.

Sets the count.

**Parameters:**

**int n**

The count to set.

**Prototype:**

**bool getFollow();**

**Remarks:**

Implemented by the System.

Returns true if Follow is set (checked); otherwise false.

**Prototype:**

**void setFollow(bool t);**

**Remarks:**

Implemented by the System.

Sets the Follow state (checkbox).

**Parameters:**

**bool t**

Use true for checked; false for un-checked.

**Prototype:**

**OTYPE getObjectCreationType();**

**Remarks:**

Implemented by the System.

Returns the object creation type. One of the following values:

**NODE\_CPY** -- Copy

**NODE\_INST** -- Instance

**NODE\_REF** -- Reference

**Prototype:**

**void setObjectCreationType(OTYPE t);**

**Remarks:**

Implemented by the System.

Sets the object creation type. One of the following values:

**NODE\_CPY** -- Copy

**NODE\_INST** -- Instance

**NODE\_REF** -- Reference

**Prototype:**

**STYPE** getSpacingType();

**Remarks:**

Implemented by the System.

Returns the spacing type. One of the following values:

**SPACE\_CENTER**

**SPACE\_EDGE**

**Prototype:**

**void** setSpacingType(**STYPE** s);

**Remarks:**

Implemented by the System.

Sets the spacing type.

**Parameters:**

**STYPE** s

One of the following values:

**SPACE\_CENTER**

**SPACE\_EDGE**

**Prototype:**

**void** setMessage(**char** \*buf);

**Remarks:**

Implemented by the System.

Sets the message string.

**Prototype:**

**void** setCountLimit(**int** limit);

**Remarks:**

Implemented by the System.

Sets a limit on the count.

**Parameters:**

**int limit**

The limit to set.

**Prototype:**

**int getCountLimit();**

**Remarks:**

Implemented by the System.

Returns the limit on the count.

## List of Directory Names

See Also: [Class Interface](#).

The following is a list of the various standard 3ds max directories whose name may be retrieved. See the method **Interface::GetDir()**.

### **APP\_FONT\_DIR**

Sample result = **D:\3dsmax\Fonts**

### **APP\_SCENE\_DIR**

Sample result = **D:\3dsmax\Scenes**

### **APP\_IMPORT\_DIR**

Sample result = **D:\3dsmax\Meshes**

### **APP\_EXPORT\_DIR**

Sample result = **D:\3dsmax\Meshes**

### **APP\_HELP\_DIR**

Sample result = **D:\3dsmax\Help**

### **APP\_EXPRESSION\_DIR**

Sample result = **D:\3dsmax\Express**

### **APP\_PREVIEW\_DIR**

Sample result = **D:\3dsmax\Previews**

### **APP\_IMAGE\_DIR**

Sample result = **D:\3dsmax\Images**

### **APP\_SOUND\_DIR**

Sample result = **D:\3dsmax\Sounds**

### **APP\_PLUGCFG\_DIR**

Sample result = **D:\3dsmax\PlugCFG**

### **APP\_MAXSTART\_DIR**

Sample result = **D:\3dsmax\Scenes**

### **APP\_VPOST\_DIR**

Sample result = **D:\3dsmax\VPost**

### **APP\_DRIVERS\_DIR**

Sample result = **D:\3dsmax\Drivers**

### **APP\_AUTOBACK\_DIR**

Sample result = **D:\3dsmax\AutoBack**

### **APP\_MATLIB\_DIR**

This option is available in release 3.0 and later only.

Sample result =**D:\3dsmax\MatLibs**

### **APP\_SCRIPTS\_DIR**

This option is available in release 3.0 and later only.

Sample result =**D:\3dsmax\Scripts**

### **APP\_STARTUPSCRIPTS\_DIR**

This option is available in release 3.0 and later only.

Sample result =**D:\3dsmax\Scripts\Startup**

### **APP\_UI\_DIR**

This option is available in release 3.0 and later only.

Sample result=**D:\3dsmax\UI**

### **APP\_MAXROOT\_DIR**

Sample result = **d:\3dsmax\**

## Class DllDir

See Also: [Class DllDesc](#), [Class ClassDirectory](#), [Class ClassDesc](#), [Class ClassEntry](#), [Class Interface](#).

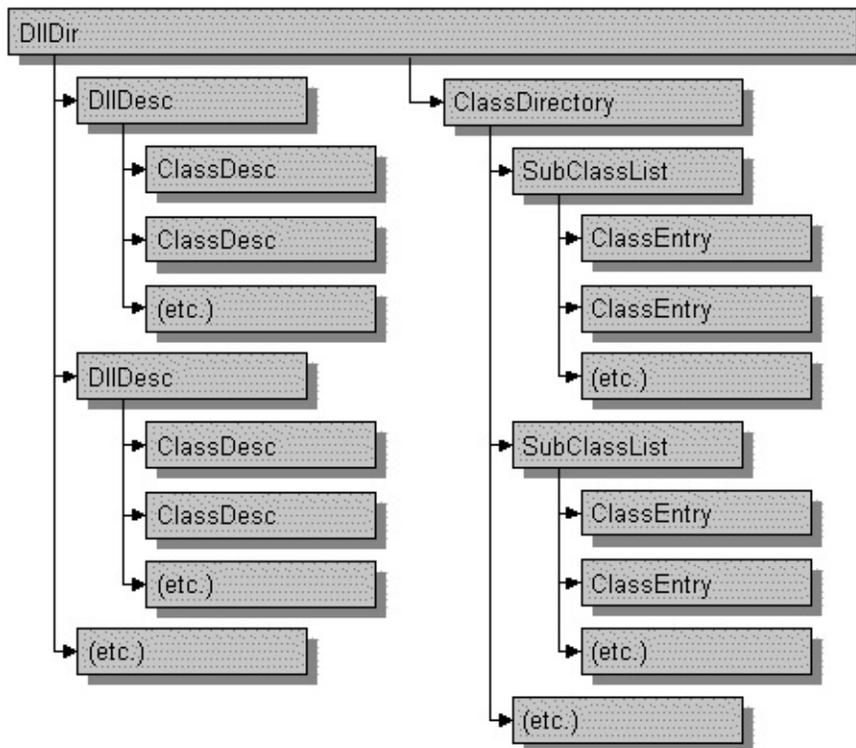
class DllDir

### Description:

This class is available in release 2.0 and later only.

It provides access to the DLL Directory which is a list of every DLL loaded in 3ds max. It also contains the Class Directory which maintains lists of all classes implemented in these DLLs.

The following diagram shows the relationship between the classes that make up the Dll Directory structure. The lines in the diagram indicate how the objects are accessed. Class **DllDir** provides access to a list of **DllDesc** objects. Each of these has a list of **ClassDesc** objects. The **ClassDesc** is the object implemented by the plug-in to let 3ds max know about its classification and capabilities. Class **DllDir** also provides access to the **ClassDirectory** which provides access to the a table of **SubClassList** objects which are grouped by super class ID. Each **SubClassList** has a series of **ClassEntry** objects. The **ClassEntry** object provides information about the plug-in classes (some of the same information as the class descriptor, usage counts, etc.).



Note: To get a reference to the central DLL directory see the method **Interface::GetDllDir()**.

All methods of this class are implemented by the system.

**Methods:**

**Prototype:**

**int Count();**

**Remarks:**

Returns the number of DLLs currently loaded.

**Prototype:**

**int LoadDllsFromDir(TCHAR \*directory, TCHAR \*wildcard, HWND hwnd=NULL);**

**Remarks:**

This method may be called to load DLLs from the specified directory.

**Parameters:**

**TCHAR \*directory**

Point to a null-terminated string that specifies a valid directory or path to load

the DLLs from.

**TCHAR \*wildcard**

Points to a null-terminated string that contains wildcard characters (\* and ?).

**HWND hwnd=NULL**

This parameter is not currently used.

**Return Value:**

Nonzero if successful; zero on error.

**Prototype:**

**ClassDirectory& ClassDir();**

**Remarks:**

Returns a reference to the **ClassDirectory** for this DLL directory.

**Prototype:**

**void UnloadAllDlls();**

**Remarks:**

This method unloads every DLL 3ds max has loaded. This calls the Win32 function **FreeLibrary()** on every DLL handle in the DLL directory.

**Prototype:**

**bool LoadADll(TCHAR \*d, bool late);**

**Remarks:**

This method is available in release 3.0 and later only.

This method is for internal use only.

**Operators:**

**Prototype:**

**DllDesc& operator[](int i);**

**Remarks:**

Access operator. This returns a reference to the 'i-th' **DllDesc**.

**Parameters:**

**int i**

Zero based index of specifying which **DllDesc** to return.

## Class LogSys

See Also: [Class Interface](#).

class LogSys

### Description:

MAX maintains a log file that contains the text of error / warning / information / debug messages generated by the system and plug-ins. This class is used to work with the log and send messages to it. The log file is placed in the **Network** directory and is called **Max.Log**. To access this facility from anywhere in MAX use the pointer returned from the method **Interface::Log()**. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual DWORD LogTypes();**

#### Remarks:

This method is used to find out what log types are enabled . See [List of System Error Log Message Types](#). The type values are ORed together to create the value returned.

#### Prototype:

**virtual void SetLogTypes(DWORD types);**

#### Remarks:

This method is used to set the log types that are enabled.

#### Parameters:

**DWORD types**

See [List of System Error Log Message Types](#).

#### Prototype:

**virtual void LogEntry(DWORD type, BOOL dialogue, TCHAR \*title, TCHAR \*text,... )=0;**

#### Remarks:

This method is used to log the error.

#### Parameters:

## **DWORD type**

Defines the type of log entry. See [List of System Error Log Message Types](#).

## **BOOL dialogue**

One of the following values:

### **NO\_DIALOG**

If this entry is just some information you don't want a dialogue for, or if you are handling the dialogue yourself use this value.

### **DISPLAY\_DIALOG**

Use this value if you want the message to be displayed in a dialogue. The system will determine if displaying a dialogue is appropriate based on network rendering mode.

## **TCHAR \*title**

This title string is optional. If non NULL, it will be used to define the module.

## **TCHAR \*text,...**

This parameter (and any other additional arguments that follow) make up the format specification. The format matches the standard C printf() function.

## **Sample Code:**

```
TheManager->Max()->Log()->LogEntry(SYSLOG_ERROR,  
NO_DIALOG, NULL, _T("%s - %s\n"), ShortDesc(), errText);
```

## **Prototype:**

```
virtual int Longevity();
```

## **Remarks:**

Returns the conditions under which the log is deleted.

## **Return Value:**

One of the following values:

### **SYSLOG\_LIFE\_EVER**

The log is never deleted.

### **SYSLOG\_LIFE\_DAYS**

This log is maintained for this number of days.

### **SYSLOG\_LIFE\_SIZE**

The log is maintained until it reaches this many kilobytes (KB).

**Prototype:****virtual void SetLongevity(int type);****Remarks:**

Sets the conditions under which the log is deleted.

**Parameters:****int type**

One of the following values:

**SYSLOG\_LIFE\_EVER**

The log is never deleted.

**SYSLOG\_LIFE\_DAYS**

This log is maintained for this number of days.

**SYSLOG\_LIFE\_SIZE**

The log is maintained until it reaches this many kilobytes (KB).

**Prototype:****virtual DWORD LogDays();****Remarks:**

Returns the conditions under which the log is cleared.

**Return Value:**

One of the following values:

**SYSLOG\_LIFE\_EVER**

The log is never deleted.

**SYSLOG\_LIFE\_DAYS**

This log is maintained for this number of days.

**SYSLOG\_LIFE\_SIZE**

The log is maintained until it reaches this many kilobytes (KB).

**Prototype:****virtual DWORD LogSize();****Remarks:**

Returns the size of the current log file in kilobytes (KB).

**Prototype:**

**virtual void SetLogDays(DWORD days);**

**Remarks:**

Set the number of days the log is maintained.

**Parameters:**

**DWORD days**

The number of days to maintain the log. After this many days after creation the log is deleted.

**Prototype:**

**virtual void SetLogSize(DWORD size);**

**Remarks:**

Set the maximum size in kilobytes (KB) of the log file. After this size is reached the log file is deleted.

**Parameters:**

**DWORD size**

The maximum size in kilobytes (KB) of the log file.

**Prototype:**

**void SetQuietMode(bool quiet);**

**Remarks:**

This method is available in release 3.0 and later only.

Enables or disables 'quiet' mode. When set to quiet mode, the

**LogSys::LogEntry(...)** method will not bring up any dialog boxes -- it will act as it does in network rendering mode. Note: After setting quiet mode, do not forget to clear it when you are done, since the user will not see any error messages from the renderer while quiet mode is enabled.

**Parameters:**

**bool quiet**

TRUE to enable; FALSE to disable.

**Prototype:**

**bool GetQuietMode();**

**Remarks:**

This method is available in release 3.0 and later only.

Returns TRUE if 'quiet' mode is enabled or FALSE if it's disabled. See **SetQuietMode()** above.

**Prototype:**

**virtual void SaveState()=0;**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual void LoadState()=0;**

**Remarks:**

This method is used internally.

## List of MAX Command IDs

See Also: [Class Interface](#).

The following command IDs may be passed to the method:

**Interface::ExecuteMAXCommand(int id);**

**MAXCOM\_RESET\_FILE**  
**MAXCOM\_TIME\_CONFIG**  
**MAXCOM\_UNFREEZE\_BY\_HIT**  
**MAXCOM\_BOX\_TOGGLE**  
**MAXCOM\_CYCLE\_SELECT\_METHOD**  
**MAXCOM\_ZOOM\_OUT\_2X\_ALL**  
**MAXCOM\_ZOOM\_IN\_2X\_ALL**  
**MAXCOM\_IZOOM\_OUT**  
**MAXCOM\_IZOOM\_IN**  
**MAXCOM\_IPAN**  
**MAXCOM\_SHOW\_LAST\_IMG**  
**MAXCOM\_APPLY\_IK**  
**MAXCOM\_KEY\_MODE**  
**MAXCOM\_TOGGLE\_IK**  
**MAXCOM\_SHADE\_SELECTED**  
**MAXCOM\_SELECT\_BY\_COLOR**  
**MAXCOM\_ZOOMEXT\_SEL**  
**MAXCOM\_ZOOMEXT\_SEL\_ALL**  
**MAXCOM\_CREATE\_MODE**  
**MAXCOM\_MODIFY\_MODE**  
**MAXCOM\_HIERARCHY\_MODE**  
**MAXCOM\_MOTION\_MODE**  
**MAXCOM\_DISPLAY\_MODE**  
**MAXCOM\_UTILITY\_MODE**  
**MAXCOM\_TEXTURE\_CORRECT**  
**MAXCOM\_ZOOM\_OUT\_2X**  
**MAXCOM\_ZOOM\_IN\_2X**  
**MAXCOM\_DEF\_LGT\_TOGGLE**  
**MAXCOM\_VPT\_SHAPE**

MAXCOM\_GROUP\_ATTACH  
MAXCOM\_GROUP\_DETACH  
MAXCOM\_PREV\_MOD  
MAXCOM\_NEXT\_MOD  
MAXCOM\_SAVEPLUS  
MAXCOM\_VIEW\_FILE  
MAXCOM\_UNHIDE\_BY\_NAME  
MAXCOM\_UNFREEZE\_BY\_NAME  
MAXCOM\_SPINSNAP\_TOGGLE  
MAXCOM\_HIDE\_INV  
MAXCOM\_FREEZE\_INV  
MAXCOM\_UNFREEZE\_ALL  
MAXCOM\_WIRE\_SMOOTH  
MAXCOM\_WIRE\_FACET  
MAXCOM\_BOX\_MODE  
MAXCOM\_BACKFACE  
MAXCOM\_TRAJECTORIES  
MAXCOM\_UNHIDE\_ALL  
MAXCOM\_SCALE\_CYCLE  
MAXCOM\_IK\_TERMINATOR  
MAXCOM\_RENDER\_SCENE  
MAXCOM\_RENDER\_LAST  
MAXCOM\_QUICK\_RENDER  
MAXCOM\_GRID\_NUDGE\_UP  
MAXCOM\_GRID\_NUDGE\_DOWN  
MAXCOM\_CYCLE\_SUBLEVEL  
MAXCOM\_HIDE\_SELECTION  
MAXCOM\_FREEZE\_SELECTION  
MAXCOM\_SHADE\_TOGGLE  
MAXCOM\_MIRROR  
MAXCOM\_ARRAY  
MAXCOM\_ALIGN  
MAXCOM\_ALIGNNORMALS  
MAXCOM\_HOLD

**MAXCOM\_FETCH**  
**MAXCOM\_SWAP\_LAYOUTS**  
**MAXCOM\_SAFEFRAME\_TOGGLE**  
**MAXCOM\_FILE\_MERGE**  
**MAXCOM\_TIME\_BACK**  
**MAXCOM\_TIME\_FORWARD**  
**MAXCOM\_TIME\_PLAY**  
**MAXCOM\_VIEWS\_REDRAW**  
**MAXCOM\_UNITSETUP**  
**MAXCOM\_DRAWINGAIDS**  
**MAXCOM\_SHOWHOMEGRID**  
**MAXCOM\_ACTHOMEGRID**  
**MAXCOM\_ACTGRIDOBJ**  
**MAXCOM\_GRIDS\_ALIGN**  
**MAXCOM\_BACKGROUND**  
**MAXCOM\_SHOWAXISICON**  
**MAXCOM\_FULLINTERACT**  
**MAXCOM\_VPTCONFIG**  
**MAXCOM\_VIDEOPOST**  
**MAXCOM\_PREVIEW**  
**MAXCOM\_VIEWPREVIEW**  
**MAXCOM\_RENAMEPREVIEW**  
**MAXCOM\_TOOL\_DUALPLANES**  
**MAXCOM\_LINK**  
**MAXCOM\_UNLINK**  
**MAXCOM\_BINDWSM**  
**MAXCOM\_SELECT**  
**MAXCOM\_MOVE**  
**MAXCOM\_ROTATE**  
**MAXCOM\_SCALE**  
**MAXCOM\_TREEVIEW**  
**MAXCOM\_MTLEDIT**  
**MAXCOM\_PANVIEW**  
**MAXCOM\_DOLLY**

**MAXCOM\_PERSP  
MAXCOM\_ROLL  
MAXCOM\_FOV  
MAXCOM\_TRUCK  
MAXCOM\_PANCAMERA  
MAXCOM\_ANGLE\_SNAP\_TOGGLE  
MAXCOM\_EDIT\_REDO  
MAXCOM\_VIEW\_REDO  
MAXCOM\_VPT\_TRACK  
MAXCOM\_VPT\_BOTTOM  
MAXCOM\_SUBOBJECT\_SEL  
MAXCOM\_VPT\_CAMERA  
MAXCOM\_VPT\_SPOT  
MAXCOM\_HIDE\_CAMERA\_TOGGLE  
MAXCOM\_VPT\_DISABLE  
MAXCOM\_VPT\_FRONT  
MAXCOM\_VPT\_GRID  
MAXCOM\_GRID\_TOGGLE  
MAXCOM\_TOOL\_HLIST  
MAXCOM\_HIDE\_HELPER\_TOGGLE  
MAXCOM\_VPT\_ISO\_USER  
MAXCOM\_VPT\_BACK  
MAXCOM\_VPT\_LEFT  
MAXCOM\_HIDE\_LIGHT\_TOGGLE  
MAXCOM\_TOOL\_ANIMMODE  
MAXCOM\_FILE\_NEW  
MAXCOM\_OVERRIDE  
MAXCOM\_FILE\_OPEN  
MAXCOM\_HIDE\_OBJECT\_TOGGLE  
MAXCOM\_VPT\_PERSP\_USER  
MAXCOM\_ACCEL\_PAN  
MAXCOM\_HIDE\_SHAPE\_TOGGLE  
MAXCOM\_VPT\_RIGHT  
MAXCOM\_ROTATEVIEW**

MAXCOM\_SNAP\_TOGGLE  
MAXCOM\_FILE\_SAVE  
MAXCOM\_FILE\_SAVEAS  
MAXCOM\_FILE\_IMPORT  
MAXCOM\_FILE\_PREFERENCES  
MAXCOM\_HIDE\_SYSTEM\_TOGGLE  
MAXCOM\_VPT\_TOP  
MAXCOM\_EDIT\_DELETE  
MAXCOM\_EDIT\_SELECTALL  
MAXCOM\_EDIT\_SELECTNONE  
MAXCOM\_EDIT\_SELECTINVERT  
MAXCOM\_RNS  
MAXCOM\_TTI  
MAXCOM\_PROPERTIES  
MAXCOM\_GROUP\_GROUP  
MAXCOM\_GROUP\_OPEN  
MAXCOM\_GROUP\_CLOSE  
MAXCOM\_GROUP\_UNGROUP  
MAXCOM\_TIME\_END  
MAXCOM\_HELP\_ABOUT  
MAXCOM\_TOOL\_X  
MAXCOM\_TOOL\_Y  
MAXCOM\_TOOL\_Z  
MAXCOM\_TOOL\_XY  
MAXCOM\_TIME\_START  
MAXCOM\_SELECT\_CHILD  
MAXCOM\_SELECT\_PARENT  
MAXCOM\_SPACEBAR  
MAXCOM\_TOOL\_MAXIMIZE  
MAXCOM\_TOOL\_ZOOMREGION  
MAXCOM\_HIDE\_WSM\_TOGGLE  
MAXCOM\_TOOL\_CENTER  
MAXCOM\_TOOL\_ZOOM  
MAXCOM\_TOOL\_ZOOMALL

**MAXCOM\_EDIT\_UNDO**  
**MAXCOM\_TOOL\_ZOOMEXTENTS**  
**MAXCOM\_VIEWS\_UNDO**  
**MAXCOM\_TOOL\_ZOOMEXTENTS\_ALL**  
**MAXCOM\_TOGGLE\_SOUND**  
**MAXCOM\_VPT\_TAB**  
**MAXCOM\_VPT\_SHIFTTAB**  
**MAXCOM\_CONFIGURE\_PATHS**  
**MAXCOM\_ADAPTIVE\_PERSP\_GRID\_TOGGLE**  
**MAXCOM\_TOOL\_EDIT\_MOD\_STACK**  
**MAXCOM\_TOOL\_EDGES\_ONLY\_TOGGLE**  
**MAXCOM\_PERCENT\_SNAP\_TOGGLE**  
**MAXCOM\_SNAPMODE\_TOGGLE**  
**MAXCOM\_RENDER\_ATMOSPHERE**  
**MAXCOM\_VIEWS\_SAVEACTIVEVIEW**  
**MAXCOM\_VIEWS\_RESTOREACTIVEVIEW**  
**MAXCOM\_VIEWS\_SHOWDEP**  
**MAXCOM\_FILE\_EXPORT**  
**MAXCOM\_EDIT\_PLACEHIGHLIGHT**  
**MAXCOM\_EDIT\_SNAPSHOT**  
**MAXCOM\_TOOL\_REGION\_TOGGLE**  
**MAXCOM\_FILE\_SUMMARYINFO**  
**MAXCOM\_SCHEMATICVIEW**

## List of Interface::Execute Command Options

See Also: [Class Interface](#)

In MAX 2.5, the following **#defines** may be passed as the **cmd** argument to the method **Interface::Execute()**.

Note that these may be removed from MAX 3.0. Developers may wish to use the API provided by [Structure NotifyInfo](#) for pre and post save callbacks.

### **I\_EXEC\_REGISTER\_POSTSAVE\_CB**

This command registers a callback which is called after a File / Save operation has occurred.

For example, here's how you would use it:

```
static GenericCallback mycb;
```

```
GetCOREInterface()-
```

```
>Execute(I_EXEC_REGISTER_POSTSAVE_CB,&mycb);
```

and when you are done:

```
GetCOREInterface()-
```

```
>Execute(I_EXEC_UNREGISTER_POSTSAVE_CB,&mycb);
```

For another way to do this, see [Structure NotifyInfo](#).

### **I\_EXEC\_UNREGISTER\_POSTSAVE\_CB**

This command un-registers a callback which is called after a File / Save operation has occurred.

### **I\_EXEC\_REGISTER\_PRESAVE\_CB**

This command registers a callback which is called before a File / Save operation has occurred.

### **I\_EXEC\_UNREGISTER\_PRESAVE\_CB**

This command un-registers a callback which is called before a File / Save operation has occurred.

## List of Extended Display Modes

See Also: [Class Interface](#) (methods **SetExtendedDisplayMode()** and **GetExtendedDisplayMode()**).

Extended display modes. One or more of the following values:

**EXT\_DISP\_NONE**

None in effect.

**EXT\_DISP\_SELECTED**

The object is selected.

**EXT\_DISP\_TARGET\_SELECTED**

The object's target is selected.

**EXT\_DISP\_LOOKAT\_SELECTED**

The object's lookat node is selected.

**EXT\_DISP\_ONLY\_SELECTED**

The object is the only thing selected.

**EXT\_DISP\_DRAGGING**

The object is being "dragged".

**EXT\_DISP\_ZOOM\_EXT**

The object is being tested for zoom extents.

## Class MAXFileOpenDialog

See Also: [Class Interface](#).

class MAXFileOpenDialog

### Description:

This class is available in release 3.0 and later only.

This class allows a custom file open dialog to be used. This object is set using the method **Interface::SetMAXFileOpenDlg()**.

### Methods:

public:

### Prototype:

```
virtual BOOL BrowseMAXFileOpen(TSTR& fileName, TSTR*  
defDir, TSTR* defFile)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called to bring up the custom file open dialog. It request a file name from the user and stores the result in **fileName**.

### Parameters:

**TSTR& fileName**

Set this to the file name choosen by the user.

**TSTR\* defDir**

The default directory to look in.

**TSTR\* defFile**

The default file name to use.

### Return Value:

TRUE if the user OKed the dialog; FALSE on cancel.

## Class MAXFileSaveDialog

See Also: [Class Interface](#).

class MAXFileSaveDialog

### Description:

This class is available in release 3.0 and later only.

This class allows a custom file save dialog to be used. This object is set using the method **Interface::SetMAXFileSaveDlg()**.

### Methods:

public:

### Prototype:

**virtual BOOL BrowseMAXFileSave(TSTR& fileName)=0;**

### Remarks:

This method is called to bring up the custom file save dialog. It request a file name from the user and stores the result in **fileName**.

### Parameters:

**TSTR& fileName**

Set this to the file name chosen by the user.

### Return Value:

TRUE if the user OKed the dialog; FALSE on cancel.

# Class Material

See Also: [Class TextureInfo](#), [Template Class Tab](#), [Class Point3](#), [Rendering Limits](#).

```
class Material : public BaseInterfaceServer
```

## Description:

This class describes the properties of a material used by the interactive renderer.

## Data Members:

public:

### **Point3 Ka;**

Ambient color setting. The values for x, y, and z range from 0.0 to 1.0. These correspond to red, green, and blue respectively.

### **Point3 Kd;**

Diffuse color setting. The values for x, y, and z range from 0.0 to 1.0. These correspond to red, green, and blue respectively.

### **Point3 Ks;**

Specular color setting. The values for x, y, and z range from 0.0 to 1.0. These correspond to red, green, and blue respectively.

### **float shininess;**

Shininess setting. This value ranges from 0.0 to 1.0.

### **float shinStrength;**

Shininess strength setting. This value ranges from 0.0 to 1.0.

### **float opacity;**

Opacity setting. This value ranges from 0.0 to 1.0.

### **float selfIllum;**

Self Illumination setting. This value ranges from 0.0 to 1.0.

### **int dblSided;**

Double sided material setting. Nonzero indicates double sided material; otherwise single sided.

### **int shadeLimit;**

Shading limit setting. See [Rendering Limits](#).

### **Tab<TextureInfo> texture;**

The table of textures used by this material.

**Methods:****Prototype:****Material();****Remarks:**

Class constructor. The data members are initialized as follows:

**Ka[0] = Ka[1] = Ka[2] = 0.3f;****Kd[0] = Kd[1] = Kd[2] = 0.9f;****Ks[0] = Ks[1] = Ks[2] = 0.9f;****shininess = 10.0f;****shinStrength = 1.0f;****opacity = 1.0f;****selfIllum = 0.0f;****dblSided = 0;****shadeLimit = 3;****TextureInfo texInfo;****texture.Append(1, &texInfo);****Prototype:****~Material();****Remarks:**

Class destructor.

## Class ModContextList

See Also: [Template Class Tab](#), [Class Interface](#) (method **GetModContexts()**)

```
class ModContextList : public Tab<ModContext*>;
```

### **Description:**

A modifier may be applied to several objects in the scene. The

**Interface::GetModContexts()** method retrieves a list of all the ModContexts for the current place in the history. This class is used as a table to hold the list of ModContexts.

# Class NameMaker

See Also: [Class Interface](#).

class NameMaker

## Description:

This class is used with method **Interface::NewUniqueName()**. All methods of this class are implemented by the system.

## Methods:

### Prototype:

```
virtual void MakeUniqueName(TSTR &name)=0;
```

### Remarks:

This method is used to make a unique version of the name passed and return it. This method is much more efficient than

**Interface::MakeNameUnique()**, which does a brute force enumeration of the scene hierarchy for every call. On creation, the **NameMaker** builds a directory which has one entry for each "base" name, such as "Sphere", "Camera", "Block", and along with each the maximum number suffix for that base name in the scene. When you call:

```
nm->MakeNameUnique
```

it only has to look at this relatively small list. It also caches the last hit, for further efficiency.

### Parameters:

**TSTR &name**

This is the name that is made unique (it contains both the source and the result).

### Prototype:

```
virtual void AddName(TSTR &name)=0;
```

### Remarks:

This method is available in release 2.0 and later only.

This method allows you to seed the name maker table with your own names.

This was used, for example, because 3ds max uses NameMakers to name sub-

objects in the NURBS object, and it's not appropriate to have the table seeded with top-level object names, but only with the existing sub-object names.

**Parameters:**

**TSTR &name**

The name to add.

## Class PickNodeCallback

See Also: [Class Interface](#).

class PickNodeCallback

### Description:

This class is used to filter nodes during a hit test. See the methods **Interface::PickNode()** and **PickModeCallback::GetFilter()**.

### Methods:

### Prototype:

```
virtual BOOL Filter(INode *node)=0;
```

### Remarks:

Implemented by the Plug-In.

This method should return TRUE if the node passed is an acceptable hit and FALSE otherwise.

### Parameters:

**INode \*node**

The node to test.

### Return Value:

Return TRUE if this is an acceptable hit, FALSE otherwise.

## Class PickModeCallback

See Also: [Class IObjParam](#), [Class ViewExp](#), [Class Interface](#), [Class IPoint2](#).

class PickModeCallback : public InterfaceServer

### Description:

This class is the callback object passed to **Interface::SetPickMode()**.

### Methods:

### Prototype:

```
virtual BOOL HitTest(IObjParam *ip, HWND hWnd, ViewExp *vpt, IPoint2 m, int flags)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called whenever the pick mode needs to hit test.

### Parameters:

#### **IObjParam \*ip**

An interface pointer available to call functions defined by 3ds max.

#### **HWND hWnd**

The window handle.

#### **ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports.

#### **IPoint2 m**

Point to check in screen coordinates.

#### **int flags**

The flags for hit testing. See [List of Hit Test Flags](#).

### Return Value:

Return TRUE if something was hit; otherwise FALSE.

### Sample Code:

```
// This implementation use the Interface::PickNode method  
// to perform the hit test.  
{  
    return ip->PickNode(hWnd,m,&thePickFilt)?TRUE:FALSE;
```

}

**Prototype:**

**virtual BOOL Pick(IObjParam \*ip,ViewExp \*vpt)=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called when the user picks something.

**Parameters:**

**IObjParam \*ip**

An interface pointer available to call functions defined by 3ds max.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports. The **vpt** should have the result of the hit test in it.

**Return Value:**

Return TRUE to end the pick mode; FALSE to stay in the pick mode. Note that returning TRUE will set the command mode to MOVE. When a plug-in is in the create branch, setting the command mode to move ends the creation process.

**Prototype:**

**virtual BOOL PickAnimatable(Animatable\* anim);**

**Remarks:**

This method is available in release 4.0 and later only.

This method gets called when a node or controller is picked from Trackview or Schematic view The track will not be selected in trackview if the this function returns FALSE. Override this function if you wish to support and receive nodes or controllers picked from trackview.

**Parameters:**

**Animatable\* anim**

A pointer to the animatable object.

**Return Value:**

TRUE if the Pick Mode callback accepts the animatable object, otherwise FALSE.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual BOOL RightClick(IObjParam *ip,ViewExp *vpt);
```

**Remarks:**

Implemented by the Plug-In.

This method is called when the user right-clicks or presses ESC.

**Parameters:**

**IObjParam \*ip**

An interface pointer available to call functions defined by 3ds max.

**ViewExp \*vpt**

An interface pointer that may be used to call methods associated with the viewports. The **vpt** should have the result of the hit test in it.

**Return Value:**

TRUE to end the pick mode, FALSE to continue picking.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

```
virtual void EnterMode(IObjParam *ip)
```

**Remarks:**

Implemented by the Plug-In.

Called when the mode is entered. The developer may provide any pre-processing here.

**Parameters:**

**IObjParam \*ip**

An interface pointer available to call functions defined by 3ds max.

**Prototype:**

```
virtual void ExitMode(IObjParam *ip)
```

**Remarks:**

Implemented by the Plug-In.

Called when the mode is exited. The developer may provide any post-processing here.

**Parameters:**

**IObjParam \*ip**

An interface pointer available to call functions defined by 3ds max.

**Prototype:**

**virtual HCURSOR GetDefCursor(IObjParam \*ip)**

**Remarks:**

Implemented by the Plug-In.

Called to get the default cursor to use.

**Return Value:**

The handle of the default cursor.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

**virtual HCURSOR GetHitCursor(IObjParam \*ip)**

**Remarks:**

Implemented by the Plug-In.

Called to get the hit test cursor to use.

**Return Value:**

The handle of the hit test cursor.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

**virtual PickNodeCallback \*GetFilter()**

**Remarks:**

Implemented by the Plug-In.

This method is called if the user hits the H key while in your pick mode. You can provide a filter to filter the name list. See [Class PickNodeCallback](#).

**Return Value:**

A pointer to an instance of **PickNodeCallback**.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual BOOL AllowMultiSelect();
```

**Remarks:**

This method is available in release 2.0 and later only.

Implement this method to return TRUE to allow the user to pick more than one thing. In that case the **Pick()** method may be called more than once.

**Return Value:**

TRUE to allow multiple picks; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

# Class PreviewParams

See Also: [Class Interface](#).

## Description:

The data members of this class are used to specify the options for creating a preview of the active viewport. A pointer to an instance of this class is passed into the **Interface** method:

```
virtual void CreatePreview(PreviewParams *pvp=NULL)=0;
```

## Data Members:

public:

### **BOOL outputType;**

Specifies the output file or device. This value may be either **0**, **1** or **2**.

**0** specifies the default AVI codec.

**1** specifies the user picks a file.

**2** specifies the user picks device.

### **nit pctSize;**

Specifies the percentage (0-100) of current rendering output resolution.

### **int start;**

The start frame limits.

### **int end;**

The end frame limit.

### **int skip;**

Specifies how many frames are skipped between rendered frames. The is similar to the 'Every Nth Frame' parameter in the user interface.

### **int fps;**

The frames per second setting.

### **BOOL dspGeometry;**

If TRUE is displayed in the viewports; otherwise it is not shown.

### **BOOL dspShapes;**

If TRUE shapes are displayed in the preview; otherwise they are not shown.

### **BOOL dspLights;**

If TRUE lights are displayed in the preview; otherwise they are not shown.

**BOOL dspCameras;**

If TRUE cameras are displayed in the preview; otherwise they are not shown.

**BOOL dspHelpers;**

If TRUE helper objects are displayed in the preview; otherwise they are not shown.

**BOOL dspSpaceWarps;**

If TRUE space warp helper objects are displayed in the preview; otherwise they are not shown.

**BOOL dspGrid;**

If TRUE the grid lines are displayed in the preview; otherwise they are not shown.

**BOOL dspSafeFrame;**

If TRUE the safe frames are displayed in the preview; otherwise they are not shown.

**BOOL dspFrameNums;**

If TRUE frame numbers are shown in the preview; otherwise they are not shown.

**int rndLevel;**

Specifies the rendering level used (these are the same options available to the user when setting viewport rendering levels). Valid values are

**0,1,2,3,4,6,7.**

**0** specifies smooth with highlights.

**1** specifies smooth without highlights.

**2** specifies faceted with highlights

**3** specifies faceted without highlights.

**4** specifies lit wireframe mode.

**6** specifies unlit wireframe.

**7** specifies box mode.

**int dspBkg;**

If nonzero the background image is displayed; otherwise it is not used.

## List of Render Types

See Also: [Class Interface](#).

One of the following values:

### **RENDTYPE\_NORMAL**

The currently selected viewport (the same as the MAX 'View' option).

### **RENDTYPE\_REGION**

This is the same as the MAX 'Region' option.

### **RENDTYPE\_BLOWUP**

This is the same as the MAX 'Blowup' option.

### **RENDTYPE\_SELECT**

This renders the selected objects only (the same as the MAX 'Selected' option).

### **RENDTYPE\_REGIONCROP**

This is the same as the MAX 'Crop' option.

### **RENDTYPE\_REGION\_SEL**

This type is available in release 4.0 and later only.

Does a region render using the bounding rectangle of the selection.

Note: Not to be passed into plugin renderers. The purpose is for passing to

**Interface::OpenCurRenderer()**, which converts them into

RENDTYPE\_REGION and RENDTYPE\_REGIONCROP, respectively.

### **RENDTYPE\_CROP\_SEL**

This type is available in release 4.0 and later only.

Does a crop render using the bounding rectangle of the selection

Note: Not to be passed into plugin renderers. The purpose is for passing to

**Interface::OpenCurRenderer()**, which converts them into

RENDTYPE\_REGION and RENDTYPE\_REGIONCROP, respectively.

# Class ViewParams

See Also: [Class Matrix3](#), [Class Renderer](#), [Class RendParams](#).

```
class ViewParams : public BaseInterfaceServer
```

## Description:

This class describes the properties of a view that is being rendered. These are properties such as the type of view (parallel or perspective), its clipping distances, width, height, zoom factor, field-of-view, etc.

## Data Members:

public:

### **Matrix3 prevAffineTM;**

This is the world space to camera space transformation matrix computed 2 ticks before the **affineTM** matrix below.

### **Matrix3 affineTM;**

This matrix will take a point in world space and convert it to camera space (or world to view space if it's a viewport). The camera coordinates are set up looking down the -Z axis, X is to the right, and Y is up.

### **int projType;**

One of the following values:

#### **PROJ\_PERSPECTIVE**

The view is a perspective projection.

#### **PROJ\_PARALLEL**

The view is a parallel projection.

### **float hither, yon;**

The hither and yon clipping distances.

### **float distance;**

This data member is available in release 3.0 and later only.

The distance from the view point to the image (view) plane.

### **float zoom;**

The zoom factor of the viewport for parallel projection. The zoom factor gives the amount of magnification relative to a standard view width of 400 pixels.

This is best explained via the following code fragment:

**ComputeViewParams()** computes the projection factors for a given view,

and **MapToScreen()** applies these factors to map a point from 3D camera coordinates to 2D screen coordinates.

```
#define VIEW_DEFAULT_WIDTH ((float)400.0)
void SRendParams::ComputeViewParams(const
ViewParams&vp) {
    if (vp.projType == PROJ_PERSPECTIVE) {
        float fac = -(float)(1.0 / tan(0.5*(double)vp.fov));
        xscale = fac*dw2; // dw2 = float(devWidth)/2.0
        yscale = -devAspect*xscale;
    }
    else {
        xscale =
(float)devWidth/(VIEW_DEFAULT_WIDTH*vp.zoom);
        yscale = -devAspect*xscale;
    }
}
Point2 SRendParams::MapToScreen(Point3 p) {
    Point2 s;
    if (proj_type==PROJ_PERSPECTIVE) {
        s.x = dw2 + xscale*p.x/p.z;
        s.y = dh2 + yscale*p.y/p.z;
    }
    else {
        s.x = dw2 + xscale*p.x;
        s.y = dh2 + yscale*p.y;
    }
    return s;
}
```

**float fov;**

Field of view in radians for perspective projections.

**float nearRange;**

This data member is available in release 2.0 and later only.

The near environment range setting (used for fog effects).

**float farRange;**

This data member is available in release 2.0 and later only.  
The far environment setting (used for fog effects).

**Methods:**

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);**

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:**

**int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

# Class RendProgressCallback

See Also: [Class Renderer](#).

class RendProgressCallback

## Description:

This class is a callback passed in to the renderer. The system passes this callback to the renderer, and the renderer will use these methods whenever it is doing something that is going to take some time. For instance when transforming objects it can update the progress bar. This is also passed in to the shadow buffer code so the shadow buffer can show its progress. All methods of this class are implemented by the system. They are called by a plug-in renderer.

## Methods:

### Prototype:

**virtual void SetTitle(const TCHAR \*title)=0;**

### Remarks:

Allows the plug-in to set the string displayed in renderer dialog.

### Parameters:

**const TCHAR \*title**

The string to display.

### Prototype:

**virtual int Progress(int done, int total)=0;**

### Remarks:

Allows the plug-in to update the renderer progress display.

### Parameters:

**int done**

The number of items completed so far.

**int total**

The total number of items to process.

### Return Value:

**RENDPROG\_CONTINUE**

Continue to process.

## **RENDPROG\_ABORT**

Stop processing.

### **Prototype:**

**virtual void SetCurField(int which);**

### **Remarks:**

Sets the field number display.

### **Parameters:**

**int which**

**FIELD\_FIRST**

**FIELD\_SECOND**

**FIELD\_NONE**

### **Prototype:**

**virtual void SetSceneStats(int nlights, int nrayTraced, int nshadowed, int nobj, int nfaces)**

### **Remarks:**

The plug-in renderer should call this on every frame, passing in values for the various parameters. These are displayed in the rendering in progress dialog.

### **Parameters:**

**int nlights**

The total number of lights.

**int nrayTraced**

The number of lights using raytraced shadows.

**int nshadowed**

The number of lights using shadows.

**int nobj**

The total number of objects.

**int nfaces**

The total number of faces.

## List of Render Setting IDs

See Also: [Class Interface](#).

One of the following values:

### **RS\_Production**

The MAX Production renderer.

### **RS\_Draft**

The MAX Draft renderer.

### **RS\_IReshade**

The MAX Interactive Renderer.

## Hit Test Types

The following are the various type of hit testing:

### **HITTYPE\_POINT**

Hit test by a single pick point.

### **HITTYPE\_BOX**

Hit test by a rectangular area.

### **HITTYPE\_CIRCLE**

Hit test by circular selection area.

### **HITTYPE\_FENCE**

Hit testing by an arbitrary polygon fence.

### **HITTYPE\_SOLID**

Hit test a face as if it was solid (even in wireframe mode). Treating an item as solid means the face will be hit if the mouse is anywhere inside the face region and not just over a visible edge. For example in 3ds max when an object is not selected and you put the mouse over it to select it, you need to put it over the wireframe. When an object is selected however you can put the mouse anywhere over the object and the system still considers this a valid area for hit testing. This later case is treating the faces of the selected object as solids.

## Hit Test Flags

The following are the various hit test flags. You may use these flags in combination, for example: **HIT\_SELONLY | HIT\_SELSOLID**.

### **HIT\_SELONLY**

Hit test selected items only.

### **HIT\_UNSELONLY**

Hit test unselected items only.

### **HIT\_ABORTONHIT**

Abort the process of hit testing after finding any hit.

### **HIT\_SELSOLID**

This treats selected items as solid and unselected items as not solid. Treating an item as solid means the face will be hit if the mouse is anywhere inside the face region and not just over a visible edge.

### **HIT\_ANYSOLID**

This treats any item as solid.

### **HIT\_TRANSFORMGIZMO**

This option is available in release 3.0 and later only.

Hit test the transform gizmo.

### **HIT\_SWITCH\_GIZMO**

This option is available in release 3.0 and later only.

The selection processor that does the hit-testing will include this flag when hit-testing on a `MOUSE_POINT` message, because when this flag is active and the transform gizmo's hit-testing hits the manipulator, it should switch the axis mode to the axis that is hit. Normally the transform gizmo hit-testing will only highlight the axis if it hits it - but when this flag is active it should also set the axis mode (using `PushAxisMode()` or `SetAxisMode()`)

### **HIT\_MANIP\_SUBHIT**

This option is available in release 4.0 and later only.

Hit test sub-manipulators.

## List of PropertySet Options

See Also: [Class Interface](#).

One of the following values:

### **PROPSET\_SUMMARYINFO**

This corresponds to the File Properties Summary tab properties

### **PROPSET\_DOCSUMMARYINFO**

This corresponds to the File Properties Contents tab properties (Document Contents)

### **PROPSET\_USERDEFINED**

This corresponds to the File Properties Custom tab properties

## List of Marker Types

See Also: [Class GraphicsWindow](#).

The following are the available marker types:

### **POINT\_MRKR**

This marker is a single pixel on the display.

### **HOLLOW\_BOX\_MRKR**

This marker is a small box centered on the point.

### **PLUS\_SIGN\_MRKR**

This marker is a plus sign (+) at the point.

### **ASTERISK\_MRKR**

This marker is an asterisk (\*) at the point.

### **X\_MRKR**

This marker is an X at the point.

### **BIG\_BOX\_MRKR**

This marker is a large box centered on the point.

### **CIRCLE\_MRKR**

This marker is a circle at the point.

### **TRIANGLE\_MRKR**

This marker is triangle centered on the point.

### **DIAMOND\_MRKR**

This marker is diamond centered on the point.

### **SM\_HOLLOW\_BOX\_MRKR**

This marker is a hollow box at the point.

### **SM\_CIRCLE\_MRKR**

This marker is small circle at the point.

### **SM\_TRIANGLE\_MRKR**

This marker is small triangle centered on the point.

### **SM\_DIAMOND\_MRKR**

This marker is small diamond centered on the point.

### **DOT\_MRKR**

This option is available in release 3.0 and later only.

This marker is a large dot.

**SM\_DOT\_MRKR**

This option is available in release 3.0 and later only.

This marker is a smaller dot.

# Class ParticleSys

See Also: [Class GraphicsWindow](#), [Marker Types](#), [Class HitRegion](#).

class ParticleSys

## Description:

This class describes a particle system. Methods are available to display, hit test, and compute the bounding box of the particle system. Other methods allocate and free the particles and allow custom particle drawing procedures to be used.

## Data Members:

public:

**Tab<Point3> points;**

The location of each particle.

**Tab<Point3> vels;**

The velocity of each particle (optional).

**Tab<TimeValue> ages;**

The age of each particle (optional).

**float size;**

The world space radius of a particle.

## Methods:

### Prototype:

```
void Render(GraphicsWindow *gw, MarkerType  
type=POINT_MRKR);
```

### Remarks:

Implemented by the System.

Draws the particle system into the GraphicsWindow.

### Parameters:

**GraphicsWindow \*gw**

The graphics window into which to particle system is to be drawn.

**MarkerType type=POINT\_MRKR**

One of the following values:

See [Marker Types](#).

**Prototype:**

```
BOOL HitTest(GraphicsWindow *gw, HitRegion *hr,  
int abortOnHit=FALSE, MarkerType type=POINT_MRKR);
```

**Remarks:**

Implemented by the System.

This method hit tests the particle system and returns TRUE if the particle system was hit.

**Parameters:**

**GraphicsWindow \*gw**

The graphics window to hit test in.

**HitRegion \*hr**

Pointer to an instance of HitRegion describing the hit test region.

**int abortOnHit=FALSE**

If TRUE the hit testing should be aborted upon the first successful hit; otherwise hit testing should continue through all particles.

**MarkerType type=POINT\_MRKR**

The type of particle marker being used. One of the following values:

See [Marker Types](#).

**Return Value:**

TRUE if a particle is hit; otherwise FALSE.

**Prototype:**

```
Box3 BoundBox(Matrix3 *tm=NULL);
```

**Remarks:**

Implemented by the System.

Returns the 3D bounding box of the particles.

**Parameters:**

**Matrix3 \*tm=NULL**

If not NULL, this is the optional space to compute the bounding box in.

**Prototype:**

```
void FreeAll();
```

**Remarks:**

Implemented by the System.

Sets all the counts to 0. This will be the points, and if used, the velocities and ages.

**Prototype:**

```
void SetCount(int c, DWORD flags);
```

**Remarks:**

Implemented by the System.

Sets the size of the particle system. This is to at least set the number of points in the particle system. The flags indicate if optional parameters velocities and ages should be allocated as well.

**Parameters:**

**int c**

The size for each allocated table.

**DWORD flags**

One or more of the following values:

**PARTICLE\_VELS** - Velocities should be allocated.

**PARTICLE\_AGES** - Particles ages should be allocated.

**Prototype:**

```
int Count();
```

**Remarks:**

Implemented by the System.

Returns the number of points in the particle system.

**Prototype:**

```
BOOL Alive(int i)
```

**Remarks:**

Implemented by the System.

Determines if particle **i** is alive (has not expired).

**Parameters:**

**int i**

The index of the particle to check.

**Return Value:**

TRUE if the 'i-th' particle is alive; otherwise FALSE.

**Prototype:**

**void SetCustomDraw(CustomParticleDisplay \*d);**

**Remarks:**

Implemented by the System.

Establishes a custom draw callback object. This allows the particles to be displayed in any manner desired (not just using the standard point markers).

See [Class CustomParticleDisplay](#).

**Parameters:**

**CustomParticleDisplay \*d**

The custom draw callback object.

**Operators:**

**Prototype:**

**Point3& operator[](int i)**

**Remarks:**

Implemented by the System.

Returns the 'i-th' point of the particle system.

## Class ISubMtlAPI

See Also: [Working with Materials](#).

### Description:

This class is available in release 2.5 and later only.

In 3ds max 2.5 this is only supported for objects flowing down the geometry pipeline.

This class is used if you wish to support the direct assignment of sub materials to selected faces. For a reference implementation of this class, please refer to the code in

**\\MAXSDK\\SAMPLES\\MESH\\EDITABLEMESH\\TRIOBJED.CPP.**

### Methods:

public:

### Prototype:

**virtual MtlID GetNextAvailMtlID()=0;**

### Remarks:

This method returns a material ID that is currently not used by the object. If the current face selection share one single MtlID that is not used by any other faces, you should use it.

### Prototype:

**virtual BOOL HasFaceSelection()=0;**

### Remarks:

This method indicates if you are active in the modifier panel and have an active face selection. Return TRUE if so, otherwise FALSE.

### Prototype:

**virtual void SetSelFaceMtlID(MtlID id, BOOL bResetUnsel = FALSE)=0;**

### Remarks:

This method sets the selected faces to the specified material ID..

### Parameters:

**MtlID id**

The material id to set for the selected faces.

**BOOL bResetUnsel = FALSE**

If TRUE, then you should set the remaining unselected face material IDs to 0.

**Prototype:**

**virtual int GetSelFaceUniqueMtlID()=0;**

**Remarks:**

This method returns the material ID of the selected face(s). If multiple faces are selected they should all have the same MtlID -- otherwise you should return -1. If faces other than the selected share the same material ID, then return -1.

**Prototype:**

**virtual int GetSelFaceAnyMtlID()=0;**

**Remarks:**

This method returns the material ID of the selected face(s). If multiple faces are selected they should all have the same MtlID, otherwise return -1.

**Prototype:**

**virtual int GetMaxMtlID()=0;**

**Remarks:**

This method returns the highest MtlID on the object.

## Class AttachMatDlgUser

See Also: [Class Object](#), [Class Modifier](#).

class AttachMatDlgUser

### Description:

This class is available in release 3.0 and later only.

This class provides a general way for objects to handle the attach materials dialog presented when users attach objects to each other. To use this class do the following:

- 1) Subclass your modifier or editable object off this class as shown below. Implement the four functions that give access to the attach/condense options.
- 2) Now your class simply calls the global **DoAttachMatOptionDialog()** function, which deals with the parameters uniformly for all users. The implementation in Edit Spline is like so:

```
class EditSplineMod : public Modifier, ..., AttachMatDlgUser {  
...  
    static int attachMat;  
    static BOOL condenseMat;  
...  
    // from AttachMatDlgUser  
    int GetAttachMat() { return attachMat; }  
    void SetAttachMat(int value) { attachMat = value; }  
    BOOL GetCondenseMat() { return condenseMat; }  
    void SetCondenseMat(BOOL sw) { condenseMat = sw; }
```

And the statics are defined as:

```
int EditSplineMod::condenseMat = FALSE;  
int EditSplineMod::attachMat = ATTACHMAT_NEITHER;
```

### Function:

```
    BOOL DoAttachMatOptionDialog(IObjParam *ip,  
    AttachMatDlgUser *user);
```

**Remarks:**

This global function is available in release 3.0 and later only.

This displays the dialog that you currently get in Edit(able) mesh when you attach objects to each other. It gives options for how to deal with combining materials, and whether or not to condense materials which have excess material slots.

**Parameters:**

**IObjParam \*ip**

The interface pointer.

**AttachMatDlgUser \*user**

Points to the AttachMatDlgUser object.

**Return Value:**

TRUE if the user OKed the dialog; FALSE if the user Cancelled.

**Methods:**

public:

**Prototype:**

**virtual int GetAttachMat()=0;**

**Remarks:**

Returns the attach material setting. One of the following values:

**ATTACHMAT\_IDTOMAT**

Match material IDs to material.

**ATTACHMAT\_MATTOID**

Match Material to Material IDs.

**ATTACHMAT\_NEITHER**

Do not modify Material IDs or Material.

**Prototype:**

**virtual void SetAttachMat(int value)=0;**

**Remarks:**

Sets the attach material value.

**Parameters:**

**int value**

One of the following values:

**ATTACHMAT\_IDTOMAT**

Match material IDs to material.

**ATTACHMAT\_MATTOID**

Match Material to Material IDs.

**ATTACHMAT\_NEITHER**

Do not modify Material IDs or Material.

**Prototype:**

**virtual BOOL GetCondenseMat()=0;**

**Remarks:**

Returns the condense material and IDs settings.

**Prototype:**

**virtual void SetCondenseMat(BOOL sw)=0;**

**Remarks:**

Sets the condense material and IDs setting.

**Parameters:**

**BOOL sw**

TRUE for on; FALSE for off.

## List of Display Flags

One or more of the following values:

### **USE\_DAMAGE\_RECT**

If this flag is set, only the damaged area needs to be displayed. The damaged rectangle may be retrieved using **INode::GetDamagedRect()**. See [Class INode](#).

### **DISP\_SHOWSUBOBJECT**

This indicates if an item should display its sub-object selection state. The system will set this flag if the item is selected, the user is in the modify branch, and the item is in sub-object selection mode.

## Class RemapDir

See Also: [Class ReferenceTarget](#).

class RemapDir

### Description:

This class is used for remapping references during a Clone. It is used when cloning items that are instanced so that the plug-in can maintain the same instance relationship within the clone. All methods of this class are implemented by the system.

### Methods:

### Prototype:

**virtual RefTargetHandle CloneRef(RefTargetHandle oldTarg);**

### Remarks:

In the **ReferenceTarget::Clone()** procedure when an item is cloning itself it should clone all its references. Instead of calling **Clone()** on all of its references it should instead call this method passing it the item to copy. This method will return a copy of the item or a pointer to a copy of the item if it was already copied.

### Parameters:

**RefTargetHandle oldTarg**

This is the item that is to be copied.

### Return Value:

A copy of the item or a pointer to a copy of the item if it was already copied.

### Prototype:

**virtual void PatchPointer(RefTargetHandle\* patchThis,  
RefTargetHandle oldTarg)=0;**

### Remarks:

This method is used to patch the pointer for cloned items. This method is used by system plug-ins for example. The Ring Array system has an array of nodes it maintains. When the system is cloned this method is used to clone the table of nodes. The new ring array master controller will also have a table of nodes,

but it does not want the pointer to point to the old nodes, it should point to the new cloned nodes. The nodes may not be cloned yet at the time the master controller was cloned however. This method allows the pointer to be changed to point at the new nodes.

**Parameters:**

**RefTargetHandle\* patchThis**

The pointer should point at this item.

**RefTargetHandle oldTarg**

The original target.

**Sample Code:**

The following sample code demonstrates the use of this method to patch the pointer to the table of nodes maintained by the ring array system.

```
RefTargetHandle RingMaster::Clone(RemapDir& remap) {  
    int i;  
    RingMaster* newm = new RingMaster();  
    newm->ReplaceReference(0,pblock->Clone(remap));  
    newm->numNodes = numNodes;  
    newm->nodeTab.SetCount(numNodes);  
    for (i=0; i<numNodes; i++) newm->nodeTab[i] = NULL;  
    for (i=0; i<numNodes; i++) {  
        remap.PatchPointer((RefTargetHandle*)&newm->nodeTab[i],  
            (RefTargetHandle)nodeTab[i]);  
    }  
    return(newm);  
}
```

**Prototype:**

**virtual RefTargetHandle FindMapping(RefTargetHandle from)=0;**

**Remarks:**

If an item has been cloned, this method will return the cloned copy of it. If it has not been cloned, NULL will be returned.

**Parameters:**

**RefTargetHandle from**

The item to check.

**Prototype:**

**virtual void AddEntry(RefTargetHandle hfrom, RefTargetHandle hto)=0;**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual void Backpatch()=0;**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual void Clear()=0;**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

This method is used internally.

## Class CreateMouseButton

See Also: [Class BaseObject](#) (method `GetCreateMouseButton()`), [Class ViewExp](#), [Class Matrix3](#), [Class IPoint2](#).

class CreateMouseButton

### Description:

This is the callback object for handling the creation process of a plug-in object.

### Methods:

### Prototype:

```
virtual int proc(ViewExp *vpt, int msg, int point, int flags, IPoint2 m, Matrix3& mat)=0;
```

### Remarks:

Implemented by the Plug-In.

This is the method where the developer defines the user / mouse interaction that takes place during the creation phase of an object.

### Parameters:

#### **ViewExp \*vpt**

The viewport the creation process is taking place in.

#### **int msg**

This message describes the type of event that occurred. See [List of Mouse Callback Messages](#).

#### **int point**

The point number. this is 0 for the first click, 1 for the second, etc.

#### **int flags**

These flags describe the state of the mouse button and keyboard Shift/Ctrl/Alt keys. See [List of Mouse Callback Flags](#).

#### **IPoint2 m**

The 2D screen point that the user clicked on. Methods in the viewport interface allow this point to be converted into a world space ray or a 3D view space point. A world space ray can be intersected with the active construction plane which results in a point on the active construction plane. See [Class ViewExp](#).

## **Matrix3& mat**

This represents the transformation of the object relative to the construction plane. Typically the plug-in would get a point on the construction plane based on the screen point that the user clicked on and set the translation component of this matrix based on that point.

### **Return Value:**

Return one of the following value to indicate the state of the creation process:

#### **CREATE\_CONTINUE**

The creation process should continue. In this case the mouse is captured.

#### **CREATE\_STOP**

The creation process has terminated normally. In this case the mouse is no longer captured and input is then allowed again from any viewport.

#### **CREATE\_ABORT**

The creation process has been aborted. The system will delete the created object and node.

### **Prototype:**

**virtual int override(int mode)**

### **Remarks:**

Implemented by the Plug-In.

This method is used to override the default drag mode. Most plug-in will not need to replace the default implementation of this method. What this does is change the way the messages are sent relative to the mouse clicking.

Normally the messages are sent as follows: When the user clicks down this generates a **MOUSE\_POINT** message. Then the user drags the mouse with the button down and a series of **MOUSE\_MOVE** messages are sent. When they let up on the mouse button a **MOUSE\_POINT** messages is generated. Then as the mouse is moved a series of **MOUSE\_MOVE** messages are sent. Then they click down on the mouse again, but this time a point message is not generated until the button is released. All future points are then only sent after the mouse button has been pressed and released.

### **Parameters:**

#### **int mode**

The current drag mode. See below.

**Return Value:**

One of the following drag modes should be returned:

**CLICK\_MODE\_DEFAULT**

Returned to indicate the use of the system mouse mode.

**CLICK\_DRAG\_CLICK**

This is the default behavior as described above.

**CLICK\_MOVE\_CLICK**

In this mode, the first point is entered by clicking the mouse button down and then letting it up. This generates point 0. In other words, a

**MOUSE\_POINT** message is only generated after the user has pressed and released the mouse button.

**CLICK\_DOWN\_POINT**

In this mode, point messages are sent on mouse-down only.

**Default Implementation:**

```
{ return mode; }
```

**Sample Code:**

A sample program that uses the override method is **\MAXSDK\SAMPLES\OBJECTS\SPLINE.CPP**. It uses **CLICK\_DOWN\_POINT**.

**Prototype:**

```
virtual BOOL StartNewCreation();
```

**Remarks:**

This method is available in release 3.0 and later only.

This gets called by the CreationManager to determine if the mouse proc is really starting a new object. The mouse proc for creating always returns **CREATE\_STOP**, which is how it keeps the mouse from being captured, and this function tells the system if it is really ready to start a new object. Thus, this is called only if the mouse proc returned **CREATE\_STOP** to see if the object is really in a state to start a new node.

**Return Value:**

TRUE if the mouse proc is ready to start a new object; otherwise FALSE.

**Default Implementation:**

```
{ return TRUE; }
```

**Prototype:**

```
virtual BOOL TolerateOrthoMode();
```

**Remarks:**

This method is available in release 3.0 and later only.

Called by the system to determine if ortho mode makes sense for this creation.

Typcially this only makes sense for splines and NURBS curves.

**Return Value:**

TRUE if ortho mode is okay; otherwise FALSE.

**Default Implementation:**

```
{ return FALSE; }
```

## List of Parameter Types for Shape Interpolation

See Also: [Class ShapeObject](#).

One of the following values:

### **PARAM\_SIMPLE**

Parameter space based on segments. This simple interpolation is interpolating based on parameter space -- If a spline has 4 segments, the first segment is parameter values 0-0.25, the second 0.25-0.5, the third 0.5-0.75 and the fourth 0.75-1.0. This is regardless of the length of each segment.

### **PARAM\_NORMALIZED**

Parameter space normalized to curve length. This interpolation normalizes the parameter space to distance along the length of a spline. So parameter space 0 is the start, 1.0 is the end and 0.5 is halfway along the actual length of the curve.

## Structure CameraState

See Also: [Class GenCamera](#).

```
struct CameraState {
```

```
    BOOL isOrtho;
```

Flag to indicate if the camera uses orthographic projection (TRUE) or perspective (FALSE).

```
    float fov;
```

The camera field-of-view in radians.

```
    float tdist;
```

Target distance for free cameras.

```
    BOOL horzLine;
```

Horizon line display state.

```
    int manualClip;
```

Flag to indicate if camera has manual clipping enabled.

```
    float hither;
```

Hither clipping plane distance.

```
    float yon;
```

Yon clipping plane distance.

```
    float nearRange;
```

Near camera range radius.

```
    float farRange;
```

Far camera range radius.

```
};
```

Note: The camera looks down the negative Z axis, with X to the right and Y up.

## Class ModContextEnumProc

See Also: [Class ModContext](#).

class ModContextEnumProc

### Description:

Callback object used with **Modifier::EnumModContexts()**. The **proc()** method is called by the system.

### Methods:

### Prototype:

```
virtual BOOL proc(ModContext *mc)=0
```

### Remarks:

Implemented by the Plug-In.

This is called by **Modifier::EnumModContexts()**.

### Parameters:

**ModContext \*mc**

The ModContext.

### Return Value:

Return FALSE to stop, TRUE to continue.

See Also: **Modifier::EnumModContexts()**, [Modifier](#).

## Structure TMComponentsArg

See Also: [Class Control](#), [Class Matrix3](#), [Class Point3](#), [Class Interval](#), [Class Quat](#), [Class ScaleValue](#).

```
struct TMComponentsArg
```

### Description:

This structure is available in release 4.0 and later only.

This structure is for collecting the return results of

**Control::GetLocalTMComponents**. Position, Rotation, or Scale, controllers will put results at the respective component when the corresponding pointer is not NULL.

### Function:

**TMComponentsArg():position(0),rotation(0),scale(0),rotRep(kUnkn**

### Remarks:

Constructor.

### Function:

**TMComponentsArg(Point3\* pos, Interval\* posInv, float\* rot, Interval\* rotInv, ScaleValue\* scl, Interval\* sclInv) : position(pos),posValidity(posInv),rotation(rot),rotValidity(rotInv), scale(scl),sclValidity(sclInv);**

### Remarks:

Constructor.

### Data:

**Point3\* position;**

If not NULL this is the position.

**Interval\* posValidity;**

If not NULL this points to the validity interval for the position.

**float\* rotation;**

If not NULL this is the rotation and should be a float[4].

**Interval\* rotValidity;**

If not NULL this points to the validity interval for the rotation.

**RotationRep rotRep;**

The rotation representation. This defines what the 4 numbers in the **rotation** array mean. One of the following enum values:

**kXYZ** - Same as **EULERTYPE\_XYZ**

**kXZY** - Same as **EULERTYPE\_XZY**

**kYZX** - Same as **EULERTYPE\_YZX**

**kYXZ** - Same as **EULERTYPE\_YXZ**

**kZXY** - Same as **EULERTYPE\_ZXY**

**kZYG** - Same as **EULERTYPE\_ZYG**

**kXYX** - Same as **EULERTYPE\_XYX**

**kYZY** - Same as **EULERTYPE\_YZY**

**kZXZ** - Same as **EULERTYPE\_ZXZ**

**kQuat** - A quaternion representation.

**kUnknown** – An unknown representation.

**ScaleValue\* scale;**

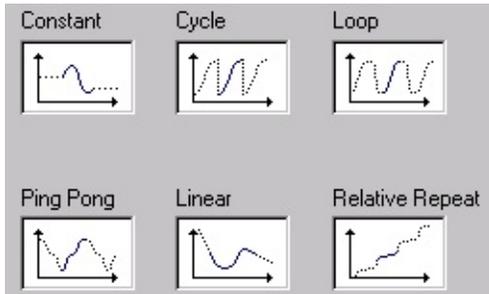
If non-NULL this is the ScaleValue.

**Interval\* sclValidity;**

The validity interval for the ScaleValue.

## List of Out of Range Types.

Out of Range Types provide several methods of extrapolating the pattern of key dots in a track. These patterns are applied to the animation outside the range of all keys in the track. The list of options is below:



One of the following values:

### **ORT\_CONSTANT**

The tracks values before or after the range of keys remains constant.

### **ORT\_CYCLE**

This causes the key pattern to repeat cyclically.

### **ORT\_LOOP**

This is the same as **ORT\_CYCLE** with continuity.

### **ORT\_OSCILLATE**

This is referred to as "Ping-Pong" in the 3ds max user interface. This reverses the range of keys values to cause the pattern to oscillate.

### **ORT\_LINEAR**

This takes the slope at the end key in the range and extrapolate with that slope.

### **ORT\_IDENTITY**

3ds max will only set this ORT for Ease Curves. This only is used when mapping time to time. The slope will be set to one (a 45 degree diagonal starting at the end of the key range).

### **ORT\_RELATIVE\_REPEAT**

This causes the key pattern to repeat with the first key taking off where the last key left off.

## Class InitJointData

See Also: [Class Control](#).

class InitJointData

### Description:

This class is passed to **Control::InitIKJoints()** which is called when importing R4 3DS files that have IK joint data.

### Data Members:

public:

**BOOL active[3];**

The joint active settings. Index 0=X, 1=Y, 2=Z.

**BOOL limit[3];**

The joint limit settings. Index 0=X, 1=Y, 2=Z.

**BOOL ease[3];**

The joint ease settings. Index 0=X, 1=Y, 2=Z.

**Point3 min, max, damping;**

The joint min, max and damping settings.

## Class InitJointData2

See Also: [Class InitJointData](#)

```
class InitJointData2 : public InitJointData
```

### Description:

This class is passed to **Control::InitIKJoints2()** which is called when importing R4 3DS files that have IK joint data. This class contains the added preferredAngle parameter.

### Data Members:

public:

**Point3 preferredAngle;**

The preferred angle.

**DWORD flags;**

Not used (must be 0), reserved for future expansion.

### Prototype:

**InitJointData2();**

### Remarks:

Constructor.

## Class DOFParams

See Also: [Class Control](#).

```
class DOFParams
```

### Description:

This class is available in release 2.0 and later only.

This structure is passed to the method **Control::GetDOFParams()**.

Controllers that support IK can provide information about their Degree Of Freedoms (DOFs) so that bones can display this information. The first 3 DOFs are assumed to be position and the next 3 are assumed to be rotation

### Data Members:

public:

**BOOL display[6];**

Indicates if this DOF should be displayed.

**Point3 axis[6];**

Specifies the DOF axis.

**Point3 pos[6];**

Specifies the base of the axis.

**BOOL limit[6];**

Indicates if the joint is limited at all.

**float min[6];**

Specifies the minimum limit.

**float max[6];**

Specifies the maximum limit.

**float curval[6];**

Specifies the current value of the parameter.

**BOOL sel[6];**

Indicates if the DOF should be highlighted.

**BOOL endEffector;**

Indicates if there is an end effector for this controller.

**Matrix3 eeTM;**

Specifies the world transformation matrix of the end effector (if present).

## Class IKClipObject

See Also: [Class Control](#).

class IKClipObject

### Description:

This class is used to store IK parameters that have been copied to a clipboard. The plug-in derives a class from this class to store their data and implements the methods that describe the creator object. The plug-in should also implement the **DeleteThis()** method to delete the instance of the class.

### Methods:

#### Prototype:

**virtual SClass\_ID SuperClassID()=0;**

#### Remarks:

Implemented by the Plug-In.

Returns the super class ID of the creator of the clip object.

#### Prototype:

**virtual Class\_ID ClassID()=0;**

#### Remarks:

Implemented by the Plug-In.

Returns the class ID of the creator of the clip object.

#### Prototype:

**virtual void DeleteThis()=0;**

#### Remarks:

Implemented by the Plug-In.

The system calls this method to delete the clip object after it has been used.

## Class CtrlHitRecord

class CtrlHitRecord

### Description:

This class provides a data structure used during controller gizmo hit-testing. All methods are implemented by the system.

### Friend Class:

class CtrlHitLog;

### Data Members:

public:

**INode \*nodeRef;**

This identifies the node the user has clicked on.

**DWORD distance;**

The 'distance' of the hit. What the distance actually represents depends on the rendering level of the viewport. For wireframe modes, it refers to the distance in the screen XY plane from the mouse to the sub-object component. In a shaded mode, it refers to the Z depth of the sub-object component. In both cases, smaller values indicate that the sub-object component is 'closer' to the mouse cursor.

**ulong hitInfo;**

A general unsigned long value. Most controllers will just need this to identify the sub-object element. The meaning of this value (how it is used to identify the element) is up to the plug-in.

**DWORD infoExtra;**

If the above **hitInfo** data member is not sufficient to describe the sub-object element this data member may be used as well.

### Methods:

### Prototype:

**CtrlHitRecord()**

### Remarks:

Constructor. The data members are initialized as follows: **next=NULL;**  
**distance=0;**

**hitInfo=0; nodeRef=NULL;**

**Prototype:**

**CtrlHitRecord(CtrlHitRecord \*nxt,INode \*nr, DWORD d,  
ulong inf, DWORD extra)**

**Remarks:**

Constructor. The data members are initialized to the data passed.

**Prototype:**

**CtrlHitRecord \*Next()**

**Remarks:**

Each **CtrlHitRecord** maintains a pointer to another **CtrlHitRecord**. This method returns the next hit record.

## List of Miscellaneous Control Functions and Templates

See Also: [Class StdControl](#), [List of Additional Controller Related Functions](#).

### Description:

The following functions are available for help with Out of Range Type (ORT) processing. All functions are implemented by the system.

### Prototype:

```
inline TimeValue CycleTime(Interval i,TimeValue t)
```

### Remarks:

Returns a TimeValue that is the specified time mod the interval length. The returned time is somewhere within the interval passed. This cycles the time so that it appears within the interval.

### Parameters:

#### **Interval i**

The interval the returned time is within.

#### **TimeValue t**

The time to cycle.

### Prototype:

```
inline int NumCycles(Interval i,TimeValue t)
```

### Remarks:

Returns the number of times the TimeValue cycles through the interval.

### Parameters:

#### **Interval i**

The interval the time is checked against.

#### **TimeValue t**

The time to check.

Note: Types that use these template functions must support the following operators:

**T + T, T - T, T \* float, T + float**

**Prototype:**

```
template <class T> T LinearExtrapolate(TimeValue t0, TimeValue  
t1,  
T &val0, T &val1, T &endVal)
```

**Remarks:**

Performs a linear extrapolation and returns the result.

**Parameters:**

**TimeValue t0**

The time you are extrapolating from.

**TimeValue t1**

The time you are extrapolating to.

**T &val0**

The first value you want to use to extrapolate from.

**T &val1**

The second value you want to use to extrapolate from.

**T &endVal**

The value at time **t0**.

**Return Value:**

The extrapolated value associated with time **t1**.

**Prototype:**

```
template <class T> T RepeatExtrapolate(Interval range,  
TimeValue t,  
T &startVal, T &endVal, T &cycleVal);
```

**Remarks:**

Performs a repeat extrapolation and returns the result. This will cycle the time **t** into the interval **range**.

**Parameters:**

**Interval range**

The range for which you are repeating over.

**TimeValue t**

The time you are extrapolating to.

**T &startVal**

The value at the start of the range.

**T &endVal**

The value at the end of the range.

**T &cycleVal**

The value of the function at the cycled point.

**Prototype:**

```
template <class T> T IdentityExtrapolate(TimeValue endPoint,  
TimeValue t,  
T &endVal);
```

**Remarks:**

Performs a linear extrapolation using a slope of one and a point and returns the result.

**Parameters:****TimeValue endPoint**

The end point time.

**TimeValue t**

The time to evaluate.

**T &endVal**

The value of the **endPoint** time.

**Prototype:**

```
Quat LinearExtrapolate(TimeValue t0, TimeValue t1, Quat &val0,  
Quat &val1, Quat &endVal);
```

**Remarks:**

The Quat version of above.

**Prototype:**

```
Quat RepeatExtrapolate(Interval range, TimeValue t, Quat  
&startVal,  
Quat &endVal, Quat &cycleVal);
```

**Remarks:**

The Quat version of above.

**Prototype:**

**Quat IdentityExtrapolate(TimeValue endPoint, TimeValue t, Quat &endVal );**

**Remarks:**

The Quat version of above.

**Prototype:**

**template <class T> T LinearInterpolate(const T &v0,const T &v1,float u)**

**Remarks:**

The functions performs a linear interpolation between **v0** and **v1** using **u** as the interpolation parameter.

**Parameters:**

**const T &v0**

The first value.

**const T &v1**

The second value.

**float u**

The interpolation parameter in the range 0 to 1.

**Prototype:**

**inline Quat LinearInterpolate(const Quat &v0,const Quat &v1,float u)**

**Remarks:**

The Quat version of above.

**Prototype:**

**inline ScaleValue LinearInterpolate(const ScaleValue &v0, const ScaleValue &v1,float u)**

**Remarks:**

The ScaleValue version of above.

**Prototype:**

**inline Interval TestInterval(Interval iv, DWORD flags)**

**Remarks:**

This function take the interval passed and checks the flags and modifies the interval based on the state of the flags.

**Parameters:****Interval iv**

The interval to modify.

**DWORD flags**

One of the following values:

**TIME\_INCLEFT**

**TIME\_INCRIGHT**

**Return Value:**

The revised interval.

**Prototype:**

**inline Quat ScaleQuat(Quat q, float s)**

**Remarks:**

Returns a quaternion scaled by the specified factor. This scales the 'angle' of the quaternion by **s**.

**Parameters:****Quat q**

The quaternion to scale.

**float s**

The scale factor.

## Class ImpInterface

See Also: [Class ImpNode](#), [Class Interval](#), [Class\\_ID](#), [Class GenLight](#), [Class GenCamera](#).

class ImpInterface

### Description:

Import Interface class. Methods of this class allow the plug-in to create nodes in the scene, create camera and light objects, and create objects by specifying a super class ID and Class ID. Methods are also available to bind target nodes to their Look At nodes. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual ImpNode \*CreateNode() = 0;**

#### Remarks:

Creates a new node. Methods of **ImpNode** may be used to assign properties to the node. See **AddNodeToScene()** to add a node to the scene given its **ImpNode** pointer.

#### Return Value:

An **ImpNode** pointer that may be used to set properties of the node.

#### Prototype:

**virtual void RedrawViews() = 0;**

#### Remarks:

Redraws the 3ds max viewports.

#### Prototype:

**virtual GenCamera \*CreateCameraObject(int type) = 0;**

#### Remarks:

Creates a camera object and returns a pointer to it. The GenCamera pointer may be used to set the properties of the camera object.

#### Parameters:

**int type**

One of the following values:

**FREE\_CAMERA**

**TARGETED\_CAMERA**

**Prototype:**

**virtual Object \*CreateTargetObject() = 0;**

**Remarks:**

Creates a target object and returns a pointer to it.

**Prototype:**

**virtual GenLight \*CreateLightObject(int type) = 0;**

**Remarks:**

Creates a light object and returns a pointer to it. The GenLight pointer may be used to set the properties of the light object.

**Parameters:**

**int type**

One of the following values:

**OMNI\_LIGHT** - Omnidirectional

**TSPOT\_LIGHT** - Targeted

**DIR\_LIGHT** - Directional

**FSPOT\_LIGHT** - Free

**Prototype:**

**virtual void \*Create(SClass\_ID sclass, Class\_ID classid)=0;**

**Remarks:**

Creates an object given its Super Class ID and its Class ID.

**Parameters:**

**SClass\_ID sclass**

The super class ID of the node to create.

**Class\_ID classid**

The unique class ID of the node to create.

**Return Value:**

A pointer to the item.

**Prototype:**

```
virtual int BindToTarget(ImpNode *laNode, ImpNode  
*targNode)=0;
```

**Remarks:**

This method binds a node to a target using a Look At controller. This is typically used with target spotlights and cameras to bind them to their target node.

**Parameters:**

**ImpNode \*laNode**

The node that will have the Look At controller assigned.

**ImpNode \*targNode**

The target node.

**Return Value:**

Nonzero if successful; otherwise 0.

**Prototype:**

```
virtual void AddNodeToScene(ImpNode *node)=0;
```

**Remarks:**

Adds a node to the scene given its ImpNode pointer.

**Parameters:**

**ImpNode \*node**

The node to add to the scene.

**Prototype:**

```
virtual void SetAnimRange(Interval& range)=0;
```

**Remarks:**

Sets the animation range for the node.

**Parameters:**

**Interval& range**

Specifies the animation range.

**Prototype:**

**virtual Interval GetAnimRange()=0;**

**Remarks:**

Retrieves the animation range for the node (as an Interval).

**Prototype:**

**virtual void SetEnvironmentMap(Texture \*txm)=0;**

**Remarks:**

Sets the current environment map to the specified map.

**Parameters:**

**Texture \*txm**

The map to set.

**Prototype:**

**virtual void SetAmbient(TimeValue t, Point3 col)=0;**

**Remarks:**

Sets the ambient light color at the specified time.

**Parameters:**

**TimeValue t**

The time to set the color.

**Point3 col**

The light color to set.

**Prototype:**

**virtual void SetBackGround(TimeValue t,Point3 col)=0;**

**Remarks:**

Sets the background color at the specified time.

**Parameters:**

**TimeValue t**

The time to set the color.

**Point3 col**

The light color to set.

**Prototype:**

**virtual void SetUseMap(BOOL onoff)=0;**

**Remarks:**

Sets the state of the environment 'Use Map' toggle.

**Parameters:**

**BOOL onoff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual void AddAtmosphere(Atmospheric \*atmos)=0;**

**Remarks:**

Adds the specified atmospheric effect to the environment.

**Parameters:**

**Atmospheric \*atmos**

The atmospheric effect. See [Class Atmospheric](#).

**Prototype:**

**virtual int NewScene()=0;**

**Remarks:**

This method deletes all existing geometry in the scene.

## Class ExpInterface

See Also: [Class IScene](#).

class ExpInterface

### **Description:**

Export Interface class. A data member of this class allows the plug-in to enumerate all nodes in the scene.

### **Data Members:**

**IScene \*theScene;**

A pointer to the scene. See [Class IScene](#).

## Class SFXParamDlg

See Also: [Class Effect](#), [Class Atmospheric](#) , [Class Sampler](#).

class SFXParamDlg : public InterfaceServer

### Description:

This class is available in release 3.0 and later only.

An instance of this class is returned by a rendering effect, atmospheric plug-in, or sampler when it is asked to put up its rollout page in the user interface.

### Methods:

public:

### Prototype:

**virtual Class\_ID ClassID()=0;**

### Remarks:

Implemented by the Plug-In.

Returns the unique Class\_ID of this item.

### Prototype:

**virtual void SetThing(ReferenceTarget \*m)=0;**

### Remarks:

Implemented by the Plug-In.

This sets the current filter, sampler, atmospheric or rendering effect being edited to the one passed and updates the user interface controls to reflect the state of the new 'thing'.

### Parameters:

**ReferenceTarget \*m**

The effect to save as current.

### Prototype:

**virtual ReferenceTarget\* GetThing()=0;**

### Remarks:

Implemented by the Plug-In.

This returns the current filter, sampler, atmospheric or rendering effect being edited.

**Prototype:**

**virtual void SetTime(TimeValue t);**

**Remarks:**

Implemented by the Plug-In.

This method is called when the current time has changed. This gives the developer an opportunity to update any user interface data that may need adjusting due to the change in time.

**Parameters:**

**TimeValue t**

The new current time.

**Default Implementation:**

```
{}
```

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Implemented by the Plug-In.

Deletes this instance of the class.

**Prototype:**

**virtual INT\_PTR Execute(int cmd, ULONG arg1=0, ULONG arg2=0, ULONG arg3=0);**

**Remarks:**

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new cmd numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:**

**int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

**Default Implementation:**

```
{ return 0; }
```

## Class IRendParams

See Also: [Class TimeChangeCallback](#), [Class RendPickProc](#).

class IRendParams : public InterfaceServer

### Description:

This is the interface given to a renderer, or atmospheric effect when it needs to display its parameters. For a renderer, the Render Scene dialog may be extended using this class. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual TimeValue GetTime()=0;**

#### Remarks:

Returns the current position of the frame slider.

#### Prototype:

**virtual void RegisterTimeChangeCallback(TimeChangeCallback \*tc)=0;**

#### Remarks:

Register a callback object that will get called every time the user changes the frame slider.

#### Parameters:

**TimeChangeCallback \*tc**  
The callback object to register.

#### Prototype:

**virtual void  
UnRegisterTimeChangeCallback(TimeChangeCallback \*tc)=0;**

#### Remarks:

Un-registers a callback object registered using  
**RegisterTimeChangeCallback()**.

#### Parameters:

## **TimeChangeCallback \*tc**

The callback object to un-register.

### **Prototype:**

```
virtual MtlBase *DoMaterialBrowseDlg(HWND hParent,DWORD  
flags,BOOL &newMat,BOOL &cancel)=0;
```

### **Remarks:**

Brings up the material browse dialog allowing the user to select a material.

### **Parameters:**

#### **HWND hParent**

The parent window handle.

#### **DWORD flags**

See [List of Material Browser Flags](#).

#### **BOOL &newMat**

Set to TRUE if the material is new OR cloned; otherwise FALSE.

#### **BOOL &cancel**

Set to TRUE if the user cancels the dialog; otherwise FALSE.

### **Return Value:**

The material returned will be NULL if the user selects 'None'

### **Prototype:**

```
virtual HWND AddRollupPage(HINSTANCE hInst, TCHAR  
*dlgTemplate, DLGPROC dlgProc, TCHAR *title, LPARAM  
param=0,DWORD flags=0, int category =  
ROLLUP_CAT_STANDARD)=0;
```

### **Remarks:**

This method adds rollup pages to the dialog and returns the window handle of the page.

### **Parameters:**

#### **HINSTANCE hInst**

The DLL instance handle of the plug-in.

#### **TCHAR \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here. This may be later retrieved using the GetWindowLong() call from the Windows API.

**DWORD flags=0**

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the **RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USERREPLACEDCAT**. This is mainly done, so that this system works with param maps as well.

**Return Value:**

The window handle of the rollup page.

**Prototype:**

```
virtual HWND AddRollupPage(HINSTANCE hInst,  
DLGTEMPLATE *dlgTemplate, DLGPROC dlgProc, TCHAR  
*title, LPARAM param=0,DWORD flags=0, int category =  
ROLLUP_CAT_STANDARD)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method adds rollup pages to the dialog and returns the window handle of the page. This method is currently not being used.

**Parameters:****HINSTANCE hInst**

The DLL instance handle of the plug-in.

**DLGTEMPLATE \*dlgTemplate**

The dialog template for the rollup page.

**DLGPROC dlgProc**

The dialog proc to handle the message sent to the rollup page.

**TCHAR \*title**

The title displayed in the title bar.

**LPARAM param=0**

Any specific data to pass along may be stored here. This may be later retrieved using the GetWindowLong() call from the Windows API.

**DWORD flags=0****APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**int category = ROLLUP\_CAT\_STANDARD**

The category parameter provides flexibility with regard to where a particular rollup should be displayed in the UI. RollupPanels with lower category fields will be displayed before RollupPanels with higher category fields. For RollupPanels with equal category value the one that was added first will be displayed first. Although it is possible to pass any int value as category there exist currently 5 different category defines: **ROLLUP\_CAT\_SYSTEM**, **ROLLUP\_CAT\_STANDARD**, and **ROLLUP\_CAT\_CUSTATTRIB**.

When using **ROLLUP\_SAVECAT**, the rollup page will make the provided category sticky, meaning it will not read the category from the

**RollupOrder.cfg** file, but rather save the category field that was passed as argument in the **CatRegistry** and in the **RollupOrder.cfg** file.

The method will take the category of the replaced rollup in case the flags argument contains **ROLLUP\_USEREPLACEDCAT**. This is mainly

done, so that this system works with param maps as well.

**Return Value:**

The window handle of the rollup page.

**Prototype:**

**virtual void DeleteRollupPage(HWND hRollup)=0;**

**Remarks:**

Removes a rollup page and destroys it.

**Parameters:**

**HWND hRollup**

The handle of the rollup window. This is the handle returned from **AddRollupPage()**.

**Prototype:**

**virtual void RollupMouseMessage(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)=0;**

**Remarks:**

This allows hand cursor scrolling when the user clicks the mouse in an unused area of the dialog. When the user mouses down in dead area of the dialog, the plug-in should pass mouse messages to this function which will pass them on to the rollup.

Note: In 3ds max 2.0 and later only use of this method is no longer required -- the functionality happens automatically.

**Parameters:**

**HWND hDlg**

The window handle of the dialog.

**UINT message**

The message sent to the dialog proc.

**WPARAM wParam**

Passed in to the dialog proc. Pass along to this method.

**LPARAM lParam**

Passed in to the dialog proc. Pass along to this method.

**Prototype:**

```
virtual void SetPickMode(RendPickProc *proc)=0;
```

**Remarks:**

This will set the command mode to a standard pick mode. The callback implements hit testing and a method that is called when the user actually picks an item.

**Parameters:**

**RendPickProc \*proc**

The callback object. See [Class RendPickProc](#).

**Prototype:**

```
virtual void EndPickMode()=0;
```

**Remarks:**

If a plug-in is finished editing its parameters it should not leave the user in a pick mode. This method will flush out any pick modes in the command stack.

**Prototype:**

```
virtual void PutMtlToMtlEditor(MtlBase *mb)=0;
```

**Remarks:**

When a plugin has a Texmap, clicking on the button associated with that map should cause this routine to be called.

**Parameters:**

**MtlBase \*mb**

The **MtlBase** (**Texmap** or **Mtl**) to put to the materials editor.

**Prototype:**

```
virtual float GetMaxPixelSize() = 0;
```

**Remarks:**

This method is used internally.

**Prototype:**

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0);
```

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:**

**int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

## Class ShadeOutput

See Also: [Class ShadeContext](#), [Class Color](#).

```
class ShadeOutput : public BaseInterfaceServer
```

### Description:

An instance of this class is a data member of the **ShadeContext**. This is used to contain the computed color and transparency of the pixel being shaded by a material. All methods of this class are implemented by the system.

### Data Members:

public:

**ULONG flags;**

These flags are not currently used.

**Color c;**

Shaded color of the pixel.

**Color t;**

Transparency of the pixel.

**float ior;**

Index of refraction of the pixel.

**int gbufId;**

The G-buffer ID. This allows the **MixIn()** method to pick the id of the material which was blended in the highest proportion.

### Methods:

#### Prototype:

```
void MixIn(ShadeOutput& a, float f);
```

#### Remarks:

This method is used to blend the output of two texmaps, for example, in the Mix texmap. The function is :

```
void ShadeOutput::MixIn(ShadeOutput &a, float f) {  
    if (f<=0.0f) {  
        (*this) = a;  
        return;  
    }  
}
```

```

else if (f>=1.0f) {
    return;
}
else {
    float s = 1.0f - f;
    flags |= a.flags;
    c = s*a.c + f*c;
    t = s*a.t + f*t;
    ior = s*a.ior + f*ior;
    if (f<=0.5f) gbufId = a.gbufId;
}
}

```

This does a blend of **a** with (**\*this**). This blend is applied to the color, transparency, and index of refraction. The flags of are OR'ed together.

**Parameters:**

**ShadeOutput& a**

The output of the texmap to blend in.

**float f**

The amount to blend in, i.e.:

**a.MixIn(b, 1.0f)** results in 100% of **a** and none of **b**.

**Prototype:**

**void Reset(int n = -1)**

**Remarks:**

Implemented by the System.

This method resets the data member such that: **c** is set to black, **t** is set to black, **ior** is set to 1.0, **gbufId** is set to 0, and the **flags** are set to 0.

**Parameters:**

**int n = -1**

By supplying a negative value this method will clear elements but leave the number of elements unchanged.

# Class TexHandleMaker

See Also: [Class TexHandle](#), [Class Bitmap](#), [Class BitmapInfo](#).

class TexHandleMaker

## Description:

This class provides several ways to create a texture handle. The handle may be created from a 3ds max bitmap or a Windows Device Independent Bitmap. This class also provides methods to determine the desired size of the bitmap.

## Methods:

### Prototype:

```
virtual TexHandle* CreateHandle(Bitmap *bm, int symflags=0,  
int extraFlags=0)=0;
```

### Remarks:

Implemented by the System

This method is called to create a texture handle from a 3ds max bitmap.

### Parameters:

**Bitmap \*bm**

The bitmap to create a handle to.

**int symflags=0**

See [List of Texture Symmetry Flags](#).

**int extraFlags=0;**

One of the following values:

**EX\_MULT\_ALPHA**

Set this flag if alpha is **not** premultiplied in the Bitmap.

**EX\_RGB\_FROM\_ALPHA**

Set this flag to make the map using the alpha channel of the bitmap to define the gray level.

**EX\_OPAQUE\_ALPHA**

Specifies to make the map using opaque alpha.

**EX\_ALPHA\_FROM\_RGB**

Specifies to make alpha from the intensity of the map.

**Return Value:**

A pointer to the texture handle.

**Prototype:**

```
virtual TexHandle* CreateHandle(BITMAPINFO *bminf, int  
symflags=0, int extraFlags=0)=0;
```

**Remarks:**

Implemented by the System

This method is called to create a texture handle from a 32 bit Windows Device Independent Bitmap.

**Parameters:**

**BITMAPINFO \*bminf**

The bitmap to create a handle to.

**int symflags=0**

See [List of Texture Symmetry Flags](#).

**int extraFlags=0;**

One of the following values:

**EX\_MULT\_ALPHA**

Set this flag if alpha is **not** premultiplied in the Bitmap.

**EX\_RGB\_FROM\_ALPHA**

Set this flag to make the map using the alpha channel of the bitmap to define the gray level.

**EX\_OPAQUE\_ALPHA**

Specifies to make the map using opaque alpha.

**EX\_ALPHA\_FROM\_RGB**

Specifies to make alpha from the intensity of the map.

**Return Value:**

A pointer to the texture handle.

**Prototype:**

```
virtual BITMAPINFO *BitmapToDIB(Bitmap *bm, int symflags,  
int extraFlags, BOOL forceW=0, BOOL forceH=0)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

This method creates a 32 bit Windows Device Independent Bitmap with the specified symflags and extraflags already incorporated and returns a pointer to the associated BitmapInfo.

**Parameters:****Bitmap \*bm**

Points to the bitmap to create the handle to.

**int symflags**

See [List of Texture Symmetry Flags](#).

**int extraFlags**

One of the following values:

**EX\_MULT\_ALPHA**

Set this flag if alpha is **not** premultiplied in the Bitmap.

**EX\_RGB\_FROM\_ALPHA**

Set this flag to make the map using the alpha channel of the bitmap to define the gray level.

**EX\_OPAQUE\_ALPHA**

Specifies to make the map using opaque alpha.

**EX\_ALPHA\_FROM\_RGB**

Specifies to make alpha from the intensity of the map.

**BOOL forceW=0**

If this parameter is non-zero it is used as the width of the final DIB.

**BOOL forceH=0**

If this parameter is non-zero it is used as the height of the final DIB

**Prototype:**

```
virtual TexHandle* MakeHandle(BITMAPINFO* bminf)=0;
```

**Remarks:**

This method is available in release 4.0 and later only.

Returns a texture handle made from a 32 bit DIB that has the symflags, and extraflags already incorporated. This takes ownership of the BITMAPINFO\*.

**Parameters:****BITMAPINFO\* bminf**

Points to the BitmapInfo for the DIB.

**Prototype:****virtual BOOL UseClosestPowerOf2()=0;****Remarks:**

This method is available in release 4.0 and later only.

Returns TRUE if the bitmap does not need to be square and FALSE if it does need to be square.

**Prototype:****virtual int Size()=0;****Remarks:**

Implemented by the System.

This method may be called to determine the desired size for the bitmap. The system ultimately needs a square bitmap that is a power of 2 in width and height. If you already have a bitmap around, just pass it in to

**CreateHandle()** and it will be converted. If you are creating a bitmap from scratch (i.e. a procedural texture), then you should make it **Size()** in width in height, and save the system an extra step. In either case you own your bitmap, and are responsible for ultimately freeing it.**Return Value:**

The size of the desired bitmap.

## Class NameAccum

See Also: [Class Texmap](#).

class NameAccum

### Description:

This class provides a method **AddMapName()** that is called when a Texmap is attempting to load its map and it cannot be found.

### Methods:

#### Prototype:

```
virtual void AddMapName(TCHAR *name)=0;
```

#### Remarks:

Implemented by the System.

This method is called to add the name of a map that was not found.

#### Parameters:

**TCHAR \*name**

The name to add.

# List of Procedural Texture Clamping, Noise and Misc Functions

See Also: [Class Texmap](#), [Class AColor](#), [Class Color](#), [Class Point2](#), [Class Point3](#).

The following functions are available for use in the creation of procedural textures. These are based on the code provided in the book:

Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, Texturing and Modeling A Procedural Approach (Cambridge, MA: Academic Press, Inc., 1994). ISBN: 0-12-228760-6.

Consult this book for additional insight into the use of these functions.

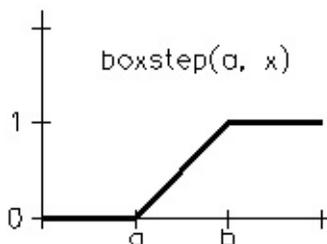
## Prototype:

```
float boxstep(float a, float b, float x);
```

## Remarks:

This function returns 0 if  $x$  is less than  $a$ , a linear interpolation between 0 and 1 if  $x$  is greater than or equal to  $a$  and less than or equal to  $b$ , and 1 if  $x$  is greater than  $b$ . This function is a smoother version of `step()` using a linear ramp. The function `boxstep()` is defined like this:

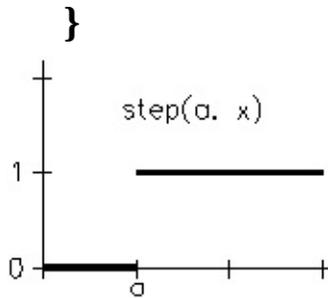
```
float boxstep(float a, float b, float x) {  
    return clamp((x-a)/(b-a),0.0f, 1.0f);  
}
```



## The `boxstep()` function.

For comparison, note that `step()` returns values of 0 when  $x$  is less than  $a$  and 1 when  $x$  is greater than or equal to  $a$ . This function is not part of the SDK but `step()` is defined like this:

```
float step(float a, float x) {  
    return (float)(x >= a);  
}
```



**The step() function.**

**Parameters:**

**float a**

The limit for the **x** value where the function will return 0.

**float b**

The limit for the **x** value where the function will return 1.

**float x**

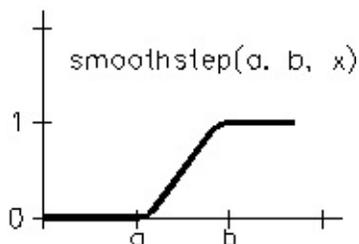
A floating point value.

**Prototype:**

**float smoothstep(float a, float b, float x);**

**Remarks:**

This function is similar to **step()**, but instead of a sharp transition from 0 to 1 at a specified threshold, it makes a gradual transition from 0 to 1 beginning at threshold **a** and ending at threshold **b**. To do this, this function uses a cubic function whose slope is 0 at **a** and **b** and whose value is 0 at **a** and 1 at **b**. This function thus provides a still smoother version of **step()** using a cubic spline. The **smoothstep()** function is used (instead of **step()**) in many procedural textures because sharp transitions often result in artifacts.



**The smoothstep() function.**

**Parameters:**

**float a**

The limit for the **x** value where the function will return 0.

**float b**

The limit for the **x** value where the function will return 1.

**float x**

A floating point value.

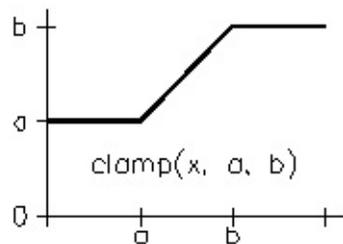
**Prototype:**

**float clamp(float x, float a, float b);**

**Remarks:**

This function returns **a** when **x** is less than **a**, the value of **x** when **x** is between **a** and **b**, and the value **b** when **x** is greater than **b**. The function **clamp()** is defined as follows:

```
float clamp( float x, float a, float b) {  
    return (x < a ? a : (x > b ? b : x));  
}
```



**The clamp() function.**

**Parameters:**

**float x**

A floating point value.

**float a**

A floating point value.

**float b**

A floating point value.

**Prototype:**

**float mod(float x, float m);**

**Remarks:**

This function returns **x Mod m** and handles negatives correctly. The standard

math functions **fmod()** and **fmodf()** return negative results if the first operand, **x**, is negative. This function will return the positive remainder when dividing **x** by **m**.

**Parameters:**

**float x**

A floating point value.

**float m**

A floating point value.

**Prototype:**

**int mod(int x, int m);**

**Remarks:**

This function returns  $x \text{ Mod } m$  and handles negatives correctly.

This function returns  $x \text{ Mod } m$  and handles negatives correctly. The standard C remainder operator `%` return negative results if the first operand, **x**, is negative. This function will return the positive remainder when dividing **x** by **m**.

**Parameters:**

**int x**

An integer value.

**int m**

An integer value.

**Prototype:**

**float sramp(float x, float a, float b, float d);**

**Remarks:**

This function makes a sort of straight segment S curve.

**sramp()** is **a** for  $x$  less than **a-d** and **b** for  $x$  greater than **b+d**.

for  $a+d < x < b-d$  **sramp(x) = x**

for  $a-d < x < a+d$  **sramp()** makes a smooth transition (parabolic) from  $\text{sramp}' = 0$  to  $\text{sramp}' = 1$

for  $b-d < x < b+d$  **sramp()** makes a smooth transition (parabolic) from  $\text{sramp}' = 1$  to  $\text{sramp}' = 0$

**Parameters:****float x**

A floating point value.

**float a**

A floating point value.

**float b**

A floating point value.

**float d**

A floating point value.

**Prototype:****float threshold(float x, float a, float b);****Remarks:**

This function returns 0 if **x** is less than **a**, 1 if **x** is greater than **b**, otherwise it returns **x**.

**Parameters:****float x**

A floating point value.

**float a**

A floating point value.

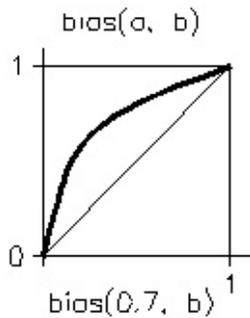
**float b**

A floating point value.

**Prototype:****float bias(float a, float b);****Remarks:**

This function performs a mapping across the unit interval [0, 1] where the result is within the unit interval. That is, if **b** is within the interval 0 to 1, the result of **bias()** is within the interval 0 to 1. This function is defined such that **bias(a, 0.5)=a**. The function looks like:

```
float bias(float a, float b) {  
    return (float)pow(a, log(b) / log(0.5f));  
}
```



**The bias() function.**

**Parameters:**

**float a**

The parameter that determines the shape of the mapping.

**float b**

A floating point value.

**Prototype:**

**float gain(float a, float b);**

**Remarks:**

This function performs a mapping across the unit interval  $[0, 1]$  where the result is within the unit interval. That is, if **b** is within the interval 0 to 1, the result of **gain()** is within the interval 0 to 1. This function is defined such that **gain(a, 0.5)=a**. Above and below 0.5, this function consists of two scaled down **bias()** curves forming an S-shaped curve. The function looks like:

```

float gain(float a, float b)
{
  float p = (float)log(1.0f - b) / (float)log(0.5f);

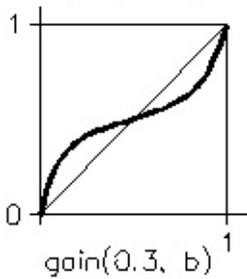
  if (a < .001f)
    return 0.f;
  else if (a > .999f)
    return 1.0f;
  if (a < 0.5f)
    return (float)pow(2 * a, p) / 2;
  else

```

```

    return 1.0f - (float)pow(2.0 * (1. - a), (double)p) / 2;
}
gain(a, b)

```



**The gain() function.**

**Parameters:**

**float a**

The parameter that determines the shape of the mapping.

**float b**

A floating point value.

**The following are noise functions over 1, 2, and 3 dimensions:**

These are all simply noise functions of different dimension. They return values in the range [-1,1].

**Prototype:**

**float noise1(float arg);**

**Remarks:**

This function is an approximation of white noise blurred to dampen frequencies beyond some value. The return value is in the range [-1, 1].

**Parameters:**

**float arg**

A floating point value.

**Prototype:**

**float noise2(Point2 p);**

**Remarks:**

This function is an approximation of white noise blurred to dampen frequencies beyond some value. The return value is in the range [-1, 1].

**Parameters:**

**Point2 p**

A Point2 value.

**Prototype:**

**float noise3(Point3 p);**

**Remarks:**

This is a noise function over R3 -- implemented by a pseudo-random tricubic spline. This function is an approximation of white noise blurred to dampen frequencies beyond some value. The return value is in the range [-1, 1].

**Parameters:**

**Point3 p**

A Point3 value.

**Prototype:**

**float noise4(Point3 p, float time);**

**Remarks:**

This function is an approximation of white noise blurred to dampen frequencies beyond some value. The return value is in the range [-1, 1].

**Parameters:**

**Point3 p**

A Point3 value.

**float time**

A floating point value.

**Prototype:**

**float noise3DS(Point3 p);**

**Remarks:**

This is 3DStudio's Noise function: its only slightly different from **noise3()** scaled up by factor of 1.65 and clamped to -1,+1.

The following **#define** is also available:

```
#define NOISE(p) ((1.0f+noise3DS(p))*0.5f)
```

Macro to map the value returned from the **noise3DS()** function into interval [0,1].

**Parameters:**

**Point3 p**

A Point3 value.

**Prototype:**

**float turbulence(Point3& p, float freq);**

**Remarks:**

This turbulence function is a simple fractal generating loop built on top of the noise function. It is used to make marble, clouds, explosions, etc. It returns a value in the range [0, 1].

**Parameters:**

**Point3& p**

The input point.

**float freq**

A floating point frequency.

**Prototype:**

**int Perm(int v);**

**Remarks:**

This function simply uses **v** as a lookup into a table to return a different number. It only uses the low 9 bits of the number (0-512) and returns a number in that range.

**Parameters:**

**int v**

An integer value.

**Prototype:**

**float fBm1(float point, float H, float lacunarity, float octaves);**

**Remarks:**

This function is a fractional Brownian motion fractal (or fBm for short) that returns a floating point value. This version of the fBm is said to be "homogeneous" (the same everywhere) and "isotropic" (the same in all directions).

**Parameters:**

**float point**

The function is evaluated at this value.

**float H**

The fractal increment parameter. When  $H = 1$ , the function is relatively smooth; as  $H$  goes to 0, the function approaches white noise.

**float lacunarity**

The gap between successive frequencies. This is usually set to 2.0.

**float octaves**

The number of frequencies in the function.

**Prototype:**

**float fBm1(Point2 point, float H, float lacunarity, float octaves);**

**Remarks:**

This function is a fractional Brownian motion fractal (or fBm for short) that returns a floating point value.

**Parameters:****Point2 point**

The function is evaluated at this point.

**float H**

The fractal increment parameter. When  $H = 1$ , the function is relatively smooth; as  $H$  goes to 0, the function approaches white noise.

**float lacunarity**

The gap between successive frequencies. This is usually set to 2.0.

**float octaves**

The number of frequencies in the function.

**Prototype:**

**float fBm1(Point3 point, float H, float lacunarity, float octaves);**

**Remarks:**

This function is a fractional Brownian motion fractal (or fBm for short) that returns a floating point value.

**Parameters:****Point3 point**

The function is evaluated at this point.

### **float H**

The fractal increment parameter. When  $H = 1$ , the function is relatively smooth; as  $H$  goes to 0, the function approaches white noise.

### **float lacunarity**

The gap between successive frequencies. This is usually set to 2.0.

### **float octaves**

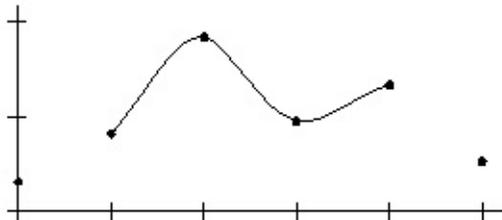
The number of frequencies in the function.

### **Prototype:**

**float spline(float x, int nknots, float \*knot);**

### **Remarks:**

This function is used to map a number into another number. The function is a one-dimensional Catmull-Rom interpolating spline through a set of knot values. The parameter of the spline is a floating point value. If  $x$  is 0, the result is the second knot value. If  $x$  is 1, the result is the final knot value. For values between 0 and 1, the value interpolates smoothly between the values of the knots from the second knot to the second to last knot. The first and last knot values determine the derivatives of the spline at the endpoint.



`spline(float x, int nknots, float *knot)`

### **The spline () function.**

### **Parameters:**

#### **float x**

A floating point value.

#### **int nknots**

The number of knots. Because the spline is a cubic polynomial there must be at least four knots.

#### **float \*knot**

An array of floating point knot values.

**Prototype:****Color color\_spline(float x, int nknots, Color \*knot);****Remarks:**

This function is used to map a number into a color. The function is a one-dimensional Catmull-Rom interpolating spline through a set of knot values. The parameter of the spline is a floating point value. If **x** is 0, the result is the second knot value. If **x** is 1, the result is the final knot value. For values between 0 and 1, the value interpolates smoothly between the values of the knots from the second knot to the second to last knot. The first and last knot values determine the derivatives of the spline at the endpoint.

**Parameters:****float x**

A floating point value.

**int nknots**

The number of knots. Because the spline is a cubic polynomial there must be at least four knots.

**Color \*knot**An array of **Color** knot values.**Prototype:****inline int FLOOR(float x);****Remarks:**

This function provides a faster version of the standard C function **floor()**. It returns a floating-point value representing the largest integer that is less than or equal to **x**.

**Parameters:****float x**

A floating point value.

**Prototype:****inline float frac(float x)****Remarks:**

This function returns the fraction (non-integer) part of the value passed. This is defined as:

```
{ return x - (float)FLOOR(x); }
```

**Parameters:**

**float x**

The value whose fractional portion is to be returned.

**Prototype:**

```
inline float fmax(float x, float y);
```

**Remarks:**

This function returns **x** if it is greater than **y**; otherwise it returns **y**.

**Parameters:**

**float x**

One of the floating point values used in the comparison.

**float y**

One of the floating point values used in the comparison.

**Prototype:**

```
inline float fmin(float x, float y);
```

**Remarks:**

This function returns **x** if it is less than **y**; otherwise it returns **y**.

**Parameters:**

**float x**

One of the floating point values used in the comparison.

**float y**

One of the floating point values used in the comparison.

**Prototype:**

```
inline AColor AComp(AColor cbot, AColor ctop);
```

**Remarks:**

This function performs an alpha-composite of **ctop** on top of **cbot**, assuming pre-multiplied alpha.

This is defined as:

```
{
```

```
    float ia = 1.0f - ctop.a;
```

```
    return (ctop + ia*cbot);  
}
```

**Parameters:**

**AColor cbot**

The color that is composited over.

**AColor ctop**

The color to composite on top.

**Return Value:**

The composited color.

**Prototype:**

```
void CellFunction(Point3 v,int n,float *dist,int  
*celIDs=NULL,Point3 *grads=NULL,float gradSmooth=0.0f);
```

**Remarks:**

This is the noise function used by the 3ds max Cellular texture. The idea is that there is a set of cells randomly distributed through space. This function returns the distances to the closest cells.

Developers using this function should refer to the following paper upon which this function is based. A Cellular Basis Function by Steven Worley in the SIGGRAPH 1996 Conference Proceedings.

**Parameters:**

**Point3 v**

The 3D input point.

**int n**

The number of elements in the arrays below.

**float \*dist**

A set of distances are returned here. This is the distance to the closest cell, the second closest cell, etc.

**int \*celIDs=NULL**

An optional array of integers to store cell IDs. This returns a value used to identify the cell. Since the functions works with a set of cells distributed through space, if two input points returned the same closest cell, then they would both have this same cell ID for their closest cell.

This is used in the Cellular texture to modulate the color. The cell color is varied by some random amount by using this value is used as the random seed. Thus the value is constant throughout the cell. See the function

**RandFromCellID(int id)** below.

**Point3 \*grads=NULL**

An optional array point Point3s used to get the partial derivatives with respect to X, Y and Z for the function. This is used for bump mapping.

**float gradSmooth=0.0f**

This equate to the 'Bump Smoothing' spinner in the Cellular texmap UI. The derivative of the function (the distance to the closest cell) has a discontinuity as it switches from cell to cell. Thus the gradients above have a discontinuity. This value basically smoothes off the discontinuity. The range is 0.0 to 1.0.

**Prototype:**

```
void FractalCellFunction(Point3 v,float iterations, float  
lacunarity,int n,float *dist,int *celIDs=NULL,Point3  
*grads=NULL,float gradSmooth=0.0f);
```

**Remarks:**

This is a fractal version of above. It has additional parameter for **iterations** and **lacunariy**.

**Parameters:**

**Point3 v**

The 3D input point.

**float iterations**

This corresponds to the 'Iterations' parameter in the Cellular UI. Varying this value gives different results.

**float lacunarity**

The is the 'Roughness' parameter in the Cellular UI. Varying this value gives different results.

**int n**

The number of elements in the arrays below.

**float \*dist**

A set of distances are returned here. This is the distance to the closest cell, the

second closest cell, etc.

**int \*celIDs=NULL**

An optional array of integers to store cell IDs. This returns a value used to identify the cell. Since the functions works with a set of cells distributed through space, if two input points returned the same closest cell, then they would both have this same cell ID for their closest cell.

This is used in the Cellular texture to modulate the color. The cell color is varied by some random amount by using this value is used as the random seed. Thus the value is constant throughout the cell. See the function **RandFromCellID(int id)** below.

**Point3 \*grads=NULL**

An optional array point Point3s used to get the partial derivatives with respect to X, Y and Z for the function. This is used for bump mapping.

**float gradSmooth=0.0f**

This equate to the 'Bump Smoothing' spinner in the Cellular texmap UI. The derivative of the function (the distance to the closest cell) has a discontinuity as it switches from cell to cell. Thus the gradients above have a discontinuity. This value basically smoothes off the discontinuity. The range is 0.0 to 1.0.

**Prototype:**

**float RandFromCellID(int id);**

**Remarks:**

Returns a random number in the range 0.0 to 1.0 based on the **cellID** passed.

**Parameters:**

**int id**

The seed for the random number generator.

**Prototype:**

**void setdebug(int i);**

**Remarks:**

This function is used internally.

## List of Texture Symmetry Flags

See Also: [Class StdUVGen](#), [Class TexHandleMaker](#).

One or more of the following flag bit values. These may be ORed together.

### **U\_WRAP**

If set this indicates the texture map is tiled in the U direction.

### **V\_WRAP**

If set this indicates the texture map is tiled in the V direction.

### **U\_MIRROR**

If set this indicates the texture map is mirrored in the U direction.

### **V\_MIRROR**

If set this indicates the texture map is mirrored in the V direction.

## List of Map Slot Types

See Also: [Class MtlBase](#).

One of the following values:

### **MAPSLOT\_TEXTURE**

Identifies a slot which holds a texture map.

### **MAPSLOT\_ENVIRON**

Identifies a slot which holds an environment map.

### **MAPSLOT\_DISPLACEMENT**

This option is available in release 3.0 and later only.

Identifies a slot which holds a displacement map.

### **MAPSLOT\_BACKGROUND**

This option is available in release 4.0 and later only.

Identifies a slot which holds a background map.

# Class TimeChange

See Also: [Class TimeChangeCallback](#), [Class ImageFilter](#).

```
class TimeChange : public TimeChangeCallback
```

## Description:

This class is available in release 2.0 and later only.

This class provides a callback when the user moves the Video Post time slider.

This happens internally, developers must only respond to the

**FLT\_TIMECHANGED** message. See [List of ImageFilter Related Messages](#).

## Data Members:

public:

**BOOL set;**

Indicate the callback is register with 3ds max.

**ImageFilter \*filter;**

Points to the filter who's notified on the time change.

## Methods:

### Prototype:

**TimeChange();**

### Remarks:

Constructor. The data member **set** is made FALSE.

### Prototype:

**void TimeChanged(TimeValue t);**

### Remarks:

This method is called when the user moves the 3ds max time slider to a new time.

### Parameters:

**TimeValue t**

The current time (position of the time slider).

## Class UndoNotify

See Also: [Class TVNodeNotify](#).

```
class UndoNotify : public TVNodeNotify
```

### Description:

This class is available in release 2.0 and later only.

This class can be used so an **ImageFilter** plug-in can get notified on a change to one of its Track View Nodes.

This class provides an implementation of the **NotifyRefChanged()** method of class **TVNodeNotify**. The constructor of this class stores a window handle. Usually this is the control dialog window handle of the ImageFilter plug-in using this class. Upon receipt of a message via

**TVNodeNotify::NotifyRefChanged()** this implementation sends a **FLT\_UNDO** message to the ImageFilter control dialog window proc and invalidates the window. Most filters will set a flag indicating that an undo has occurred when they get the **FLT\_UNDO** message, and actually update the UI controls when they process the **WM\_PAINT** message. This is because the **FLT\_UNDO** message may be sent many time and the controls shouldn't be updated each time (as they might appear to 'flicker'). See the code for the Negative filter in `\MAXSDK\SAMPLES\FILTERS\NEGATIVE\NEGATIVE.CPP` for details.

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
UndoNotify(HWND hwnd);
```

#### Remarks:

Constructor.

#### Parameters:

**HWND hwnd**

The control dialog window handle where the **FLT\_UNDO** message will be sent. This window handle is also passed to **InvalidateRectangle()** so a

**WM\_PAINT** message will be sent.

## List of Image Filter Capability Flags

One or more of the following values. These flags may be ORed together.

### **IMGFLT\_NONE**

Indicates the plug-in has none of the capabilities below. Use this constant for the capability if this is the case.

### **IMGFLT\_MASK**

Indicates the plug-in supports masking.

### **IMGFLT\_CONTROL**

Indicates the plug-In has a control panel. This informs the system to call the plug-ins **ShowControl()** method when the user selects the Setup button. If the filter does not have a control panel do not set this bit and the setup button will be grayed out in the 3ds max user interface.

### **IMGFLT\_FILTER**

Indicates the plug-in is a filter.

### **IMGFLT\_COMPOSITOR**

Indicates the plug-In is a compositor. If the plug-in is a layer type of filter, it should set this bit.

### **IMGFLT\_THREADED**

Indicates the plug-in is thread aware. If this flag is NOT set, 3ds max will avoid multithreading this plug-in.

## List of ImageFilter Related Messages

See Also: [Class ImageFilter](#).

The following message are sent to 3ds max from an Image Filter plug-in. This can be done using the **SendMessage()** Win32 function:

```
LRESULT SendMessage(  
    HWND hwnd, // handle of destination window  
    UINT uMsg, // message to send  
    WPARAM wParam, // first message parameter  
    LPARAM lParam // second message parameter  
);
```

### **FLT\_PROGRESS**

Sent by the plug-in to notify 3ds max of its current progress.

**wParam:** Current amount processed

**lParam:** Total amount to process.

### **FLT\_CHECKABORT**

Sent by the plug-in to check for process interruption. The host should return FALSE (by setting \*lParam) if it's OK to continue or TRUE to abort processing.

**wParam:** Pass 0.

**lParam:** Pointer to a BOOL.

### **FLT\_TEXTMSG**

Sent by the plug-in to display a text message to the user.

**wParam:** 0

**lParam:** LPCTSTR

The following message are sent by 3ds max to a Image Filter plug-in or to 3ds max from an Image Filter. These may be processed inside a dialog proc (for example the filter's **Control()** method).

### **Sample Code:**

```
BOOL ImageFilter_Negative::Control(HWND hWnd,UINT  
message,WPARAM wParam,LPARAM lParam) {  
    switch (message) {  
        case FLT_UNDO:
```

```
undo = true;  
break;  
case FLT_TIMECHANGED:  
// ...
```

## **FLT\_TIMECHANGED**

Sent by 3ds max to the plug-in to notify the time has changed (the user moved the time slider in 3ds max).

**wParam:** 0

**lParam:** TimeValue t

## **FLT\_UNDO**

Sent by 3ds max to the plug-in to notify that an Undo operation has been done. The plugin will set some boolean internally and wait for the next WM\_PAINT message in order to update any spinners or other values that may have been undone. The filter manager sends this message (if you register for the notification with RegisterTVNodeNotify()) and an undo operation was performed.

**wParam:** 0

**lParam:** 0

## Class GBufWriter

See Also: [Class GBuffer](#), [Class GBufReader](#), [Structure GBufData](#), [List of GBuffer Channels Indexes](#).

```
class GBufWriter : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This is the writer object returned from **GBuffer::CreateWriter()**. This class assumes pixels are created in increasing order of x.

Here is an example of writing multiple layer data to the G-Buffer using methods of this class.

```
GBuffer *gb = NewDefaultGBuffer();  
gb->SetRasterSize(100,10);  
gb->CreateChannels((1<<GB_Z)|(1<<GB_MTL_ID)|(1<<GB_BG));  
gb->InitBuffer();
```

```
GBufWriter *wrt = gb->CreateWriter();  
for (int y=0; y<10; y++) {  
  wrt->StartLine(y);  
  BOOL bb = 0;  
  for (int x=5; x<100; x+=4) {  
    wrt->StartPixel(x);  
    wrt->StartNextLayer();  
    float z = 5.0f*float(x)*float(y);  
    wrt->WriteChannelData(GB_Z,(void *)&z);  
    UBYTE mid = 36+x;  
    wrt->WriteChannelData(GB_MTL_ID,(void *)&mid);  
    Color24 c;  
    c.r = 10+x; c.g = 20+x; c.b = 30+x;  
    wrt->WriteChannelData(GB_BG,(void *)&c);  
  
    wrt->StartNextLayer();  
    z = 15.0f*float(x)*float(y);  
    wrt->WriteChannelData(GB_Z,(void *)&z);
```

```

mid = 26+x;
wrt->WriteChannelData(GB_MTL_ID,(void *)&mid);
c.r = 30+x; c.g = 20+x; c.b = 10+x;
wrt->WriteChannelData(GB_BG,(void *)&c);

if (bb) {
wrt->StartNextLayer();
z = 17.0f*float(x)*float(y);
wrt->WriteChannelData(GB_Z,(void *)&z);
mid = 64+x;
wrt->WriteChannelData(GB_MTL_ID,(void *)&mid);
c.r = 130+x; c.g = 120+x; c.b = 110+x;
wrt->WriteChannelData(GB_BG,(void *)&c);
}
bb = !bb;
}
wrt->EndLine();
}
gb->DestroyWriter(wrt);

```

All methods of this class are implemented by the System.

### Methods:

public:

### Prototype:

**virtual void StartLine(int y)=0;**

### Remarks:

This method should be called before writing every line.

### Parameters:

**int y**

The zero based index of the scan line to start.

### Prototype:

**virtual void StartPixel(int x)=0;**

**Remarks:**

This method must be called before writing each pixel and must be called with increasing x values on a line.

**Parameters:**

**int x**

The zero based index of the pixel to start.

**Prototype:**

**virtual void StartNextLayer()=0;**

**Remarks:**

This method must be called before writing the first layer.

**Prototype:**

**virtual BOOL WriteChannelData(int chan, void \*data)=0;**

**Remarks:**

Call this method to write a data element to the specified channel of the G-Buffer to the current scan line and pixel.

**Parameters:**

**int chan**

See [List of GBuffer Channels Indexes](#).

**void \*data**

Points to the G-Buffer data to write.

**Return Value:**

TRUE on success; FALSE on failure.

**Prototype:**

**virtual BOOL WriteAllData(GBufData \*data)=0;**

**Remarks:**

This method writes all the channel data from the GBufData structure passed to the current scan line and pixel.

**Parameters:**

**GBufData \*data**

Points to the G-Buffer data to write. See [Structure GBufData](#).

**Return Value:**

TRUE on success; FALSE on failure.

**Prototype:**

**virtual BOOL EndLine()=0;**

**Remarks:**

This method should be called after writing each line.

**Return Value:**

TRUE on success; FALSE on failure.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Deletes this writer object. Call this method when finished using it.

## List of G-Buffer Channel Types

See Also: [List of Image Channels](#), [List of G-Buffer Channel Indexes](#).

These are the recognized types of G-Buffer channels. The types are defined by the number of bits per pixel for the channel. One of the following values:

**BMM\_CHAN\_TYPE\_8**

1 byte per pixel

**BMM\_CHAN\_TYPE\_16**

1 word per pixel (2 bytes)

**BMM\_CHAN\_TYPE\_24**

3 bytes per pixel

**BMM\_CHAN\_TYPE\_32**

2 words per pixel (4 bytes)

**BMM\_CHAN\_TYPE\_48**

3 words per pixel (6 bytes)

**BMM\_CHAN\_TYPE\_64**

4 words per pixel (8 bytes)

**BMM\_CHAN\_TYPE\_96**

6 words per pixel (12 bytes)

**BMM\_CHAN\_TYPE\_UNKNOWN**

Channel is not of a known type.

# Class RenderInfo

See Also: [Class Ray](#), [Class Point2](#), [Class Point3](#).

class RenderInfo

## Description:

This class provides information about the rendering environment. All methods of this class are implemented by the system.

## Data Members:

public:

**ProjectionType projType;**

The projection type: One of the following values:

**ProjPerspective**

**ProjParallel**

**float kx,ky;**

3D to 2D projection scale factor.

**float xc,yc;**

The screen origin.

**BOOL fieldRender;**

Indicates if the image is field rendered.

**BOOL fieldOdd;**

If TRUE, the first field is Odd lines.

**TimeValue renderTime[2];**

Render time for the 2 fields, if field rendering. If not, use **renderTime[0]**.

**Matrix3 worldToCam[2];**

The world to camera transformation matrix; **worldToCam[0]** is for field 0, **worldToCam[1]** is for field 1. Use **worldToCam[0]** if not field rendering.

**Matrix3 camToWorld[2];**

The camera to world transformation matrix; **camToWorld[0]** is for field 0, **camToWorld[1]** is for field 1. Use **camToWorld[0]** if not field rendering.

**Rect region;**

This data member is available in release 4.0 and later only.

This rectangle holds the sub-region in the image that was rendered if the last

render was a region render. If it was not a region render then the rectangle is empty.

**Methods:**

**Prototype:**

**RenderInfo();**

**Remarks:**

Constructor. The data members are initialized as follows:

```
renderTime[0] = renderTime[1] = 0;  
worldToCam[0].IdentityMatrix();  
worldToCam[1].IdentityMatrix();  
camToWorld[0].IdentityMatrix();  
camToWorld[1].IdentityMatrix();  
fieldRender = fieldOdd = FALSE;  
projType = ProjPerspective;  
kx = ky = 1.0f;  
xc = yc = 400.0f;
```

**Prototype:**

**Point2 MapWorldToScreen(Point3 p, int field=0)**

**Remarks:**

Maps the specified world point to a screen point.

**Parameters:**

**Point3 p**

The world point.

**int field=0**

The field order. This specifies which **camToWorld** matrix is used in the conversion.

**Return Value:**

The 2D screen point.

**Prototype:**

**Point2 MapCamToScreen(Point3 p);**

**Remarks:**

Maps the specified point in camera space to screen space and returns it.

**Parameters:**

**Point3 p**

The point to convert.

**Prototype:**

**Ray MapScreenToCamRay(Point2 p);**

**Remarks:**

Returns the viewing ray through the screen point, in camera space.

**Parameters:**

**Point2 p**

The screen point.

**Prototype:**

**Ray MapScreenToWorldRay(Point2 p, int field=0);**

**Remarks:**

Returns the viewing ray through the specified screen point, in world space.

**Parameters:**

**Point2 p**

The screen space point.

**int field=0**

The field order - specifies which **camToWorld** matrix is used in the conversion.

## List of G-Buffer Channels Indexes

See Also: [List of Image Channels](#), [List of G-Buffer Channel Types](#).

These #defines are indexes to the GBuffer methods such as **GBDataSize(int i)**, **GBChannelName(int i)**, etc.

One of the following values:

### **GB\_Z**

Z-Buffer depth

### **GB\_MTL\_ID**

Material ID assigned via the Material Editor.

### **GB\_NODE\_ID**

Node ID assigned via the Properties dialog.

### **GB\_UV**

UV coordinates.

### **GB\_NORMAL**

Normal vector in view space.

### **GB\_REALPIX**

Non clamped colors in "RealPixel" format.

### **GB\_COVERAGE**

Pixel coverage of the front surface.

### **GB\_BG**

The RGB color of what's behind the front object.

### **GB\_NODE\_RENDER\_ID**

System node number (valid during a render).

### **GB\_COLOR**

The color returned by the material shader for the fragment.

### **GB\_TRANSP**

The transparency returned by the material shader for the fragment.

### **GB\_VELOC**

Velocity vector of the fragment relative to the screen.

### **GB\_WEIGHT**

Weight of layers contribution to pixel color.

### **GB\_MASK**

Sub pixel coverage mask

# Class CheckAbortCallback

See Also: [Class Effect](#).

class CheckAbortCallback

## Description:

This class is available in release 3.0 and later only.

Implemented by the System (for Render Effect plug-ins inside the **Effect::Apply()** method).

The **Check()** method of this class may be called to check if the user did something to abort the application of the effect.

## Methods:

### Prototype:

**virtual BOOL Check()=0;**

### Remarks:

Returns TRUE if user has done something to cause an abort; otherwise FALSE.

### Prototype:

**virtual int Progress(int done, int total)=0;**

### Remarks:

This method is should be called by each Effect plug-in as it proceeds through the image to update the progress bar.

### Parameters:

#### **int done**

The amount done, i.e. the current state of the image processing. This is usually the number of scan lines processed so far.

#### **int total**

The total number of updates. This is usually the number of pixels in height of the image.

### Return Value:

Returns TRUE if user has done something to cause an abort; otherwise FALSE.

**Prototype:**

**virtual void SetTitle(const TCHAR \*title)=0;**

**Remarks:**

This method is called internally by the calling code -- plug-ins don't need to call this method.

## Bitmap Open Mode Types

The following values indicate the open mode for an image:

**BMM\_NOT\_OPEN**

Image not opened yet.

**BMM\_OPEN\_R**

Image opened in Read-only mode.

**BMM\_OPEN\_W**

Image opened in Write-only mode. No reads will occur.

## BitmapIO Capability Flags

These flags describe the capabilities of the IO module. These should be OR'ed together as required to define the properties of the IO module. For example:

```
return BMMIO_WRITER | BMMIO_RANDOM_WRITES |  
BMMIO_EXTENSION |  
BMMIO_INFODLG | BMMIO_CONTROLWRITE;
```

### **BMMIO\_NONE**

Not defined yet.

### **BMMIO\_READER**

Reads files.

### **BMMIO\_WRITER**

Writes files.

### **BMMIO\_RANDOM\_WRITES**

Can write frames in any order.

### **BMMIO\_MULTIFRAME**

File contains multiple frames (i.e. FLC, AVI, ...)

### **BMMIO\_EXTENSION**

Uses file extension (File Filter Type), i.e. **.EPS**, **.PS**. Plug-In's returning this will be expected to implement **ExtCount()**, and **Ext(int i)**. This is also used to identify a plug-in as a "File Type" as opposed to a "Device Type".

### **BMMIO\_FRAMEBUFFER**

Frame Buffer Driver.

### **BMMIO\_GRABBER**

Device Grabs Video.

### **BMMIO\_THREADED**

This is not used. All plug-ins are expected to be thread aware.

### **BMMIO\_RANDOM\_ACCESS**

This is not used.

### **BMMIO\_NON\_CONCURRENT\_ACCESS**

Device cannot handle multiple, concurrent requests (FLC, AVI, VTR's, etc.).

### **BMMIO\_OWN\_VIEWER**

If this flag is set, 3ds max will call the driver's own **ShowImage()** method to

handle displaying the image as opposed to using the generic Virtual Frame Buffer

### **BMMIO\_INFODLG**

If the device is able to show its own image information dialog this flag should be set. Otherwise, the 3ds max must call

**BitmapManager::GetImageInfo()** and display a generic information dialog.

### **BMMIO\_UNINTERRUPTIBLE**

If a driver cannot be started and stopped this flag should be set. This is an 'Uninterruptible Driver'. For example the AVI, FLIC, etc. cannot stop and go and thus define this flag.

### **BMMIO\_EVALMATCH**

Drivers that may have a different image for the same given 3ds max frame and same file/device name should set this flag and implement the

**BitmapIO::EvalMatch()** method.

### **BMMIO\_IFL**

If this flag is set, instead of calling the **Load()** method 3ds max will call the **GetImageName()** method. 3ds max will then process the given image name accordingly.

The following flags relate to the plug-in's control dialog. There is only a single call to the plug-in's control panel but the call specifies the nature of the operation going on. It's up to the plug-in to provide different interfaces if needed. If one control serves two or more services, all the pertinent flags should be set.

#### **BMMIO\_CONTROLREAD**

Device Driver has a Control Panel for Read Operations

#### **BMMIO\_CONTROLWRITE**

Device Driver has a Control Panel for Write Operations

#### **BMMIO\_CONTROLGENERIC**

This flag is no longer used.

## Bitmap Error Codes

The following error codes are used by the functions that use or return

**BMMRES** types:

### **BMMRES\_SUCCESS**

Success - No error occurred.

### **BMMRES\_ERRORTAKENCARE**

Error - The function has already taken appropriate action to process the error.

**The errors below require the user to be informed.**

**BMMRES\_FILENOTFOUND**

The file being accessed was not found.

**BMMRES\_MEMORYERROR**

Insufficient memory for the requested operation.

**BMMRES\_NODRIVER**

Device driver responsible for the image not present.

**BMMRES\_IOERROR**

Input / Output error.

**BMMRES\_INVALIDFORMAT**

The file being access was not of the proper format for the requested operation.

**BMMRES\_CORRUPTFILE**

The file being accessed was corrupt.

**BMMRES\_SINGLEFRAME**

Results from a goto request on a single frame image.

**BMMRES\_INVALIDUSAGE**

Bad argument passed to function (developer mistake).

**BMMRES\_ERRORRETRY**

This is no longer used. Use **BMMRES\_RETRY** below.

**BMMRES\_NUMBEREDFILENAMEERROR**

This may be passed as an errorcode to  
**BitmapIO::ProcessImageIOError()**.

**BMMRES\_INTERNALERROR**

An internal error occurred.

**BMMRES\_BADFILEHEADER**

A file header error occurred.

**BMMRES\_CANTSTORAGE**

This is used internally.

**BMMRES\_RETRY**

This is returned if the user selects Retry from the 3ds max Image IO Error dialog box. This dialog is presented by the method  
**BitmapIO::ProcessImageIOError()**.

**BMMRES\_BADFRAME**  
Invalid Frame Number Requested

## List of Bitmap Close Types

One of the following values:

### **BMM\_CLOSE\_COMPLETE**

Close and save the image.

### **BMM\_CLOSE\_ABANDON**

Close but abandon the image.

Many bitmap loader / savers do not make use of these flags. An example of one that does is the FLC saver. When a FLC file is closed, it goes through each image and calculates the palette. This can take a long time. If the user cancels the operation they will not want to wait for this to happen. Thus this flag is passed to indicate that the user has canceled and this processing should not occur.

## Class IUtil

See Also: [Class UtilityObj](#).

class IUtil

### **Description:**

This class provides a single method to allow the developer to close their utility plug-in in the command panel.

### **Methods:**

### **Prototype:**

**virtual void CloseUtility()=0;**

### **Remarks:**

Implemented by the System.

A developer may call this method to close the current utility in the command panel. If a plug-in developer chooses to use a Close button in one of their rollup pages this method should be called when the user has selected the Close button. This method simply tells the system that the plug-in should be closed. A developer does not need to use this method. It is just available if a developer wants to have a specific termination point to their utility plug-in.

## Class ITreeEnumProc

See Also: [Class IScene](#), [Class INode](#).

class ITreeEnumProc

### Description:

This is the callback object used by **IScene::EnumTree()**. To use it, derive a class from this class, and implement the callback method.

### Methods:

### Prototype:

```
virtual int callback( INode *node )=0;
```

### Remarks:

Implemented by the Plug-In.

This method may flag the node passed based on some property of the node.

### Parameters:

**INode \*node**

The node. The **INode** class has a method **FlagForeground()** that may be used to flag this node to go into the foreground.

### Return Value:

One of the following values may be returned to control how enumeration continues:

**TREE\_CONTINUE**

Continue enumerating.

**TREE\_IGNORECHILDREN**

Don't enumerate the children of this node, but continue enumerating.

**TREE\_ABORT**

Stop enumerating.

## Class ITVUtility

See Also: [Class Animatable](#), [Class TrackViewUtility](#), [Class Interval](#), [Class Interface](#), [Class Control](#).

class ITVUtility

### Description:

This class is an interface that is given to track view utilities that allows them to access the track view they were launched from. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual int GetNumTracks()=0;**

#### Remarks:

Returns the total number of visible (open) tracks in Track View. This determines valid values to use for 'i' in the methods below.

#### Prototype:

**virtual Animatable \*GetAnim(int i)=0;**

#### Remarks:

Returns a pointer to the Animatable for the 'i-th' track.

#### Parameters:

**int i**

Specifies the track.

#### Prototype:

**virtual Animatable \*GetClient(int i)=0;**

#### Remarks:

Returns a pointer to the client of the 'i-th' track. This is the 'parent' or 'owner' of the 'i-th' item.

#### Parameters:

**int i**

Specifies the track.

**Prototype:**

**virtual int GetSubNum(int i)=0;**

**Remarks:**

Returns the sub-anim number of the 'i-th' track.

**Parameters:**

**int i**

Specifies the track.

**Prototype:**

**virtual TSTR GetTrackName(int i)=0;**

**Remarks:**

Returns the name of the 'i-th' track.

**Parameters:**

**int i**

The index of the track whose name is returned.

**Prototype:**

**virtual BOOL IsSelected(int i)=0;**

**Remarks:**

Returns TRUE if the 'i-th' track is selected; otherwise FALSE.

**Parameters:**

**int i**

The index of the track to test.

**Prototype:**

**virtual void SetSelect(int i,BOOL sel)=0;**

**Remarks:**

Sets the selected state of the 'i-th' track to the state passed.

**Parameters:**

**int i**

The index of the track whose selected state is set.

**BOOL sel**

Specifies the selected state. TRUE for selected; FALSE for not selected.

**Prototype:**

**virtual HWND GetTrackViewHWND()=0;**

**Remarks:**

Returns the windows handle of the main Track View window.

**Prototype:**

**virtual int GetMajorMode()=0;**

**Remarks:**

Returns a value to indicate the current mode Track View is operating in. This is one of five modes.

**Return Value:**

One of the following values:

**TVMODE\_EDITKEYS**

**TVMODE\_EDITTIME**

**TVMODE\_EDITRANGES**

**TVMODE\_POSRANGES**

**TVMODE\_EDITFCURVE**

**Prototype:**

**virtual Interval GetTimeSelection()=0;**

**Remarks:**

Returns the current interval of selected time.

**Prototype:**

**virtual BOOL SubTreeMode()=0;**

**Remarks:**

Returns TRUE if 'Modify Subtree' mode is active; otherwise FALSE.

**Prototype:**

**virtual Animatable \*GetTVRoot()=0;**

**Remarks:**

Returns a pointer to the Track View Root.

**Prototype:**

**virtual void TVUtilClosing(TrackViewUtility \*util)=0;**

**Remarks:**

This must be called when a track view utility is closing so that it can be unregistered from notifications

**Parameters:**

**TrackViewUtility \*util**

Points to the utility that is closing. This is usually called by passing the **this** pointer as in: **TVUtilClosing(this);**

## Class RendParamDlg

See Also: [Class Renderer](#).

class RendParamDlg

### Description:

An instance of this class is created by **Renderer::CreateParamDlg()**. Since this dialog is modeless and non-interactive, as the user changes parameters in the dialog, the renderer does not need to update its state. When the user is through, they may choose 'OK' or 'Cancel' from the dialog. If the user selects OK then the **AcceptParams()** method will be called. If the user selects Cancel, then the **RejectParams()** method is called.

### Methods:

#### Prototype:

**virtual void AcceptParams()=0;**

#### Remarks:

Implemented by the Plug-In.

If the user selects OK from the dialog, this method will be called, at which time the renderer can read the parameters out of the UI and modify its state.

#### Prototype:

**virtual void RejectParams();**

#### Remarks:

Implemented by the Plug-In.

If this method is called, typically the renderer will not have to do anything since it has not yet modified its state, but if for some reason it has, it should restore its state.

#### Default Implementation:

**{}**

#### Prototype:

**virtual void DeleteThis()=0;**

**Remarks:**

Implemented by the Plug-In.

Deletes this instance of the class. The instance is allocated by

**Renderer::CreateParamDialog()**.

## Class BaseShader

See Also: [Class SpecialFX](#), [Class ShaderParamDlg](#), [Class ShadeContext](#), [Class IllumParams](#), [Class IMtlParams](#), [Class StdMat2](#), [Class Mtl](#), [Class Color](#), [Class ILoad](#), [Class ISave](#).

```
class BaseShader : public SpecialFX
```

### Description:

This class is available in release 3.0 and later only.

This is one of the base classes for the creation of Shaders which plug-in to the Standard material. Note: Developers should derive their plug-in Shader from [Class Shader](#) rather than this class directly since otherwise the interactive renderer won't know how to render the Shader in the viewports.

Developers of this plug-in type need to understand how the Standard material and the Shader work together.

Every material has a Shader. The Shader is the piece of code which controls how light is reflected off the surface. The Standard material is basically the mapping mechanism. It handles all the texturing for the material. It also manages the user interface. This simplifies things so the Shader plug-in only needs to worry about the interaction of light on the surface.

Prior to release 3 developers could write Material plug-ins that performed their own shading, however this was usually a major programming task. Release 3 provides the simpler Shader plug-in that would benefit from sharing all the common capabilities. The Standard material, with its 'blind' texturing mechanism, makes this possible. It doesn't know what it is texturing -- it simply textures 'stuff'. The shader names the channels (map), fills in the initial values, specifies if they are a single channel (mono) or a triple channel (color). The Standard material handles the rest including managing the user interface.

Most of the code in a Shader has to do with supplying this information to a Standard material. The values are passed and received back in class

**IllumParams**. There is a single method in a shader which actually does the shading. This is the **Illum()** method.

### Plug-In Information:

Class Defined In SHADER.H

Super Class ID SHADER\_CLASS\_ID

Standard File Name Extension DLB

Extra Include File Needed SHADERS.H

### **Methods Groups:**

The hyperlinks below take you to the start of groups of related methods within the class:

[Mapping Channel Access](#)

[Dialog \(Rollup\) Access](#)

[Illumination Related Methods](#)

[Standard Parameter Access](#)

[Compositing Method](#)

[Load / Save Methods](#)

### **Methods:**

public:

### **Dialog (Rollup) Access**

#### **Prototype:**

```
virtual ShaderParamDlg* CreateParamDialog(HWND  
hOldRollup, HWND hwMtlEdit, IMtlParams *imp, StdMtl2*  
theMtl, int rollupOpen, int n=0)=0;
```

#### **Remarks:**

This method creates and returns a pointer to a **ShaderParamDlg** object and puts up the dialog which lets the user edit the Shader's parameters.

#### **Parameters:**

##### **HWND hOldRollup**

The window handle of the old rollup. If non-NULL the IMtlParams method ReplaceRollup method is usually used instead of AddRollup() to present the rollup.

##### **HWND hwMtlEdit**

The window handle of the material editor.

## **IMtlParams \*imp**

The interface pointer for calling methods in 3ds max.

## **StdMtl2\* theMtl**

Points to the Standard material being edited.

## **int rollupOpen**

TRUE to have the UI rollup open; FALSE if closed.

## **int n=0**

This parameter is available in release 4.0 and later only.

Specifies the number of the rollup to create. Reserved for future use with multiple rollups.

### **Prototype:**

```
virtual ShaderParamDlg* GetParamDlg(int n=0)=0;
```

### **Remarks:**

Returns a pointer to the **ShaderParamDlg** object which manages the user interface.

### **Parameters:**

#### **int n=0**

This parameter is available in release 4.0 and later only.

Specifies the rollup to get **ShaderParamDlg** for. Reserved for future use with multiple rollups.

### **Prototype:**

```
virtual void SetParamDlg(ShaderParamDlg* newDlg, int n=0)=0;
```

### **Remarks:**

Sets the **ShaderParamDlg** object which manages the user interface to the one passed.

### **Parameters:**

#### **ShaderParamDlg\* newDlg**

Points to the new ShaderParamDlg object.

#### **int n=0**

This parameter is available in release 4.0 and later only.

Specifies the rollup to set **ShaderParamDlg** for. Reserved for future use with multiple rollups.

## Standard Parameter Access

### Prototype:

**virtual ULONG SupportStdParams()=0;**

### Remarks:

Returns a value which indicates which of the standard parameters are supported.

### Return Value:

See [List of Shader Standard Parameter Flags](#).

### Prototype:

**virtual void ConvertParamBlk(ParamBlockDescID \*descOld, int oldCount, IParamBlock \*oldPB)=0;**

### Remarks:

This method is only required for R2.5 shaders to convert the previous Standard material parameter blocks to the current version.

### Parameters:

**ParamBlockDescID \*descOld**

Points to the old parameter block descriptor.

**int oldCount**

The number in the array of parameters above.

**IParamBlock \*oldPB**

Points to the old parameter block.

### Prototype:

**virtual int NParamDlgs();**

### Remarks:

This method is available in release 4.0 and later only.

Returns the number of rollups this shader is requesting.

### Default Implementation:

```
{ return 1; }
```

## Illumination Related Methods

### Prototype:

```
virtual void Illum(ShadeContext &sc, IllumParams &ip)=0;
```

### Remarks:

This is the illumination function for the Shader.

Developers will find it very helpful to review the **Mtl::Shade()** method of the Standard material. This is the main method of the material which computes the color of the point being rendered. This code is available in **\MAXSDK\SAMPLES\MATERIALS\STDMTL2.CPP**. This code shows how the Standard calls **Shader::GetIllumParams()**, sets up mapping channels, calls this **Illum()** method, and calls the **Shader::CombineComponents()** method when all done.

### Parameters:

#### **ShadeContext &sc**

The ShadeContext which provides information on the pixel being shaded.

#### **IllumParams &ip**

The object whose data members provide communication between 3ds max and the shader. Input values are read from here and output values are stored here.

Note that the **XOut (ambIllumout, etc)** data members of this class are initialized to 0 before the first call to this method.

The input to this method has the textured illumination parameters, the bump perturbed normal, the view vector, the raw (unattenuated) colors in the reflection and refraction directions, etc.

### Sample Code:

Below is a brief analysis of the standard Blinn shader Illum() method. This is the standard 'computer graphics look' type shader supplied with 3ds max. The entire method follows:

```
void Blinn2::Illum(ShadeContext &sc, IllumParams &ip) {  
    LightDesc *l;
```

**Color lightCol;**

**// Blinn style phong**

**BOOL is\_shiny= (ip.channels[ID\_SS].r > 0.0f) ? 1:0;**

**double phExp = pow(2.0, ip.channels[ID\_SH].r \* 10.0) \* 4.0;**

**for (int i=0; i<sc.nLights; i++) {**

**l = sc.Light(i);**

**register float NL, diffCoef;**

**Point3 L;**

**if (l->Illuminate(sc,ip.N,lightCol,L,NL,diffCoef)) {**

**if (l->ambientOnly) {**

**ip.ambIllumOut += lightCol;**

**continue;**

**}**

**if (NL<=0.0f)**

**continue;**

**// diffuse**

**if (l->affectDiffuse)**

**ip.diffIllumOut += diffCoef \* lightCol;**

**// specular (Phong2)**

**if (is\_shiny&&l->affectSpecular) {**

**Point3 H = Normalize(L-ip.V);**

**float c = DotProd(ip.N,H);**

**if (c>0.0f) {**

**if (softThresh!=0.0 && diffCoef<softThresh) {**

**c \*= Soften(diffCoef/softThresh);**

**}**

**c = (float)pow((double)c, phExp);**

**ip.specIllumOut += c \* ip.channels[ID\_SS].r \* lightCol;**

**}**

**}**

**}**

**}**

```

// Apply mono self illumination
if ( ! selfIllumClrOn ){
    float si = ip.channels[ID_SI].r;
    ip.diffIllumOut = (si>=1.0f) ? Color(1.0f,1.0f,1.0f) :
        ip.diffIllumOut * (1.0f-si) + si;
} else {
    // colored self illum,
    ip.selfIllumOut += ip.channels[ID_SI];
}
// now we can multiply by the clr,
ip.ambIllumOut *= ip.channels[ID_AM];
ip.diffIllumOut *= ip.channels[ID_DI];
ip.specIllumOut *= ip.channels[ID_SP];

// the following is applicable only in R4
    int chan = ip.stdIDToChannel[ ID_RL ];
    ShadeTransmission(sc, ip, ip.channels[chan], ip.refractAmt);
    chan = ip.stdIDToChannel[ ID_RR ];
    ShadeReflection( sc, ip, ip.channels[chan] );
    CombineComponents( sc, ip );
}

```

Some of the key parts of this method are discussed below:

The **is\_shiny** line sets a boolean based on if the Shader has a shininess setting > 0. Note that the Blinn shader uses the same channel ordering as the original Standard material so it can index its channels using the standard ID **ID\_SS**.

```
BOOL is_shiny= (ip.channels[ID_SS].r > 0.0f) ? 1:0;
```

Next the 'Phong Exponent' is computed. This is just a function that is used to give a certain look. It uses  $2^{(\text{Shininess} * 10)} * 4$ . This calculation simply 'feels right' and gives a good look.

```
double phExp = pow(2.0, ip.channels[ID_SH].r * 10.0) * 4.0;
```

The following loop sums up the effect of each light on this point on surface.

```
for (int i=0; i<sc.nLights; i++) {
```

Inside the loop, the light descriptor is grabbed from the ShadeContext:

```
l = sc.Light(i);
```

The **LightDesc::Illuminate()** method is then called to compute some data:

```
if (l->Illuminate(sc,ip.N,lightCol,L,NL,diffCoef) {
```

To **Illuminate()** is passed the ShadeContext (**sc**), and the normal to the surface (**ip.N**) (pointing away from the surface point).

The method returns the light color (**lightCol**), light vector (**L**) (which points from the surface point to the light), the dot product of N and L (**NL**) and the diffuse coefficient (**diffCoef**). The diffuse coefficient is similar to NL but has the atmosphere between the light and surface point taken into account as well.

The next piece of code checks if the light figures into the computations:

```
if (NL<=0.0f)
    continue;
```

If  $NL < 0$  then the cosine of the vectors is greater than 90 degrees. This means the light is looking at the back of the surface and is therefore not to be considered.

The next statement checks if the light affect the diffuse channel (lights may toggle on or off their ability to affect the diffuse and specular channels.)

```
if (l->affectDiffuse)
    ip.diffIllumOut += diffCoef * lightCol;
```

If the light affects the diffuse channel then the diffuse illumination output component of the **IllumParams** is added to. This is done by multiplying the diffuse coefficient (returned by **LightDesc::Illuminate()**) times the light color (also returned by **LightDesc::Illuminate()**). Notice that the **diffIllumOut** is being accumulated with each pass of the light loop.

The next section of code involves the specular component. If the light is shiny, and it affects the specular channel a vector **H** is computed.

```
if (is_shiny && l->affectSpecular) {
```

Note the following about this **H** vector computation. Most vectors are considered pointing **from** the point on the surface. The view vector (**IllumParams::V**) does not follow this convention. It points from the 'eye' towards the surface. Thus it's reversed from the usual convention.

**H** is computed to be the average of **L** and **V**. This is  $(\mathbf{L} + \mathbf{V})/2$ . Since we normalize this we don't have to divide by the 2. So, if **V** followed the standard convention this would be simply  $\mathbf{L} + \mathbf{V}$ . Since it doesn't it is  $\mathbf{L} + (-\mathbf{ip.V})$  or  $\mathbf{L} - \mathbf{ip.V}$ .

```
Point3 H = Normalize(L-ip.V);
```

Next the dot product of **N** and **H** is computed and stored in **c**. When you take the dot product of two normalized vectors what is returned is the cosine of the angle between the vectors.

```
float c = DotProd(ip.N,H);
```

If  $c > 0$  and the diffuse coefficient is less than the soften threshold then **c** is modified by a 'softening' curve.

```
if (c>0.0f) {  
  if (softThresh!=0.0 && diffCoef<softThresh) {  
    c *= Soften(diffCoef/softThresh);  
  }  
}
```

Note that the **Soften()** function is defined in `\MAXSDK\SAMPLES\MATERIALS\SHADERUTIL.CPP` and developers can copy this code.

```
c = (float)pow((double)c, phExp);
```

Next, **c** is raised to the power of the Phong exponent. This is effectively taking a cosine (a smooth S curve) and raising it to a power. As it is raised to a power it becomes a sharper and sharper S curve. This is where the shape of the highlight curve in the Materials Editor UI comes from.

That completes the pre computations for the specular function. Then **c** is multiplied by the specular strength (**ip.channels[ID\_SS].r**) times the light color (**lightCol**). The result is summed in specular illumination out (**ip.specIllumOut**).

```
ip.specIllumOut += c * ip.channels[ID_SS].r * lightCol;  
}  
}  
}
```

That completes the light loop. It happens over and over for each light.

Next the self illumination is computed. If the Self Illumination Color is not on, then the original code for doing mono self illumination is used.

```
// Apply mono self illumination  
if ( ! selfIllumClrOn ){  
  float si = ip.channels[ID_SI].r;  
  ip.diffIllumOut = (si>=1.0f) ? Color(1.0f,1.0f,1.0f) :  
    ip.diffIllumOut * (1.0f-si) + si;  
} else {
```

Otherwise the self illumination color is summed in to the Self Illumination Out (**ip.selfIllumOut**).

```
// colored self illum,  
ip.selfIllumOut += ip.channels[ID_SI];  
}
```

Then, we multiply by the colors for ambient, diffuse and specular.

```
ip.ambIllumOut *= ip.channels[ID_AM];  
ip.diffIllumOut *= ip.channels[ID_DI];  
ip.specIllumOut *= ip.channels[ID_SP];
```

The results are **ambIllumOut**, **diffIllumOut**, and **specIllumOut**. Note that these components are not summed. In R3 and earlier these results would be returned to the Standard material. However, R4 introduces a couple extra steps. Finally, we call **ShadeTransmission()** and **ShadeReflection()** to apply the transmission/refraction and reflection models. The core implementation of 3ds max provides the standard models, but the shader can override these methods to compute its own models.

```
int chan = ip.stdIDToChannel[ ID_RL ];  
ShadeTransmission(sc, ip, ip.channels[chan], ip.refractAmt);  
chan = ip.stdIDToChannel[ ID_RR ];  
ShadeReflection( sc, ip, ip.channels[chan] );
```

In R4, It is a shader's responsibility to combine the components of the shading process prior to exiting **Illum()** (in R3, this was the responsibility of the Standard material). In order to combine these values together to produce the final color for that point on the surface (**IllumParams.finalC**), the shader should call **CombineComponents()** method. The Shader base class provides a default implementation which simply sums everything together and multiplies by the opacity.

```
virtual void CombineComponents( IllumParams& ip )  
{  
ip.finalC = ip.finalOpac *  
(ip.ambIllumOut + ip.diffIllumOut + ip.selfIllumOut)  
+ ip.specIllumOut + ip.reflIllumOut + ip.transIllumOut ;  
}
```

**Prototype:**

```
virtual void AffectReflection(ShadeContext &sc, IllumParams  
&ip, Color &rcol)=0;
```

**Remarks:**

Note: This method has been superceded by **ShadeReflection()** and is **obsolete** in release 4.0 and later.

This method provides the shader with an opportunity to affect the reflection code.

**Parameters:****ShadeContext &sc**

The ShadeContext which provides information on the pixel being shaded.

**IllumParams &ip**

The object whose data members provide communication between 3ds max and the shader.

**Color &rcol**

The input reflection color is passed in here and the resulting 'affected' color is stored here.

**Sample Code:**

A simple example like Phong does the following:

```
void AffectReflection(ShadeContext &sc, IllumParams &ip, Color  
&rcol)  
{  
    rcol *= ip.channels[ID_SP];  
};
```

If a color can affect the reflection of light off a surface than it can usually affect the reflection of other things off a surface. Thus some shaders influence the reflection color using the specular color and specular level. For instance the Multi Layer Shader does the following:

```
#define DEFAULT_GLOSS2 0.03f
```

```
void MultiLayerShader::AffectReflection(ShadeContext &sc,  
IllumParams &ip, Color &rcol)  
{
```

```

float axy = DEFAULT_GLOSS2;
float norm = 1.0f / (4.0f * PI * axy );
rcol *= ip.channels[_SPECLEV1].r * ip.channels[_SPECCLR1] *
norm;
}

```

**Prototype:**

```
virtual void GetIllumParams(IllumParams* ip)=0;
```

**Remarks:**

This method updates the **channels** (as well as other) data member of the **IllumParams** object passed to it with the **local** variables of the material for possible mapping prior to being given to the Shader's **Illum()** method. The shader plug-in copies the state of all its parameters (at their current animation state) into the data members of the **IllumParams** passed.

**Parameters:**

**IllumParams\* ip**

Points to the IllumParams to update.

**Prototype:**

```
virtual void ShadeReflection(ShadeContext &sc, IllumParams
&ip, Color &mapClr);
```

**Remarks:**

This method is available in release 4.0 and later only.

Compute the reflected color from the **sc**, **ip**, and reflection map (or ray) color. The core implementation of this provides the standard 3ds max reflection model. To support the standard reflection model, a shader may call this default implementation.

**Parameters:**

**ShadeContext& sc**

The context which provides information on the pixel being shaded.

**IllumParams& ip**

The object whose data members provide communication between 3ds

max and the shader.

### **Color &mapClr**

The input reflection (or ray) color is passed in here and the resulting 'affected' color is stored here.

### **Prototype:**

**virtual void ShadeTransmission(ShadeContext &sc, IllumParams &ip, Color &mapClr, float amount);**

### **Remarks:**

This method is available in release 4.0 and later only.

Compute the transmission/refraction color for the sample.. The core implementation of this provides the standard 3ds max reflection model. To support the standard transmission/refraction model, a shader may call this default implementation.

### **Parameters:**

#### **ShadeContext& sc**

The context which provides information on the pixel being shaded.

#### **IllumParams& ip**

The object whose data members provide communication between 3ds max and the shader.

#### **Color &mapClr**

The input refraction (or ray) color is passed in here and the resulting 'affected' color is stored here.

#### **float amount**

The level of the amount spinner for the refraction channel.

## **Mapping Channel Access**

### **Prototype:**

**virtual long nTexChannelsSupported()=0;**

### **Remarks:**

Returns the number of texture map map channels supported by this Shader.

**Prototype:**

```
virtual TSTR GetTexChannelName(long nTextureChan)=0;
```

**Remarks:**

Returns the name of the specified texture map channel.

**Parameters:**

**long nTextureChan**

The zero based index of the texture map channel whose name is returned.

**Prototype:**

```
virtual TSTR GetTexChannelInternalName(long nTextureChan);
```

**Remarks:**

Returns the internal name of the specified texture map. The Standard material uses this to get the fixed, parsable internal name for each texture channel it defines.

**Parameters:**

**long nTextureChan**

The zero based index of the texture map whose name is returned.

**Default Implementation:**

```
{ return GetTexChannelName(nTextureChan); }
```

**Prototype:**

```
virtual long ChannelType(long nTextureChan)=0;
```

**Remarks:**

Returns the channel type for the specified texture map channel. There are four channels which are part of the Material which are not specific to the Shader. All other channels are defined by the Shader (what they are and what they are called.) The four which are not the province of the Shader are Bump, Reflection, Refraction and Displacement. For example, Displacement mapping is really a geometry operation and not a shading one. The channel type returned from this method indicates if the specified channel is one of these, or if it is a monochrome channel, a color channel, or is not a supported channel.

**Parameters:**

## **long nTextureChan**

The zero based index of the texture map whose name is returned.

### **Return Value:**

Texture channel type flags. One or more of the following values:

#### **UNSUPPORTED\_CHANNEL**

Indicates the channel is not supported (is not used).

#### **CLR\_CHANNEL**

A color channel. The **Color.r**, **Color.g** and **Color.b** parameters are used.

#### **MONO\_CHANNEL**

A monochrome channel. Only the **Color.r** is used.

#### **BUMP\_CHANNEL**

The bump mapping channel.

#### **REFL\_CHANNEL**

The reflection channel.

#### **REFR\_CHANNEL**

The refraction channel.

#### **DISP\_CHANNEL**

The displacement channel.

#### **ELIMINATE\_CHANNEL**

Indicates that the channel is not supported. For example, a certain Shader might not support displacement mapping for some reason. If it didn't, it could use this channel type to eliminate the support of displacement mapping for itself. It would be as if displacement mapping was not included in the material. None of the 3ds max shaders use this.

#### **SKIP\_CHANNELS**

This is used internally to indicate that the channels to be skipped.

### **Prototype:**

```
virtual long StdIDToChannel(long stdID)=0;
```

### **Remarks:**

Returns the index of this Shader's channels which corresponds to the specified Standard materials texture map ID. This allows the Shader to arrange its channels in any order it wants in the **IllumParams::channels** array but

enables the Standard material to access specific ones it needs (for instance the Bump channel or Reflection channel).

**Parameters:**

**long stdID**

The ID whose corresponding channel to return. See [List of Texture Map Indices](#).

**Return Value:**

The zero based index of the channel. If there is not a corresponding channel return -1.

**Sample Code:**

This can be handled similar to below where an array is initialized with the values of this plug-in shader's channels that correspond to each of the standard channels. Then this method just returns the correspond index from the array.

```
static int stdIDToChannel[N_ID_CHANNELS] = {  
    0, 1, 2, 5, 4, -1, 7, 8, 9, 10, 11, 12  
};  
  
long StdIDToChannel(long stdID){ return  
stdIDToChannel[stdID]; }
```

**Prototype:**

```
virtual void Reset()=0;
```

**Remarks:**

This method is called when the Shader is first activated in the dropdown list of Shader choices. The Shader should reset itself to its default values.

## **Compositing Method**

**Prototype:**

```
virtual void CombineComponents(ShadeContext &sc,  
IllumParams& ip);
```

**Remarks:**

This method does the final compositing of the various illumination components. A default implementation is provided which simply adds the

components together. Developers who want to do other more specialized composition can override this method. For example, a certain Shader might want to composited highlights over the underlying diffuse component since the light is reflected and the diffuse color wouldn't fully show through. Such a Shader would provide its own version of this method.

**Parameters:**

**ShadeContext &sc**

The ShadeContext which provides information on the pixel being shaded.

**IllumParams& ip**

The illumination parameters to composite and store.

**Default Implementation:**

```
virtual void CombineComponents(IllumParams& ip)
{
    ip.finalC = ip.finalOpac *
        (ip.ambIllumOut + ip.diffIllumOut + ip.selfIllumOut)
        + ip.specIllumOut + ip.reflIllumOut + ip.transIllumOut ;
}
```

**Prototype:**

```
virtual ULONG GetRequirements(int subMtlNum)=0;
```

**Remarks:**

Returns the requirements of the Shader for the specified sub-material. Many objects in the rendering pipeline use the requirements to tell the renderer what data needs to be available. The Shader's requirements are OR'd with the combined map requirements and returned to the renderer via the Stdmtl2's **GetRequirements()** function.

**Parameters:**

**int subMtlNum**

This parameter is not used.

**Return Value:**

One or more of the following flags:

See [List of Material Requirement Flags](#).

## **Load / Save Methods**

### **Prototype:**

**IOResult Save(ISave \*isave)=0;**

### **Remarks:**

Saves the plug-in's name. This should be called at the start of a plug-in's **Save()** method.

### **Parameters:**

**ISave \*isave**

An interface for saving data.

### **Prototype:**

**IOResult Load(ILoad \*iload)=0;**

### **Remarks:**

Loads the plug-in's name. This should be called at the start of a plug-in's **Load()** method.

### **Parameters:**

**ILoad \*iload**

An interface for loading data.

## Class SamplingCallback

See Also: [Class Sampler](#), [Class ShadeContext](#), [Class Color](#), [Class Point2](#).

```
class SamplingCallback : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

This is the callback object for the **DoSamples()** method of class **Sampler**. The **SampleAtOffset()** method is the one that actually computes the shading value for the Sampler.

### Methods:

public:

### Prototype:

```
virtual BOOL SampleAtOffset(Color &col, Color &trans, Point2& sample, float sampleScale)=0;
```

### Remarks:

This is the method that integrates the sampler into the renderer. The plug-in Sampler calls this method to actually perform a sample at the specified 2D point. This method computes the output color and transparency.

### Parameters:

#### **Color &col**

The output color.

#### **Color &trans**

The output transparency.

#### **Point2& sample**

The 2D sample point.

#### **float sampleScale**

The scale of the sample. This parameter is the way a sampler tells the shader to use the whole pixel (sampleScale=1) size for texture samples or some fraction. This scale is an **edge** scale not an **area** scale, so if you want samples 1/4 pixel large the sampleScale should be 1/2.

### Return Value:

TRUE if the sample was processed; FALSE if the clipped sample was ignored.

## Class HSVCallback

See Also: [Class ColorPicker](#), [List of Color Conversion Utilities](#).

class HSVCallback

### Description:

This class provides methods that are called when using the modeless color picker. All methods of this class are implemented by the Plug-In.

### Methods:

### Prototype:

**virtual void ColorChanged(DWORD col, BOOL buttonUp)=0;**

### Remarks:

Implemented by the Plug-In.

This callback proc gets called after the user changes the color. Implement this method to handle interactive updates.

### Parameters:

**DWORD col**

The new color.

**BOOL buttonUp**

Indicates if the mouse button has been released (is up). TRUE if the button is up; FALSE if it is down.

### Prototype:

**virtual void BeingDestroyed(IPoint2 pos)=0;**

### Remarks:

Implemented by the Plug-In.

This callback proc gets called when the color picker is closed:

### Parameters:

**IPoint2 pos**

The last screen position of the color picker before it was closed.

### Prototype:

## **virtual void ButtonDown()**

### **Remarks:**

Implemented by the Plug-In.

This method is called when the user has pressed the mouse button.

### **Default Implementation:**

```
{}
```

### **Prototype:**

**virtual void ButtonUp(BOOL accept)**

### **Remarks:**

Implemented by the Plug-In.

This method is called when the user has released the mouse button.

### **Parameters:**

**BOOL accept**

TRUE if the mouse button was released normally; FALSE if the user canceled.

### **Default Implementation:**

```
{}
```

## Class MCDeviceBinding

See Also: [Class ReferenceTarget](#), [Class MCInputDevice](#), [Class IMCParamDlg](#), [Class IRollupWindow](#).

```
class MCDeviceBinding : public ReferenceTarget
```

### Description:

An instance of this class is created when a motion capture controller binds one of its parameters to a device. The main purpose of this class is to store any parameters that describe the binding.

### Methods:

#### Prototype:

```
virtual MCInputDevice *GetDevice()=0;
```

#### Remarks:

Returns a pointer to the bound input device.

#### Prototype:

```
virtual void DeleteThis()=0;
```

#### Remarks:

Deletes this instance of the class.

#### Prototype:

```
virtual TSTR BindingName()=0;
```

#### Remarks:

Returns the name of the bound input device.

#### Prototype:

```
virtual float Eval(TimeValue t)=0;
```

#### Remarks:

A device binding is the thing that the controller evaluates to get the value of the device. Everything is simply a scalar parameter. So for example even for a device like a mouse that has X and Y motion the device binding will break down into simply X or Y. This method is used to return the value of the device

at the instant this method is called.

**Parameters:**

**TimeValue t**

The time at which this method is called.

**Prototype:**

**virtual void AddRollup(IMCParamDlg \*dlg)=0;**

**Remarks:**

This method is called to allow the binding to put up any user interface it has into the command panel via rollup pages.

**Parameters:**

**IMCParamDlg \*dlg**

The **IRollupWindow** data member of this class may be used to add the rollup page.

**Sample Code:**

```
dlg->iRoll->AppendRollup(  
    hInstance,  
    MAKEINTRESOURCE(IDD_MC_MOUSE),  
    MouseDeviceDlgProc,  
    GetString(IDS_RB_MOUSEDEVICE),  
    (LPARAM)dlg);
```

**Prototype:**

**virtual void UpdateRollup(IRollupWindow \*iRoll)=0;**

**Remarks:**

This method is called to allow the plug-in to update the values in its user interface rollup.

**Parameters:**

**IRollupWindow \*iRoll**

The interface into the command panel rollups. The **GetPanelDlg()** method may be used to return the window handle of the dialog and this **HWND** may be used to update the controls.

**Prototype:**

**virtual void BeginActivate(BOOL reset=TRUE);**

**Remarks:**

This method is called when the binding becomes active.

**Parameters:**

**BOOL reset=TRUE**

If TRUE 3ds max is being reset; otherwise this is the first time the binding is becoming active.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void EndActivate();**

**Remarks:**

This method is called when the binding has been released.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void Accumulate(TimeValue t);**

**Remarks:**

This method is called 50 times per second during motion capture.

To understand how this is used consider the following two situations for a motion capture device:

1. The motion capture device is a joystick, and the position of the joystick directly maps to a range of some parameter. In this case, if you need to evaluate the parameter, you simply evaluate the joystick (inside the **Eval()** method). The position establishes the value.
2. A different case is where you have a parameter at a starting value, and if the joystick is moved, to say the right, the value is incremented. If the joystick is moved to the left the value is decremented. In this case the value can theoretically reach any value. What is needed is for the value to be incremented and decremented in a consistent fashion. If the joystick is

polled only during the **Eval()** method, and the value is incremented or decremented there, it may be a problem. If two things are using the same motion capture device, the value will be incremented or decremented twice inside **Eval()**. This will cause the value to grow or shrink twice as fast. If three things evaluated the same joystick it would move three times as fast because it would get incremented three times per frame. The solution is to use this method. It is called 50 times per second. The increments are done inside this method, and when the **Eval()** method is called the accumulated state is simply returned. This works because the method is called a fixed number of times per second regardless of the number of items evaluating the device.

**Parameters:**

**TimeValue t**

The current time when this method is called.

**Default Implementation:**

```
{}
```

## Class IMCapManager

See Also: [Class MCDeviceBinding](#), [Class IMCapManager](#).

class IMCapManager

### Description:

This class is available in release 2.0 and later only.

This is an interface into the motion capture manager. This interface is passed to plug-ins derived from class **MCDeviceBinding**. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual void MidiNote(int channel, int note)=0;**

#### Remarks:

This method is obsolete.

#### Prototype:

**virtual TimeValue GetTime()=0;**

#### Remarks:

This method returns the current time at which it is called. This is an aide for devices where interaction is happening asynchronously (the MIDI interface is an example of this). The MIDI motion capture device uses a separate thread to track the MIDI keyboard. When the user presses a key, the MIDI device plug-in needs to record which key was pressed and when. It calls this method to grab the current 3ds max time at which it happened.

## Class DPoint3

class DPoint3

### Description:

This class describes a 3D point using double precision x, y and z coordinates. Methods are provided to add and subtract points, multiply and divide by scalars, and element by element multiply and divide two points. All methods are implemented by the system.

### Data Members:

public:

**double x,y,z;**

### Methods:

### Prototype:

**DPoint3()**

### Remarks:

Constructor. No initialization is performed.

### Prototype:

**DPoint3(double X, double Y, double Z)**

### Remarks:

Constructor. x, y, and z are initialized to the values specified.

### Prototype:

**DPoint3(const DPoint3& a)**

### Remarks:

Constructor. x, y, and z are initialized to the DPoint3 specified.

### Prototype:

**DPoint3(double af[3])**

### Remarks:

Constructor. x, y, and z are initialized to. af[0], af[1], and af[2] respectively.

**Operators:**

**Prototype:**

**double& operator[](int i)  
const double& operator[](int i) const**

**Remarks:**

Allows access to x, y and z using the subscript operator.

**Return Value:**

An index of 0 will return x, 1 will return y, 2 will return z.

**Prototype:**

**operator double\*()**

**Remarks:**

Conversion function. Returns the address of the DPoint3.x

**Prototype:**

**DPoint3 operator-() const**

**Remarks:**

Unary - operator. Negates both x, y and z.

**Prototype:**

**DPoint3 operator+() const**

**Remarks:**

Unary +. Returns the point unaltered.

**Prototype:**

**DPoint3& operator-=(const DPoint3&);**

**Remarks:**

Subtracts a DPoint3 from this DPoint3.

**Prototype:**

**DPoint3& operator+=(const DPoint3&);**

**Remarks:**

Adds a DPoint3 to this DPoint3.

**Prototype:**

**DPoint3& operator\*=(double);**

**Remarks:**

Each element of this DPoint3 is multiplied by the specified double.

**Prototype:**

**DPoint3& operator/=(double);**

**Remarks:**

Each element of this DPoint3 is divided by the specified double.

**Prototype:**

**DPoint3 operator-(const DPoint3&) const;**

**Remarks:**

Subtracts a DPoint3 from a DPoint3.

**Prototype:**

**DPoint3 operator+(const DPoint3&) const;**

**Remarks:**

Adds a DPoint3 to a DPoint3.

**Prototype:**

**double operator\*(const DPoint3&) const;**

**Remarks:**

Computes the dot product of this DPoint3 and the specified DPoint3.

**Prototype:**

**DPoint3 operator^(const DPoint3&) const;**

**Remarks:**

Computes the cross product of this DPoint3 and the specified DPoint3.

The following functions are not methods of DPoint3 but are available for use:

**Prototype:**

**double Length(const DPoint3&);**

**Remarks:**

Returns the 'Length' of the point. This is  $\text{sqrt}(v.x*v.x+v.y*v.y+v.z*v.z)$

**Prototype:**

**int MaxComponent(const DPoint3&);**

**Remarks:**

Returns the component with the maximum absolute value. 0=x, 1=y, 2=z.

**Prototype:**

**int MinComponent(const DPoint3&);**

**Remarks:**

Returns the component with the minimum absolute value. 0=x, 1=y, 2=z.

**Prototype:**

**DPoint3 Normalize(const DPoint3&);**

**Remarks:**

Returns a unit vector. This is a DPoint3 with each component divided by the point **Length()**.

**Prototype:**

**DPoint3 CrossProd(const DPoint3& a, const DPoint3& b);**

**Remarks:**

Returns the cross product of two DPoint3s.

**Prototype:**

**double DotProd(const DPoint3& a, const DPoint3& b);**

**Remarks:**

Returns the dot product of two DPoint3s.

## Class IKey

See Also: [Class IKeyControl](#), [Class Animatable](#).

class IKey

### Description:

This is the base class for keys that are part of the controller interface. This class stores the time of the key, and some flags that describe the properties of the key.

### Data Members:

public:

#### **TimeValue time;**

The time of the key.

#### **DWORD flags;**

The flag bits for keys. One or more of the following values:

General flags

#### **IKEY\_SELECTED**

The key is selected.

#### **IKEY\_XSEL**

In the function curve editor X is selected.

#### **IKEY\_YSEL**

In the function curve editor Y is selected.

#### **IKEY\_ZSEL**

In the function curve editor Z is selected.

#### **IKEY\_FLAGGED**

The key is flagged. See **Animatable::FlagKey()**.

#### **IKEY\_TIME\_LOCK**

The key is locked in time so it won't move.

TCB specific key flags:

#### **TCBKEY\_QUATVALID**

The quaternion TCB key has inside it both a quaternion and an angle axis. When this bit is set the angle/axis is derived from the quaternion instead of vice/versa.

Bezier specific key flags:

## **BEZKEY\_CONSTVELOCITY**

This key is interpolated using arclength as the interpolation parameter.

## **BEZKEY\_XBROKEN**

## **BEZKEY\_YBROKEN**

## **BEZKEY\_ZBROKEN**

Indicates if the tangent handles are locked together. Broken means not locked.

The following macros may be used to test and set the tangent locks:

**TangentsLocked(f,j);**

**SetTangentLock(f,j,l);**

The following macros may be used to access the hybrid tangent types:

**GetInTanType(f);**

**GetOutTanType(f);**

**SetINTanType(f,t);**

**SetOutTanType(f,t);**

### **Methods:**

### **Prototype:**

**IKey();**

### **Remarks:**

Constructor. The time and flags are set to zero.

## Class ITCBKey

See Also: [Class IKey](#).

```
class ITCBKey : public IKey
```

### Description:

This is the base class for Tension Continuity and Bias keys.

### Data Members:

public:

#### **float tens;**

The tension setting. Values are in the range -1.0 to 1.0, where 0.0 is the default.

#### **float cont;**

The continuity setting. Values are in the range -1.0 to 1.0, where 0.0 is the default.

#### **float bias;**

The bias setting. Values are in the range -1.0 to 1.0, where 0.0 is the default.

#### **float easeIn;**

The ease in value. Values are in the range 0.0 to 50.0, where 25.0 is the default.

#### **float easeOut;**

The ease out value. Values are in the range 0.0 to 50.0, where 25.0 is the default.

## List of Mapping Channel Index Values

See Also: [Class Object](#), [Class Mesh](#).

The mesh mapping channel may be specified as one of the following:

**0**: Vertex Color channel.

**1**: Default mapping channel (the **TVert** array).

**2** through **MAX\_MESHMAPS-1**: The new mapping channels available in release 3.0.

## Class Ray

See Also: [Class Point3](#).

class Ray

### Description:

This class describes a vector in space using an origin point **p**, and a unit direction vector **dir**.

### Data Members:

public:

**Point3 p;**

Point of origin.

**Point3 dir;**

Unit direction vector.

## List of Object Channel Numbers

See Also: [List of Channel Bits](#).

The following are the indices that may be used for the object channels. These values are used in the methods **Object::UpdateValidity()**, **Object::SetChannelValidity()** and **Object::ChannelValidity()**.

### **TOPO\_CHAN\_NUM**

The topology channel number. The actual value is 0.

### **GEOM\_CHAN\_NUM**

The geometry channel number. The actual value is 1.

### **TEXMAP\_CHAN\_NUM**

The texture vertices and procedural mappings channels number. The actual value is 2.

### **MTL\_CHAN\_NUM**

This is no longer used. Materials are rolled into the topology channel number. The actual value is 3.

### **SELECT\_CHAN\_NUM**

The sub-object selection channel number. The actual value is 4.

### **SUBSEL\_TYPE\_CHAN\_NUM**

This is the current level of selection number. The actual value is 5.

### **DISP\_ATTRIB\_CHAN\_NUM**

The display channel number. The actual value is 6.

### **VERT\_COLOR\_CHAN\_NUM**

The vertex colors number. This is also used for the second mapping channel. The actual value is 7.

### **GFX\_DATA\_CHAN\_NUM**

The stripping, edge list, etc. channel number. The actual value is 8.

### **DISP\_APPROX\_CHAN\_NUM**

The displacement approximation channel number. The actual value is 9.

### **EXTENSION\_CHAN\_NUM**

This channel number is available in release 4.0 and later only.

The extension object channel number. The actual value is 10.

Note: Developers must not get confused between these object channel numbers (**TOPO\_CHAN\_NUM**, **GEOM\_CHAN\_NUM**, etc.) and the

channel bits (**TOPO\_CHANNEL**, **GEOM\_CHANNEL**, etc.). Some methods refer to the channel by number and some by bit. Developers must not confuse these two as the compiler will not catch this as an error. See [List of Channel Bits](#).

# Class RNormal

See Also: [Class Mesh](#), [Class RVertex](#), [Class GraphicsWindow](#).

class RNormal

## Description:

The 'rendered' normal class. **RNormals** are stored as unit vectors in object space. An instance of this class is maintained by the **RVertex** class since the **RVertex** is what gets rendered (lit), and the lighting methods need to know about the normals. All methods of this class are implemented by the system.

**Note:** This class is used internally by 3ds max. Developers who need to compute face and vertex normals for a mesh should instead refer to the Advanced Topics section [Computing Face and Vertex Normals](#).

## Data Members:

private:

### **Point3 normal;**

The normal as a unit vector. Note that if you set this normal, you should call the method **normalize()** if you are not sure the normal is already unit length.

### **DWORD smGroup;**

The smoothing group. Normals are attached to smoothing groups in the sense that one vertex, if it is shared by two triangles with different smoothing groups, will have two normals associated with it -- one for each smoothing group.

### **MtlID mtlIndex;**

The material index. If one vertex is shared between two materials, there will be two normals. Therefore the material index is stored with the normal as well.

Note: **typedef unsigned short MtlID;**

### **Point3 rgb;**

The RGB value.

## Methods:

## Prototype:

**RNormal();**

**Remarks:**

Constructor. The smoothing group and material index are set to zero.

**Prototype:**

```
void setNormal(const Point3 &nor);
```

**Remarks:**

Sets the normal to the specified value.

**Parameters:**

```
const Point3 &nor
```

The normal value to set.

**Prototype:**

```
void addNormal(const Point3 &nor);
```

**Remarks:**

Adds the specified value to the value of the normal.

**Parameters:**

```
const Point3 &nor
```

The normal value to add to the existing value.

**Prototype:**

```
void normalize();
```

**Remarks:**

Converts the normal to a unit vector.

**Prototype:**

```
Point3 &getNormal();
```

**Remarks:**

Returns the normal.

**Prototype:**

```
void setSmGroup(DWORD g);
```

**Remarks:**

Sets the smoothing group to the specified value.

**Parameters:**

**DWORD g**

The smoothing group value to set.

**Prototype:**

**void addSmGroup(DWORD g);**

**Remarks:**

ORs the specified smoothing group value to the existing value.

**Parameters:**

**DWORD g**

The smoothing group bits to set.

**Prototype:**

**DWORD getSmGroup();**

**Remarks:**

Returns the smoothing group value.

**Prototype:**

**void setMtlIndex(MtlID i);**

**Remarks:**

Sets the material index to the specified value.

**Parameters:**

**MtlID i**

The material index value to set.

**Prototype:**

**MtlID getMtlIndex();**

**Remarks:**

Returns the material index.

**Prototype:**

**void setRGB(Point3 &clr);**

**Remarks:**

Sets the RGB value.

**Parameters:**

**Point3 &clr**

The RGB color to set.

**Prototype:**

**Point3 &getRGB();**

**Remarks:**

Returns the RGB value.

## Class RVertex

See Also: [Class RNormal](#), [Class Mesh](#), [Class GraphicsWindow](#).

class RVertex

### Description:

A **RVertex** is a rendered vertex. A vertex becomes a **RVertex** after it has been transformed. A **RVertex** has a position (x, y, z coordinate) that is stored in device coordinates. These are stored in the data member **fPos[3]**.

One vertex in a **Mesh** can be shared between many different smoothing groups. In the 3ds max **Mesh** database, the vertices are shared, however the normals are not. This is why an **RVertex** has a **RNormal** data member. For example, if you had a sphere that had the top and bottom hemi-spheres smoothed separately (i.e. not smoothed across the equator), then the vertices across the equator would have two **RNormals** for each **RVertex** while the other vertices would have one. There may be as many **RNormals** as there are smoothing groups colliding at a vertex. However, it is by far the most common case to have one, and anything other than one or two is very rare.

For purposes of smoothing, as many **RNormals** are allocated as required and are stored in this class. **RNormals** are kept in this **RVertex** class since the **RVertex** is what gets rendered (lit). When you light a vertex you need to know its normal direction. Thus the **RNormal(s)** are stored in this class (using data member **rn** or **\*ern**).

All methods of this class are implemented by the system.

**Note:** This class is used internally by 3ds max. Developers who need to compute face and vertex normals for a mesh should instead refer to the Advanced Topics section [Computing Face and Vertex Normals](#).

### Data Members:

public:

#### **DWORD rFlags;**

The flags contain the clip flags, the number of normals at the vertex, and the number of normals that have already been rendered. These are used internally. For example, the clipping flags are used to see if the RVertex can be either trivially accepted or rejected when rendering.

```

#define NORCT_MASK  0x000000ffUL
#define SPECIFIED_NORMAL 0x00004000UL
#define OUT_LEFT  0x00010000UL
#define OUT_RIGHT  0x00020000UL
#define OUT_TOP    0x00040000UL
#define OUT_BOTTOM 0x00080000UL
#define RECT_MASK  0x000f0000UL
#define RND_MASK   0xfff00000UL
#define RND_NOR0   0x00100000UL
#define RND_NOR(n) (RND_NOR0 << (n))
union {
  int iPos[3];
  // This position is no longer used.
  float fPos[3];
  // This is used to store the position in device coordinates, [0]=x, [1]=y,
  // [2]=z.
};

```

**RNormal rn;**

If a single RNormal is used, it is stored here.

**RNormal \*ern;**

In some cases, there may be two or more RNormals per vertex. If this is the case, these 'extra' RNormals are allocated and the pointer to the memory is stored here. If these are used, then data member **rn** is not used (**rn** is copied into **ern[0]**).

**Methods:**

**Prototype:**

**RVertex();**

**Remarks:**

Constructor. The flags are set to zero and the **ern** pointer is set to NULL.

**Prototype:**

**~RVertex();**

**Remarks:**  
Destructor.

## Class MeshMap

See Also: [Class Mesh](#).

class MeshMap

### Description:

This class is available in release 3.0 and later only.

In 3ds max 3.0 and later the user may work with more than 2 mapping channels. When the mapping channel is set to a value greater than 1 (by using a UVWMap Modifier for example) then an instance of this class is allocated for each channel up to the value specified. It maintains the mapping information for a single channel.

An array of instances of this class is carried by the Mesh class in the public data member:

**MeshMap \*maps;**

All methods of this class are implemented by the system.

### Data Members:

public:

**DWORD flags;**

The mapping flags. One or more of the following values:

**MESHMAP\_USED**

Indicates this mapping channel is actually used (carries mapping information).

**MESHMAP\_TEXTURE**

Indicates this is a texture mapping channel.

**MESHMAP\_VERTCOLOR**

Indicates this is a vertex color channel.

**MESHMAP\_USER**

Indicates the channel is used for a developer purpose.

**UVVert \*tv;**

Array of texture vertices. This stores the UVW coordinates for the mapping channel. **Note: typedef Point3 UVVert;**

**TVFace \*tf;**

The texture vertex faces. There needs to be one TVFace for every face in the

Mesh, but there can be three indices into the UVVert array that are any UV's.

**int vnum;**

The number of elements in the **UVVert** array.

**int fnum;**

The number of elements in the **TVFace** array.

**The following global functions are not part of the class but are available for use**

**Prototype:**

**DWORD MapChannelID (int mp);**

**Remarks:**

This method is available in release 4.0 and later only.

Returns the Channel ID of the map channel. if mp>=1, this always returns **TEXMAP\_CHANNEL**. For mp<1, including the hidden map channels, this is currently always **VERTCOLOR\_CHANNEL**. In the future it may include map channels that are actually part of **GEOM\_CHANNEL** or something.

**Parameters:**

**int mp**

The map channel.

**Prototype:**

**int MapChannelNum (int mp);**

**Remarks:**

This method is available in release 4.0 and later only.

Similar to MapChannelID, but this returns the CHAN\_NUM version: **TEXMAP\_CHAN\_NUM**, **VERTCOLOR\_CHAN\_NUM**, etc.

**Parameters:**

**int mp**

The map channel.

**Methods:**

public:

**Prototype:**

**MeshMap();**

**Remarks:**

Constructor. The flags are cleared, the vertex and face numbers are set to 0, and the tv and tf pointers are set to NULL.

**Prototype:**

**~MeshMap();**

**Remarks:**

Destructor. If the tv and tf arrays are allocated they are deleted.

**Prototype:**

**int getNumVerts();**

**Remarks:**

Returns the number of **UVVerts**.

**Prototype:**

**void setNumVerts(int vn, BOOL keep=FALSE);**

**Remarks:**

Sets the number of UVVerts allocated to the specified value.

**Parameters:**

**int vn**

The new number of UVVerts to allocate.

**BOOL keep=FALSE**

If TRUE any previously allocated UVVerts are maintained (up to the maximum set by **vn**). If FALSE they are discarded.

**Prototype:**

**int getNumFaces();**

**Remarks:**

Returns the number of TVFaces.

**Prototype:**

```
void setNumFaces(int fn, BOOL keep=FALSE);
```

**Remarks:**

Set the number of TVFaces allocated to the specified value.

**Parameters:**

**int fn**

The new number of TVFaces to allocate.

**BOOL keep=FALSE**

If TRUE any previously allocated TVFaces are maintained (up to the maximum set by **fn**). If FALSE they are discarded.

**Prototype:**

```
void Clear();
```

**Remarks:**

Clears (deletes) the **tv** and **tf** arrays and sets the counts to zero.

**Prototype:**

```
BitArray GetIsoVerts();
```

**Remarks:**

This method returns a BitArray with size vnum, where isolated (unused) vertices are selected.

**Prototype:**

```
void DeleteVertSet(BitArray set, BitArray *delFace=NULL);
```

**Remarks:**

This method is used to delete vertices from a mesh map.

**Parameters:**

**BitArray set**

The array of bits where mapping vertices you want to delete are set.

**BitArray \*delFace=NULL**

This is an optional parameter. If non-NULL, it's filled with a record of which faces, if any, were using the specified map verts and should therefore be deleted or considered invalid. (Note: in normal usage, it's preferable to remove any compromised faces *\_before\_* deleting vertices, so this parameter would rarely be used.)

**Prototype:**

**void DeleteFaceSet(BitArray set, BitArray \*isoVert=NULL);**

**Remarks:**

This method is used to delete faces from a mesh map.

**Parameters:**

**BitArray set**

This is a list of mapping faces to delete.

**BitArray \*isoVert=NULL**

If non-NULL, this BitArray is filled with a list of map vertices that were used by the deleted faces but not by any remaining faces. (This is a list of "newly isolated" map vertices.)

NOTE: The number and arrangement of faces in a MeshMap should always agree with the "parent" mesh. It's safest in most cases to just let this be handled by **Mesh::DeleteFaceSet()**.

**Prototype:**

**void SetFlag(DWORD fl);**

**Remarks:**

Sets the specified flag(s).

**Parameters:**

**DWORD fl**

The flags to set. See the public data member **flags** above.

**Prototype:**

**void ClearFlag(DWORD fl);**

**Remarks:**

Clears the specified flag(s).

**Parameters:**

**DWORD fl**

The flags to clear. See the public data member **flags** above.

**Prototype:**

**BOOL GetFlag(DWORD fl);**

**Remarks:**

Returns TRUE if the specified flag(s) are set; otherwise FALSE.

**Parameters:**

**DWORD fl**

The flags to check. See the public data member **flags** above.

**Prototype:**

**BOOL IsUsed();**

**Remarks:**

Returns TRUE if this mapping channel is being used; otherwise FALSE.

**Prototype:**

**void SwapContents(MeshMap &from);**

**Remarks:**

Exchanges the data between this MeshMap object and the specified one. The **flags**, **vnum** and **fnum** values are exchanged. The **UVVert** and **TVFace** pointers are swapped.

**Parameters:**

**MeshMap &from**

The MeshMap instance to swap with.

**Prototype:**

**MeshMap & operator=(MeshMap & from);**

**Remarks:**

Assignment operator.

**Parameters:**

**MeshMap &from**

The MeshMap to assign.

# Class PerData

See Also: [Class Mesh](#).

class PerData

## Description:

This class is available in release 3.0 and later only.

This class is used for per-'something' floating-point information. For example, it is used with Meshes to keep track of such per-**vertex** information as weighted (Affect Region or Soft) selections and vertex weights. It is used in MNMesh to store per-**edge** data (edge weights).

Currently there's only one "type" of data supported, floating point values, but this may be extended in the future. PerData arrays in Meshes and MNMeshes cannot be reserved for plug-ins at this time; 3ds max maintains the list in MESH.H of the reserved vertex data channels, and in MNMESH.H for the MNEdge data channels.

The methods of this class are deliberately made to look like Tab<> methods. All methods of this class are implemented by the system.

## Data Members:

All data members are public.

### **int dnum;**

The number of elements of per-vertex data.

### **int type;**

The type of data held by this class. At present the only 3ds max defined option is:

**PERDATA\_TYPE\_FLOAT**

### **int alloc;**

The number of elements currently allocated in the **data** array.

### **void \*data;**

Points to the actual data.

**The following functions are not methods of this class but are available for use:**

## Function:

**int VertexDataType(int vdID);**

**Remarks:**

Returns the type of data supported, i.e. **PERDATA\_TYPE\_FLOAT**.

**Parameters:**

**int vdID**

This parameter is ignored.

**Function:**

**void \*VertexDataDefault(int vdID);**

**Remarks:**

Returns a pointer to a default floating point value for the specified channel.

**Parameters:**

**int vdID**

One of the following values:

**VDATA\_SELECT** - Soft Selection

**VDATA\_WEIGHT** - Vertex weights (for NURMS MeshSmooth)

**VDATA\_ALPHA** - Vertex Alpha values (R4 and later only)

**VDATA\_CORNER** - Cornering values for subdivision use (R4 and later only)

**Methods:**

public:

**Prototype:**

**PerData();**

**Remarks:**

Constructor. The number of elements is set to 0, the type is set to 0 and the data pointer is set to NULL.

**Prototype:**

**PerData(int n, int tp);**

**Remarks:**

Constructor. The data members are initialized as follows:

**data=NULL; dnum=0; alloc=0; type=tp;**

**Parameters:**

**int n**

The number of elements to allocate.

**int tp**

The type to set.

**Prototype:**

**~PerData();**

**Remarks:**

Destructor. Any allocated data is freed and the count and type are set to 0.

**Prototype:**

**int DataSize();**

**Remarks:**

Returns the number of bytes used by the base data type for the vertex data.

This is only implemented for a type of **VDATA\_TYPE\_FLOAT** in which case it returns **sizeof(float)**. Other cases simply return 0.

**Prototype:**

**void \*AllocData(int num);**

**Remarks:**

Allocates and returns a pointer to an array of floats of the specified size.

**Parameters:**

**int num**

The number of floats to allocate.

**Prototype:**

**void FreeData(void \*addr);**

**Remarks:**

Deletes the specified array of floats.

**Parameters:**

**void \*addr**

Points to the array of floats to free.

**Prototype:**

**void \*Addr(void \*ptr, int at);**

**Remarks:**

Returns the address of the specified element in the array passed.

**Parameters:**

**void \*ptr**

The array whose **at**-th element address is returned.

**int at**

The zero based index of the element.

**Prototype:**

**void \*Addr(int at);**

**Remarks:**

Returns the address of the specified element in the **data** array.

**Parameters:**

**int at**

The zero based index of the element.

**Prototype:**

**void CopyData(void \*to, void \*from, int num=1);**

**Remarks:**

Copies the specified number of elements between the two data arrays.

**Parameters:**

**void \*to**

Points to the destination data array.

**void \*from**

Points to the source data array.

**int num=1**

The number of elements to copy.

**Prototype:**

**void CopyData(int to, int from, int num=1);**

**Remarks:**

Copies the specified number of elements between the two specified locations in the **data** array.

**Parameters:**

**int to**

The zero based index into the **data** array of the destination.

**int from**

The zero based index into the **data** array of the source.

**int num=1**

The number of elements to copy.

**Prototype:**

**void WeightedSum(void \*to, void \*fr1, float prop1, void \*fr2, float prop2);**

**Remarks:**

Computes the weighted sum of the arguments passed. This is effectively  $c = a*prop1 + b*prop2$ .

This is used, for example, in splitting an edge, where we would want to interpolate the vertex weight values from the edge's endpoints to create the weight for the new vertex.

**Parameters:**

**void \*to**

A pointer to the location in which the result should be stored.

**void \*fr1**

A pointer to the first value to be summed.

**float prop1**

The weight given to the first value.

**void \*fr2**

A pointer to the second value.

**float prop2**

The weight given to the second value.

**Prototype:**

**void WeightedSum(int to, int fr1, float prop1, int fr2, float prop2);**

**Remarks:**

Computes the weighted sum of the arguments passed. This is similar to the method above except to, fr1, and fr2 are indices of the values in the PerData array. That is, PerData::WeightedSum (c, a, prop1, b, prop2), where a, b, and c are ints between 0 and PerData::dnum-1, is equivalent to the call PerData::WeightedSum (PerData::Addr(c), PerData::Addr(a), prop1, PerData::Addr(b), prop2).

**Parameters:**

**int to**

The index in the PerData array of the location in which the result should be stored.

**int fr1**

The index of the first value to be summed in the PerData array.

**float prop1**

The weight given to the first value.

**int fr2**

The index of the second value to be summed in the PerData array.

**float prop2**

The weight given to the second value.

**Prototype:**

**void setAlloc(int num, BOOL keep=TRUE);**

**Remarks:**

Sets the number of elements allocated in the **data** array.

**Parameters:**

**int num**

The number of elements to allocate.

**BOOL keep=TRUE**

If TRUE previous values are kept (copied to the new storage); otherwise they are discarded.

**Prototype:**

**void SetCount(int num, BOOL keep = FALSE);**

**Remarks:**

Sets the number of elements allocated in the **data** array and sets the **dnum** member to **num**.

**Parameters:**

**int num**

The number of elements to allocate.

**BOOL keep = FALSE**

If TRUE previous values are kept (copied to the new storage); otherwise they are discarded.

**Prototype:**

**void Shrink();**

**Remarks:**

Reduces the size of the **data** array to contain **dnum** elements.

**Prototype:**

**int Count();**

**Remarks:**

Returns the number of elements used (**dnum**)

**Prototype:**

**void Clear();**

**Remarks:**

Clears (deletes) any allocated data and sets the count and type to 0.

**Prototype:**

**void DeleteSet(BitArray del);**

**Remarks:**

Removes any element whose corresponding element in the BitArray is not set.

**Parameters:**

**BitArray del**

Specifies which elements to delete. Data elements corresponding to bits that are on remain; for bits that are off the elements are deleted.

**Prototype:**

**void Delete(int at, int num);**

**Remarks:**

Deletes the specified number of elements from the specified location in the data array.

**Parameters:**

**int at**

The location to delete elements.

**int num**

The number of elements to delete.

**Prototype:**

**void Insert(int at, int num, void \*el);**

**Remarks:**

Inserts the specified number of data elements into the specified location in the **data** array.

**Parameters:**

**int at**

The zero based index of the location for the insert.

**int num**

The number of elements to insert.

**void \*el**

The data to insert.

**Prototype:**

**void Append(int num, void \*el);**

**Remarks:**

Appends the specified elements to the **data** array.

**Parameters:**

**int num**

The number of elements to append.

**void \*el**

The data to append.

**Prototype:**

**void InsertCopies(int at, int num, void \*el);**

**Remarks:**

Inserts the specified number of elements into the **data** array at the given location.

**Parameters:**

**int at**

The zero based index of the location to insert the data.

**int num**

The number of elements to insert.

**void \*el**

**Prototype:**

**void AppendCopies(int num, void \*el);**

**Remarks:**

Appends the specified number of elements to the **data** array.

**Parameters:**

**int num**

The number of elements to append.

**void \*el**

The data to append.

**Prototype:**

**void SwapContents(PerData &from);**

**Remarks:**

Swaps the contents of this PerData object and the specified one.

**Parameters:**

**PerData &from**

The object to swap with.

**Prototype:**

**PerData &operator=(PerData &from);**

**Remarks:**

Assignment operator.

**Parameters:**

**PerData &from**

The VertexData source.

## List of Vertex Data Index Options

See Also: [Class Mesh](#), [Class MNMesh](#), [Class PerData](#).

The following are the vertex data channel index values for use with the vertex data methods of class **Mesh**, class **EPoly**, and class **MNMesh**.

**VDATA\_SELECT**: The vertex soft selection data. This is index 0.

**VDATA\_WEIGHT**: The vertex weight data. This is index 1.

2 through **MAX\_VERTDATA-1**: Developer defined data. Note: **#define MAX\_VERTDATA 100**

## Class SubObjHitList

See Also: [Class MeshSubHitRec](#).

class SubObjHitList

### Description:

This class describes a list of sub-object hit records. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**SubObjHitList();**

#### Remarks:

Constructor. The list is set to NULL.

#### Prototype:

**~SubObjHitList()**

#### Remarks:

Destructor. All the hit records are deleted.

#### Prototype:

**MeshSubHitRec \*First();**

#### Remarks:

Returns the first item in the hit list.

#### Prototype:

**void AddHit( DWORD dist, int index );**

#### Remarks:

Allocates and adds a new hit record to the list.

#### Parameters:

**DWORD dist**

The distance of the hit.

**int index**

The index of the hit.

## List of Mesh Flags

See Also: [Class Mesh](#).

One or more of the following values:

### **MESH\_EDGE\_LIST**

This flag is obsolete.

### **MESH\_LOCK\_RENDDATA**

Setting this flag prevents render data from being deleted (except when the mesh is deleted).

### **MESH\_BEEN\_DSP**

This flag is reserved for internal use.

## Class MeshOpProgress

See Also: [Class Mesh](#).

class MeshOpProgress

### Description:

A callback used while doing a lengthy operation to a mesh. A developer creates an instance of this class and passes a pointer to it into the **CalcBoolOp()** or **Optimize()** function.

### Methods:

#### Prototype:

**virtual void Init(int total)=0;**

#### Remarks:

Implemented by the Plug-In.

This method is called once with the total number of increments.

#### Parameters:

**int total**

The total number of increments.

#### Prototype:

**virtual BOOL Progress(int p)=0;**

Implemented by the Plug-In.

This method is called over and over with a new value for **p**. The percentage complete is **p/total**.

#### Parameters:

**int p**

The number completed so far.

#### Return Value:

If TRUE processing will continue. If FALSE processing is aborted.

## Class MeshRenderData

See Also: [Class Mesh](#).

class MeshRenderData

### Description:

A developer may derive a class from this class, put any required data in it, and then hang this data off a Mesh. This is done using the methods of Class Mesh:

**void SetRenderData(MeshRenderData \*p);**

Sets the mesh render data hung off this Mesh.

**MeshRenderData \* GetRenderData();**

Gets a pointer to the MeshRenderData.

### Methods:

### Prototype:

**virtual void DeleteThis()=0;**

### Remarks:

Implemented by the Plug-In.

Deletes this instance of the class.

## Class FlyOffData

See Also: [Custom Controls](#), [Class ICustButton](#).

class FlyOffData

### Description:

This class uses four indices into the image list to describe the button in each of the possible states: Out&Enabled, In&Enabled, Out&Disabled and In&Disabled. An array of instances of this class are passed into the method

**ICustButton::SetFlyOff()**.

### Data Members:

These four data members are indices into the image list. They indicate which images to use for each of the four possible button states:

You may specify a unique image for each one of these states by passing a different index for each state. Or you may supply a single image to be used for all the states by specifying the same index four times.

public:

**int iOutEn;**

Out&Enabled.

**int iInEn;**

In&Enabled

**int iOutDis;**

Out&Disabled.

**int iInDis;**

In&Disabled.

## Class MaxBmpFileIcon

See Also: [Class MaxIcon](#), [Class ICustButton](#).

class MaxBmpFileIcon: public MaxIcon

### Description:

This class is available in release 4.0 and later only.

This implementation of MaxIcon is for the icon images that are stored as ".bmp" files in 3ds max's UI directory. This is used by the macroScript facility in MAXScript to specify icons. See the MAXScript documentation for "macroScript" for the exact meaning of the filename and index and details.

### Methods:

public:

### Prototype:

**MaxBmpFileIcon(TCHAR\* pFilePrefix, int index);**

### Remarks:

Constructor.

### Parameters:

**TCHAR\* pFilePrefix**

The file prefix to initialize with.

**int index**

The index of the icon.

### Prototype:

**MaxBmpFileIcon(SClass\_ID sid, Class\_ID cid);**

### Remarks:

Constructor.

### Parameters:

**SClass\_ID sid**

The superclass ID

**Class\_ID cid**

The class ID/

**Prototype:**

**HIMAGELIST GetDefaultImageList();**

**Remarks:**

Returns the handle to the image list for the size of icons that the user has chosen.

**Prototype:**

**HIMAGELIST GetSmallImageList();**

**Remarks:**

Returns the image list for small icons.

**Prototype:**

**HIMAGELIST GetLargeImageList();**

**Remarks:**

Returns the image list for large icons.

**Prototype:**

**int GetSmallImageIndex(bool enabledVersion = true, COLORREF  
backgroundColor = GetCustSysColor(COLOR\_BTNFACE));**

**Remarks:**

Returns the zero based index into the image list for the small version of this particular icon.

**Parameters:**

**bool enabledVersion = true**

Pass true for the enabled version of the icon; false for the disabled version.

**COLORREF backgroundColor =  
GetCustSysColor(COLOR\_BTNFACE)**

The background color for use in alpha blending. The files that define these icons always have an alpha mask, and so a background color is needed to blend it with.

**Prototype:**

```
int GetLargeImageIndex(bool enabledVersion = true,  
COLORREF backgroundColor =  
GetCustSysColor(COLOR_BTNFACE));
```

**Remarks:**

Returns the zero based index into the image list for the large version of this particular icon.

**Parameters:**

**bool enabledVersion = true**

Pass true for the enabled version of the icon; false for the disabled version.

**COLORREF backgroundColor =  
GetCustSysColor(COLOR\_BTNFACE)**

The background color for use in alpha blending. The files that define these icons always have an alpha mask, and so a background color is needed to blend it with.

**Prototype:**

```
bool UsesAlphaMask();
```

**Remarks:**

Returns true if the icon uses an alpha mask; otherwise false.

**Prototype:**

```
TSTR& GetFilePrefix();
```

**Remarks:**

Returns the directory of the icon.

**Prototype:**

```
int GetIndex();
```

**Remarks:**

Returns the index of the icon in the image file list.

## Class SplineKnot

See Also: [Class Spline3D](#).

class SplineKnot

### Description:

This class describes a single knot in a spline.

In 3ds max 2.0 and later there are methods which provide full access to the private data members of the class.

All methods of this class are implemented by the system.

### Description:

private:

**int ktype;**

The knot type.

**int ltype;**

The line type.

**Point3 point;**

The point location.

**Point3 inVec;**

The in vector.

**Point3 outVec;**

The out vector.

**int aux;**

This is simply an integer value which may be used for temporary storage of data associated with the knot. This data will be overwritten by the internal EditSpline code.

**int aux2;**

Used to track topo changes in spline editing.

### Spline Knot Summary:

**Picture a bezier spline with three knots, going from left to right:**

**A--->AB----BA<---B--->BC----CB<---C**

The knot points are A, B and C. The vectors are labeled the same as patch vectors (AB is the vector from A going toward B, the vector from B to A is labeled BA, and so on).

In this diagram, AB is the OUT vector for knot A. BA is the IN vector for knot B. BC is the OUT vector for knot B, and CB is the IN vector for knot C. Because this is an open spline, knot A doesn't use its IN vector, and knot C doesn't use its OUT vector.

The IN and OUT terminology is based on the way a spline flows from the first knot to the last. If the spline is reversed, the IN and OUT sense is reversed, as well.

Regarding the vectors, the only difference between a circle and a square is that the square has vectors that are at the same location as the knot point (in other words, zero length vectors) causing sharp corners. The circle uses vectors which cause each segment to bulge to form a quarter-circle.

Take a look at the `\MAXSDK\SAMPLES\OBJECTS\NGON.CPP` source file for an example of how the vectors are generated to form a linear NGON versus a circular one.

**Friend Classes:**

```
friend class Spline3D;  
friend class SplineKnotAssy;
```

**Methods:**

**Prototype:**

```
SplineKnot(int k, int l, Point3 p, Point3 in, Point3 out, int a1= -1,  
int a2= -1, int a3= -1, int Ia1= -1, int Ia2= -1, int Ia3= -1, int Oa1=  
-1, int Oa2= -1, int Oa3= -1, DWORD f=0);
```

**Remarks:**

This method is available in release 2.0 and later only.

Constructor. The data members are initialized to the values passed.

**Prototype:**

```
inline int Ktype();
```

**Remarks:**

Returns the knot type.

**Return Value:**

See [List of Spline Knot Types](#).

**Prototype:**

**inline void SetKtype(int t);**

**Remarks:**

This method is available in release 2.0 and later only.  
Sets the knot type.

**Parameters:**

**int t**

The type to set. See [List of Spline Knot Types](#).

**Prototype:**

**inline int Ltype();**

**Remarks:**

Returns the line type.

**Return Value:**

See [List of Spline Line Types](#).

**Prototype:**

**inline void SetLtype(int t);**

**Remarks:**

This method is available in release 2.0 and later only.  
Sets the line type.

**Parameters:**

**int t**

The type to set. See [List of Spline Line Types](#).

**Prototype:**

**inline int Aux();**

**Remarks:**

This method is available in release 2.0 and later only.  
Provides access to the first integer chunk of auxiliary data for the knot.

**Prototype:**

**inline void SetAux(int a);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the first integer of auxiliary data for the knot.

**Parameters:**

**int a**

The value to set.

**Prototype:**

**inline int Aux2();**

**Remarks:**

This method is available in release 2.0 and later only.

Provides access to the second integer of auxiliary data for the knot.

**Prototype:**

**inline void SetAux2(int a);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the second integer of auxiliary data for the knot.

**Parameters:**

**int a**

The value to set.

**Prototype:**

**inline Point3 Knot();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns the point location for the knot.

**Prototype:**

**inline void SetKnot(Point3 p);**

**Remarks:**

This method is available in release 2.0 and later only.  
Sets the point location for the knot.

**Parameters:**

**Point3 p**

The point to set.

**Prototype:**

**inline Point3 InVec();**

**Remarks:**

This method is available in release 2.0 and later only.  
Returns the in vector for the knot.

**Prototype:**

**inline void SetInVec(Point3 p);**

**Remarks:**

This method is available in release 2.0 and later only.  
Sets the in vector for the knot.

**Parameters:**

**Point3 p**

The in vector to set.

**Prototype:**

**inline Point3 OutVec();**

**Remarks:**

This method is available in release 2.0 and later only.  
Returns the out vector for the knot.

**Prototype:**

**inline void SetOutVec(Point3 p);**

**Remarks:**

This method is available in release 2.0 and later only.

Sets the out vector for the knot.

**Parameters:**

**Point3 p**

The out vector to set.

## Structure Knot

See Also: [Class Spline3D](#).

This structure holds data about a single knot.

```
typedef struct {
```

```
    int ktype;
```

See [List of Spline Knot Types](#).

```
    int ltype;
```

See [List of Spline Line Types](#).

```
    float du;
```

This parameter is used internally.

```
    int aux;
```

This auxiliary data is used internally in capping to link the knot point to a point on the patch mesh. A developer may use this data member to store information with the knot, but this data will be overwritten if any capping is done.

```
} Knot;
```

## List of Spline Knot Types

Knot types. One of the following values:

### **KTYPE\_AUTO**

This type of knot generates the bezier handles automatically to produce a smooth curve. With this knot type the bezier handles are invisible.

### **KTYPE\_CORNER**

This knot type produces a sharp corner.

### **KTYPE\_BEZIER**

This knot type produces bezier handles that are collinear (the bezier vectors coming out of the knot are collinear). These handles are constrained to be collinear so if the user moves one handle the other will move to remain collinear. Also if the user moves the handle towards or away from the knot the other handle will move the corresponding amount in the proper scale.

### **KTYPE\_BEZIER\_CORNER**

This knot type has bezier handles but they are not constrained to be opposite each other.

## List of Spline Line Types

Line types. One of the following values:

### **LTYPE\_CURVE**

This specifies that the segment should interpolate based on the bezier handles for the segment.

### **LTYPE\_LINE**

This specifies that the segment should go straight from knot to knot and ignore the bezier handles.

## Class ShapeHierarchy

See Also: [Class GenericHierarchy](#), [Class BitArray](#).

class ShapeHierarchy

### Description:

This class stores the hierarchy tree of a shape object. In addition it stores a **BitArray** with an entry for each polygon in the shape which indicates whether that polygon should be reversed in order to provide the proper clockwise/counterclockwise ordering for the nested shapes. All methods of this class are implemented by the system.

### Data Members:

public:

#### **GenericHierarchy hier;**

Describes the hierarchy.

#### **BitArray reverse;**

Indicates whether that polygon should be reversed in order to provide the proper clockwise / counterclockwise ordering for the nested shapes. There is one bit in the bit array for every polygon in the shape. For example, if you pass in two nested circles and they are both clockwise, the outermost circle will have its reverse bit set to indicate it should be reversed in order to be properly extruded or lofted. This is because for nested shapes the outermost circle should be counterclockwise.

### Methods:

#### **Prototype:**

**ShapeHierarchy();**

#### **Remarks:**

Constructor. No initialization is performed.

#### **Prototype:**

**ShapeHierarchy(int polys);**

#### **Remarks:**

Constructor. This constructor clears out the hierarchy, sets the number of

polygons to **poly** and clears all the bits in the BitArray.

**Parameters:**

**int polys**

The number of polygons in the hierarchy.

**Prototype:**

**void New(int polys = 0);**

**Remarks:**

This methods clears out the hierarchy, sets the number of polygons to **poly** and clears all the bits in the BitArray.

**Parameters:**

**int polys = 0**

The number of polygons in the hierarchy.

**Operators:**

**Prototype:**

**ShapeHierarchy &operator=(ShapeHierarchy &from);**

**Remarks:**

Assignment operator.

**Parameters:**

**ShapeHierarchy &from**

The source shape hierarchy.

## List of Shape Capping Types

The following are the capping types supported:

### **CAPTYPE\_MORPH**

This type of cap only uses the existing vertices in the PolyShape to generate the cap. The capping code does the best job it can given this constraint however it is possible to wind up with long sliver-like faces on the cap. This is referred to as a morph cap because if you cap a shape using this method it does not generate any new vertices and you can then morph between shapes with the same number of vertices.

### **CAPTYPE\_GRID**

This type of cap generates new vertices in the interior of the shape in a grid pattern. This helps to break up the shape and helps reduce slivering. Grid capping will generate different number of vertices based on the shape and thus the shapes are not morphable.

## Class AnimPropertyList

See Also: [Class AnimProperty](#), [Class Animatable](#), [Class Tab](#).

```
class AnimPropertyList : public Tab<AnimProperty*>
```

### Description:

This class is simply a table of anim properties. It has a single method to locate a given id in the list. See [Class Tab](#) for details on how to manipulate the table.

### Methods:

#### Prototype:

```
int FindProperty(DWORD id,int start=0);
```

#### Remarks:

Implemented by the System.

Returns the table index of the property whose id is passed.

#### Parameters:

##### **DWORD id**

The id to find.

##### **int start=0**

A table index that may be used to define an alternate starting point for the search. This is a index into the table of properties.

#### Return Value:

The table index of the property, or if not found, -1.

# Class TrackClipObject

See Also: [Class Animatable](#).

class TrackClipObject

## Description:

If a plug-in supports track view copy/paste operations this object is used. A plug-in should derive a class from this base class to store the data associated with the objects tracks, and implement the methods that identify the creator of the clip object.

## Data Members:

public:

### **Interval clip;**

Specifies the interval of time clipped.

## Methods:

### Prototype:

**TrackClipObject(Interval iv);**

### Remarks:

Constructor.

### Parameters:

#### **Interval iv**

The interval of the time clip.

### Prototype:

**virtual int NumKeys();**

### Remarks:

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Returns the number of keys in the clip object.

### Default Implementation:

**{return 0;}**

**Prototype:**

**virtual BOOL GetKeyVal(int i, void \*val);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Retrieves the value of the 'i-th' key.

**Parameters:**

**int i**

Specifies the key to return.

**void \*val**

The value of the key is stored here.

**Return Value:**

TRUE if the value was retrieved; otherwise FALSE.

**Default Implementation:**

**{return FALSE;}**

**Prototype:**

**virtual BOOL SetKeyVal(int i, void \*val);**

**Remarks:**

This method is available in release 2.0 and later only.

Implemented by the Plug-In.

Sets the value of the 'i-th' key.

**Parameters:**

**int i**

Specifies the key to store.

**void \*val**

The value of the key is passed here.

**Return Value:**

TRUE if the value was stored; otherwise FALSE.

**Default Implementation:**

**{return FALSE;}**

**Prototype:**

**virtual SClass\_ID SuperClassID()=0;**

**Remarks:**

Implemented by the Plug-In.

This method is used to identify the creator of the clip object by returning the SuperClassID of the creator.

**Prototype:**

**virtual Class\_ID ClassID()=0;**

**Remarks:**

Implemented by the Plug-In.

Returns the ClassID of the creator of the clip object.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to delete this instance of the clip object.

# Class TimeMap

See Also: [Class Animatable](#).

class TimeMap

## Description:

This class is used by **Animatable::MapKeys()**. It provides a method **map()** that is used to apply the mapping required by the system to the TimeValue passed.

## Methods:

### Prototype:

**virtual TimeValue map(TimeValue t)=0;**

### Remarks:

Implemented by the System.

This is a time map function that takes a TimeValue and returns a TimeValue. A plug-in developer just calls this map method to alter the TimeValue.

### Parameters:

**TimeValue t**

This is the input time.

### Return Value:

The modified time.

### Prototype:

**virtual TimeValue prevmap(TimeValue t)=0;**

### Remarks:

This method is not used.

## Class TrackHitRecord

See Also: [Class Animatable](#), [Template Class Tab](#).

class TrackHitRecord

### Description:

When hit testing is done, this class has data members used to identify the key that was hit. The manner these are used is up to the developer. For example, the hit member may be used as an index into the table of keys. If a developer needed additional information, the flags could be used. The system does not use these itself - the developer just needs to establish a consistent way to recognize its own keys and tag them as hit.

There is a typedef that defines a table of these **TrackHitRecords**. It is defined as:

```
typedef Tab<TrackHitRecord> TrackHitTab;
```

### Data Members:

public:

```
DWORD hit;
```

```
DWORD flags;
```

### Methods:

### Prototype:

```
TrackHitRecord(DWORD h=0,DWORD f=0)
```

### Remarks:

Constructor. The data members are initialized to the values passed.

## Class ParamDimensionBase

See Also: [Class ParamDimension](#).

### Description:

This class (along with ParamDimension) describes the dimension of a parameter. This dimension can be considered a unit of measure. It describes the parameter's type and order of magnitude.

The dimension type and possibly the dimension scale (if the type is custom) are used to determine a scale factor for the parameter. When a controller is drawing a function curve, it only needs to use the Convert() function - the scale factor is rolled into the single 'vzoom' parameter passed to

**Animatable::PaintFCurves()**. So, for a controller to plot a value 'val' at time t it would do the following:

```
int x = TimeToScreen(t,tzoom,tscroll);  
int y = ValueToScreen(dim-  
>Convert(val),rect.h()-1,vzoom,vscroll);
```

### Methods:

#### Prototype:

```
virtual DimType DimensionType()=0;
```

#### Remarks:

Implemented by the Plug-In.

Returns the dimension type of the parameter. See [List of Dimension Types](#).

#### Prototype:

```
virtual float Convert(float value)=0;
```

#### Remarks:

Implemented by the Plug-In.

When a controller needs to display the parameter values (for example in the function curve editor) it converts the value using this method.

#### Parameters:

**float value**

The value to convert.

**Return Value:**

The converted value.

**Prototype:**

```
virtual float UnConvert(float value)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method is used to un-converted a converted value.

**Parameters:**

**float value**

The value to un-convert.

**Return Value:**

The un-converted value.

## Class AnimEnum

See Also: [Class Animatable](#).

class AnimEnum

### Description:

This class is the callback object for **Animatable::EnumAnimTree()**. This keeps track of the depth of the enumeration.

### Methods:

### Prototype:

```
virtual int proc(Animatable *anim, Animatable *client, int  
subNum)=0;
```

### Remarks:

Implemented by the Plug-In.

This is the method called by **EnumAnimTree()**.

### Parameters:

**Animatable \*anim**

The sub anim.

**Animatable \*client**

The client anim. This is the parent with a sub-anim of **anim**.

**int subNum**

The index of the sub-anim that **anim** is to **client**. For example, if you were to call **client->SubAnim(subNum)** it would return **anim**.

### Return Value:

One of the following values:

**ANIM\_ENUM\_PROCEED**

Continue the enumeration process.

**ANIM\_ENUM\_STOP**

Stop the enumeration process at this level.

**ANIM\_ENUM\_ABORT**

Abort the enumeration processing.

**Prototype:**

**AnimEnum(int s = SCOPE\_OPEN, int deep = 0)**

**Remarks:**

Constructor. Sets default scope and depth if specified.

**Return Value:**

A new AnimEnum object.

**Prototype:**

**void SetScope(int s)**

**Remarks:**

Implemented by the System.

Sets the scope. See below for possible values.

**Parameters:**

**int s**

Specifies the scope to set. See below.

**Prototype:**

**int Scope()**

**Remarks:**

Implemented by the System.

Returns the scope.

**Return Value:**

One or more of the following scope values:

**SCOPE\_DOCLOSED**

Do closed animatables.

**SCOPE\_SUBANIM**

Do the sub anims

**SCOPE\_CHILDREN**

Do the node children

**SCOPE\_OPEN**

Do all open animatables

Equal to **(SCOPE\_SUBANIM|SCOPE\_CHILDREN)**

## **SCOPE\_ALL**

Do all animatables.

Equal to (**SCOPE\_OPEN|SCOPE\_DOCLOSED**)

### **Prototype:**

**void IncDepth()**

### **Remarks:**

Implemented by the System.

Increments the depth count.

### **Prototype:**

**void DecDepth()**

### **Remarks:**

Implemented by the System.

Decrements the depth count.

### **Prototype:**

**int Depth()**

### **Remarks:**

Implemented by the System.

Returns the depth count.

## Class DefNoteTrack

See Also: [Class NoteKeyTab](#), [Class Animatable](#).

```
class DefNoteTrack : public NoteTrack
```

### Description:

This class is available in release 3.0 and later only.

This class is 3ds max's implementation of Note Tracks. It provides implementation for the Animatable methods that let the keys work in Track View. Developers use this class to access the table of keys associated with a track. Methods of class Animatable are available to get access to this class.

### Function:

```
NoteTrack *NewDefaultNoteTrack();
```

### Remarks:

This global function is not part of this class but is available for use. It creates and returns a pointer to a new Note Track.

### Data Members:

public:

```
NoteKeyTab keys;
```

The table of note keys for the track.

### Methods:

The methods of this class are all implemented and used internally.

## List of Schematic View AddAnimatable Flags

See Also: [Class Animatable](#).

These are bit flags which can be passed to **IGraphObjectManager::AddAnimatable(...)** and **Animatable::SvTraverseAnimGraph(...)**.

One or more of the following values:

### **SV\_INITIALLY\_HIDDEN**

If set, newly created node will be in the hidden state. If the node already exists in the graph, the flag is ignored.

### **SV\_DUPLICATE\_INSTANCES**

If set, shared instances of an animatable will produce multiple graph nodes in the schematic view instead of a single shared graph node.

### **SV\_INITIALLY\_CLOSED**

If set, the newly created children of the newly created node will be in the hidden state. If the node already exists in the graph, the flag is ignored. Children of this node that already exist in the graph will not have their visibility state changed.

## Class MultiSelectCallback

See Also: [Class Animatable](#).

class MultiSelectCallback

### Description:

This class is available in release 3.0 and later only.

This is the callback object used to perform the (de)selection via **Animatable::SvGetMultiSelectCallback()**.

Schematic view supports multiple selection. When the user selects a set of objects in the schematic view and then "transfers" that selection set to the rest of max (either by having "synchronize selection" on or by manually moving the selection out), there are a number of ambiguities that can arise. For example, some of the objects in the schematic view cannot be selected in the viewports, material editor, or modifier stack. Another example: the material editor only supports one active material/map but many materials and maps can be selected simultaneously in the schematic view. The "MultiSelectCallback" system exists order to handle these cases and to handle selection synchronization between SV and future editors in 3ds max. When the schematic view attempts to synchronize selection by moving the SV selection set to the "outside" world, it follows this procedure:

1. First SV calls SvGetMultiSelectCallback(...) on all the visible SV nodes to "collect" MultiSelectCallback objects. Objects that want to synchronize their selection state with the schematic view (not a common or trivial operation -- this is really more associated with adding a new editor in 3ds max rather than adding new plugin) return a pointer to a static instance of a MultiSelectCallback derived object. There is only one instance of a MultiSelectCallback per editor.

Furthermore, if an editor displays objects of many classes, all the classes should override SvGetMultiSelectCallback(...) to return the same MultiSelectCallback instance. This implies that, as far as the schematic view is concerned, there is never more than one primary editor class associated with any particular object class (currently, viewports for nodes, material editor for materials and maps and the modifier panel for modifiers).

For example, here is the code in BaseNode that returns the MultiSelectCallback instance for nodes (this is the MultiSelectCallback used for viewports):

```
class BaseNodeMSelCB : public MultiSelectCallback
```

```

{
private:
bool clear;
BaseNodeTab selNodeTab;
BaseNodeTab deselNodeTab;

public:
int Priority() { return 1000; }
void Begin(IGraphObjectManager *gom, bool clear);
void Select(IGraphObjectManager *gom, IGraphNode *gNode,
bool isSelected);
void End(IGraphObjectManager *gom);
};
static BaseNodeMSELCB baseNodeMSELCB;
MultiSelectCallback*
BaseNode::SvGetMultiSelectCallback(IGraphObjectManager
*gom, IGraphNode *gNode)
{
return &baseNodeMSELCB;
}

```

2. For each selection class (unique MultiSelectCallback instance), the schematic views calls "Begin(...)". This is the spot where any "pre-selection" preparation takes place. The order that the MultiSelectCallback instances are called in is determined by their priority. The priority is returned by the "Priority()" method. MultiSelectCallback's with a higher priority (lower value) are called before those with a lower priority (higher value). For example, here is the Begin associated with the viewports:

```

void BaseNodeMSELCB::Begin(IGraphObjectManager *gom, bool
clear)
{
this->clear = clear;
//
// If the "clear" bool is true, the current viewport selection set is
cleared...

```

```

//
if (clear)
  GetActiveSelSet()->Clear(FALSE);
//
// Some housekeeping in preparation for the select...
//
selNodeTab.Resize(0);
deselNodeTab.Resize(0);
}

```

3. For each of objects in the schematic view whose selection state is changing, the object's MultiSelectCallback instance is retrieved (again) and the "Select" method is called. Here is where the actual selection/deselection work can take place. I say "can" because, in practice, the select method usually just collects all the objects to be selected and all the objects to be deselected into lists which are then processed in the "End(...)" method. This is simply for performance -- it is often more efficient to set the selection state of a group of objects all at once. Here's the "Select(...)" method from BaseNode:

```

void Select(IGraphObjectManager *gom, IGraphNode *gNode, bool
isSelected)
{
  BaseNode *baseNode = (BaseNode *) gNode->GetAnim();

  if (isSelected)
  {
    if (!baseNode->IsRootNode() && !baseNode->IsFrozen() &&
!baseNode->IsHidden())
      selNodeTab.AppendNode(baseNode, FALSE);
    }
  else
  {
    if (baseNode->Selected())
      deselNodeTab.AppendNode(baseNode, FALSE);
    }
  }

```

4. Finally, for each selection class (unique MultiSelectCallback instance), the schematic views calls "End(...)". This is the spot where any "post-selection" operations take place. For example, here is the "End(...)" for the BaseNode (viewports):

```
void End(IGraphObjectManager *gom)  
{  
  if (selNodeTab.Count() > 0 || deselNodeTab.Count() > 0)  
  {  
    theHold.Begin();  
    if (selNodeTab.Count() > 0)  
      GetActiveSelSet()->SelectMultiple(selNodeTab, FALSE);  
  
    if (deselNodeTab.Count() > 0)  
      GetActiveSelSet()->DeselectMultiple(deselNodeTab, FALSE);  
  
    theHold.Accept(getResMgr().getString(IDS_SV_SELECT,  
appInst));  
    RedrawViewports(GetCurTime(), VP_DONT_SIMPLIFY);  
  }  
  else  
  {  
    if (clear)  
      RedrawViewports(GetCurTime(), VP_DONT_SIMPLIFY);  
  }  
}
```

**Methods:**

public:

**Prototype:**

**virtual int Priority()=0;**

**Remarks:**

Returns the priority of the callback. MultiSelectCallback's with a higher priority (lower value) are called before those with a lower priority (higher value).

**Prototype:**

**virtual void Begin(IGraphObjectManager \*gom, bool clear)=0;**

**Remarks:**

Called to begin the multi-selection process. This is the spot where any "pre-selection" operations take place.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**bool clear**

true to clear the previous selection; false to leave intact.

**Prototype:**

**virtual void Select(IGraphObjectManager \*gom, IGraphNode \*gNode, bool isSelected)=0;**

**Remarks:**

This method selects or deselects the node passed.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

**IGraphNode \*gNode**

Points to the node in schematic view.

**bool isSelected**

true if select; false if deselect.

**Prototype:**

**virtual void End(IGraphObjectManager \*gom)=0;**

**Remarks:**

Called when done. This is the spot where any "post-selection" operations take place.

**Parameters:**

**IGraphObjectManager \*gom**

Points to the schematic view window manager.

## Class ICustAttribContainer

See Also: [Class CustAttrib](#) , [Class RemapDir](#) , [Class ReferenceTarget](#)

class ICustAttribContainer : public ReferenceTarget

### Description:

This class is available in release 4.0 and later only.

This class represents the interface class to a custom attributes container.

### Methods:

public:

### Prototype:

**virtual int GetNumCustAttribs()=0;**

### Remarks:

This method returns the number of custom attributes.

### Prototype:

**virtual CustAttrib \*GetCustAttrib(int i)=0;**

### Remarks:

This method allows you to retrieve the custom attribute by its specified index.

### Parameters:

**int i**

The index of the custom attribute you wish to obtain.

### Prototype:

**virtual void AppendCustAttrib(CustAttrib \*attribute)=0;**

### Remarks:

This method allows you to append a custom attribute.

### Parameters:

**CustAttrib \*attribute**

A pointer to the custom attribute you wish to add.

**Prototype:**

**virtual void SetCustAttrib(int i, CustAttrib \*attribute)=0;**

**Remarks:**

This method allows you to set the custom attribute at the specified index.

**Parameters:**

**int i**

The index for which to set the custom attribute.

**CustAttrib \*attribute**

A pointer to the custom attribute you wish to set.

**Prototype:**

**virtual void InsertCustAttrib(int i, CustAttrib \*attribute)=0;**

**Remarks:**

This method allows you to insert a custom attribute at the specified index.

**Parameters:**

**int i**

The index at which to insert the custom attribute.

**CustAttrib \*attribute**

A pointer to the custom attribute you wish to insert.

**Return Value:**

**Prototype:**

**virtual void RemoveCustAttrib(int i)=0;**

**Remarks:**

This method allows you to remove a custom attribute.

**Parameters:**

**int i**

The index of the custom attribute to remove.

**Prototype:**

**virtual ParamDlg\* CreateParamDlg(HWND hwMtlEdit, IMtlParams \*imp)=0;**

**Remarks:**

This method gets called when the material or texture is to be displayed in the material editor parameters area. The plug-in should allocate a new instance of a class derived from ParamDlg to manage the user interface.

**Parameters:**

**HWND hwMtlEdit**

The window handle of the materials editor.

**IMtlParams \*imp**

The interface pointer for calling methods in 3ds max.

**Return Value:**

A pointer to the created instance of a class derived from **ParamDlg**.

**Prototype:**

**virtual void CopyParametersFrom(ReferenceMaker \*from, RemapDir &remap)=0;**

**Remarks:**

This method will copy the parameters from a specified reference maker.

**Parameters:**

**ReferenceMaker \*from**

A pointer to the reference maker to copy the parameters from.

**RemapDir &remap**

This class is used for remapping references during a Clone. See [Class RemapDir](#).

**Prototype:**

**virtual Animatable \*GetOwner()=0;**

**Remarks:**

This method returns a pointer to the owner of the custom attributes.

**Prototype:**

```
virtual void DeleteThis()=0;
```

**Remarks:**

Self deletion.

## Class NotifyCollapseEnumProc

See Also: [Class GeomPipelineEnumProc](#), [Class BaseObject](#), [Class Object](#), [Class IDerivedObject](#).

```
class NotifyCollapseEnumProc : public GeomPipelineEnumProc
```

### Description:

This class is available in release 4.0 and later only.

Whenever the modifier stack is collapsed the code has to notify all objects in the stack with a Pre and a Post notification. In order to do this, this class can be used in conjunction with the method **EnumGeomPipeLine()**. In the constructor one can specify, if it is a pre- or post- collapse notification. In case it is a post collapse the object that represents the result of the collapse has to be provided as well. The INode pointer to the beginning of the pipeline that was collapsed has to be provided in both cases.

All methods of this class are implemented by the System.

### Methods:

public:

### Prototype:

```
NotifyCollapseEnumProc(bool preCollapse, INode *n, Object  
*collapsedObj = NULL) : bPreCollapse(preCollapse), node(n),  
collapsedObject(collapsedObj);
```

### Remarks:

Constructor. The private data members are initialized by the passed parameters.

### Parameters:

#### **bool preCollapse**

Indicates if this is a pre- collapse or a post- collapse. Pass true for pre and false for post.

#### **INode \*n**

Points to the node at the beginning of the pipeline that was collapsed.

#### **Object \*collapsedObj = NULL**

If this is a post- collapse then points to the object which is the result of the collapse.

**Sample Code:**

The following code fragment shows using this constructor and then the EnumGeomPipeline with this object:

```
NotifyCollapseEnumProc PreNCEP(true,node);  
EnumGeomPipeline(&PreNCEP,node);
```

**Prototype:**

```
virtual PipeEnumResult proc(ReferenceTarget  
*object, IDerivedObject *derObj, int index);
```

**Remarks:**

This is the implementation of the EnumGeomPipeline callback method proc() which calls BaseObject::NotifyPreCollapse or BaseObject::NotifyPostCollapse as required.

## Transform Lock Types

See Also: [Class INode](#).

One of the following values:

**INODE\_LOCKPOS**

Position locked.

**INODE\_LOCKROT**

Rotate locked.

**INODE\_LOCKSCL**

Scale locked.

## Transform Lock Axis

See Also: [Class INode](#).

One of the following values:

**INODE\_LOCK\_X**

X axis locked.

**INODE\_LOCK\_Y**

Y axis locked.

**INODE\_LOCK\_Z**

Z axis locked.

## Class RenderData

See Also: [Class INode-Access To Render Data](#), [Class LightDesc](#).

```
class RenderData : public InterfaceServer
```

### Description:

This class has a single virtual method to delete the class. For example, when **LightDesc** objects are deleted this is the method to do so.

### Methods:

#### Prototype:

```
virtual void DeleteThis();
```

#### Remarks:

Implemented by the Plug-In.

This method is used to delete an instance of the class.

#### Default Implementation:

```
{delete this; }
```

#### Prototype:

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0);
```

#### Remarks:

This method is available in release 3.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

#### Parameters:

**int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value (defined uniquely for each **cmd**).

**Default Implementation:**

```
{ return 0; }
```

## List of XRef Flag Bits

See Also: [Class INode](#).

The following flags are used by the methods **INode::SetXRefFlags** and **GetXRefFlags**.

### **XREF\_UPDATE\_AUTO**

Indicates automatic XRef file updating is on.

### **XREF\_BOX\_DISP**

Indicates the Box display option is set.

### **XREF\_HIDDEN**

Indicates the XRef is hidden.

### **XREF\_DISABLED**

Indicates the XRef is disabled.

### **XREF\_IGNORE\_LIGHTS**

Indicates the XRef will ignore the lights in the file.

### **XREF\_IGNORE\_CAMERAS**

Indicates the XRef will ignore the cameras in the file.

### **XREF\_IGNORE\_SHAPES**

Indicates the XRef will ignore the shapes in the file.

### **XREF\_IGNORE\_HELPERS**

Indicates the XRef will ignore the helpers in the file.

### **XREF\_IGNORE\_ANIM**

Indicates the XRef will ignore the animation in the file.

### **XREF\_FILE\_CHANGE**

This bit is set when a change notification is sent indicating that the file may have changed. We don't know for sure if the file actually changed but the ref should be reloaded.

### **XREF\_LOAD\_ERROR**

This bit will be set when a ref can't be resolved.

## Class ISubMap

See Also: [Class MtlBase](#), [Class Texmap](#), [Class ParamDlg](#), [Class ReferenceTarget](#).

class ISubMap

### Description:

This class is available in release 4.0 and later only.

The methods of this class provide access to the sub-textures of a MtlBase. These properties include the number of sub-maps, the slot type, on / off state, etc.

The MtlBase class sub-classes from this class. If a developer is creating a plug-in derived from MtlBase (for instance a Material or Texture) then implementations of these methods are required. Developers may call these methods on an existing MtlBase sub-class.

### Methods:

public:

### Prototype:

**virtual int NumSubTexmaps()**

### Remarks:

Implemented by the Plug-In.

Returns the number of sub-textures managed by this material or texture.

### Default Implementation:

**{ return 0; }**

### Prototype:

**virtual Texmap\* GetSubTexmap(int i)**

### Remarks:

Implemented by the Plug-In.

Returns a pointer to the 'i-th' sub-texmap managed by the material or texture.

Note: For the 3ds max Standard material, the sub-texmap index used with this method is shown in [List of Texture Map Indices](#).

### Parameters:

**int i**

Specifies the texmap to return.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual int MapSlotType(int i)
```

**Remarks:**

Implemented by the Plug-In.

In the Coordinates rollup in the user interface for a texture map are two options. These options are Texture or Environment. The slot type is one of these two options, texture coordinates or environment coordinates. There are a variety of texture coordinate types. There are the type assigned to the object and the environment type (Spherical, Cylindrical, Shrink-wrap, Screen). This method is used to determine the type required by the particular sub-texture. This is either texture coordinates (**MAPSLOT\_TEXTURE**) or environment coordinates (**MAPSLOT\_ENVIRON**).

**Parameters:**

**int i**

The index of the sub-texture whose slot type to return.

**Return Value:**

See [List of Map Slot Types](#).

**Default Implementation:**

```
{ return MAPSLOT_TEXTURE; }
```

**Prototype:**

```
virtual void SetSubTexmap(int i, Texmap *m);
```

**Remarks:**

Implemented by the Plug-In.

Stores the 'i-th' sub-texmap managed by the material or texture.

Note: For the 3ds max Standard material, the sub-texmap index used with this method is shown in [List of Texture Map Indices](#).

**Parameters:**

**int i**

The index of the storage for the texmap.

**Texmap \*m**

The texmap to store.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int SubTexmapOn(int i)
```

**Remarks:**

Implemented by the Plug-In.

Returns nonzero if the specified sub-texmap is on; otherwise zero. Some materials may have user interface controls to toggle the sub-maps on and off. The Standard material has such controls for example.

**Parameters:**

**int i**

The index of the sub-texmap to check.

**Default Implementation:**

```
{ return 1; }
```

**Prototype:**

```
virtual TSTR GetSubTexmapSlotName(int i);
```

**Remarks:**

Implemented by the Plug-In.

This method returns the slot name of the 'i-th' sub-texmap. This name appears in the materials editor dialog. For instance, if you are in a material and then you go down into a map, this is the name that appears just below the 'Get Material' icon. For example, in the Standard material when you choose one of the maps, the map name appears to let you know which slot you are working on. For the Standard material, this may be "Ambient", "Diffuse", "Specular", etc.

**Parameters:**

**int i**

Specifies the slot whose name is returned.

**Default Implementation:**

The default implementation returns an empty ("" ) string.

**Prototype:**

**TSTR GetSubTexmapTVName(int i);**

**Remarks:**

Implemented by the Plug-In.

Returns the name to appear in Track View of the 'i-th' sub-texmap.

**Parameters:**

**int i**

Specifies the sub-texmap whose name is returned.

**Prototype:**

**virtual BOOL SetDlgThing(ParamDlg\* dlg);**

**Remarks:**

This method is called by the ParamMap2 **AUTO\_UI** system if the material/texmap is letting the system build an **AutoMParamDlg** for it. This method is called on a material/texmap coming into an existing set of ParamDlgs, once for each secondary **ParamDlg** and it should set the appropriate 'thing' into the given dlg (the 'thing' being, for example, a **Texout\*** or **UVGen\***). Return **FALSE** if **dlg** is unrecognized.

Note: See the discussion above in **CreateParamDlg()** for additional details on this method.

**Parameters:**

**ParamDlg\* dlg**

Points to the ParamDlg to check. See [Class ParamDlg](#).

**Return Value:**

**TRUE** if the **dlg** passed is recognized; **FALSE** if unrecognized.

**Default Implementation:**

```
{ return FALSE; }
```

**Sample Code:**

```

BOOL Gradient::SetDlgThing(ParamDlg* dlg)
{
    if (dlg == uvGenDlg)
        uvGenDlg->SetThing(uvGen);
    else if (dlg == texoutDlg)
        texoutDlg->SetThing(texout);
    else
        return FALSE;
    return TRUE;
}

```

**Prototype:**

```

void CopySubTexmap(HWND hwnd, int ifrom, int ito);

```

**Remarks:**

Implemented by the System.

This method is used to handle the drag-and-drop of texmaps. A developer implements the logic to handle the drag and drop. Once they have the information about what slot was dropped on what other slot, this method may be called to handle the copying. This is used by the 3ds max Standard material.

**Parameters:**

**HWND hwnd**

The rollup page window handle.

**int ifrom**

The source texmap.

**int ito**

The destination texmap.

**Prototype:**

```

virtual ReferenceTarget *GetRefTarget()=0;

```

**Remarks:**

Implemented by the System.

The implementation of this method is provided by **MtlBase**. It returns its **this** pointer.

## Class RenderMapsContext

See Also: [Class INode](#), [Class ViewParams](#), [Structure SubRendParams](#), [Class Matrix3](#), [Class Box3](#), [Class Point4](#), [Class Bitmap](#), [Class RenderGlobalContext](#).

class RenderMapsContext

### Description:

An instance of this class is passed into the **MtlBase::BuildMaps()** method. This is used for the Mirror and Automatic Cubic maps. These maps callback to methods of this class to perform a rendering from a particular view. Sample code using these methods is available in **\MAXSDK\SAMPLES\MATERIALS\MIRROR.CPP** and **ACUBIC.CPP**. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

```
virtual INode *GetNode()=0;
```

#### Remarks:

Returns the INode pointer of the node being rendered. This pointer allows a developer to access the properties of the node. See [Class INode](#).

#### Prototype:

```
virtual int NodeRenderID()=0;
```

#### Remarks:

Returns the node ID for the item being rendered or -1 if not set. This ID is assigned when the scene is being rendered - each node is simply given an ID: 0, 1, 2, 3, etc. The **NodeRenderID()** is simply a number automatically assigned to every node being rendered so that they can be differentiated in texture maps such as the Auto-cubic, which needs to store a cubic map for each node it is applied to.

#### Prototype:

```
virtual void GetCurrentViewParams(ViewParams &vp)=0;
```

#### Remarks:

Retrieves the current view dependent parameters.

**Parameters:**

**ViewParams &vp**

The **ViewParams** instance to update.

**Prototype:**

**virtual void GetSubRendParams(SubRendParams &srp)=0;**

**Remarks:**

Retrieves the sub-render parameters.

**Parameters:**

**SubRendParams &srp**

The **SubRendParams** instance to update.

**Prototype:**

**virtual int SubMtlIndex()=0;**

**Remarks:**

Returns the current sub material index or -1 if at node material level.

**Prototype:**

**virtual void FindMtlPlane(float pl[4])=0;**

**Remarks:**

Computes the plane containing the current material or sub material. This is used by the Flat Mirror material.

**Parameters:**

**float pl[4]**

The plane containing the current material or sub material. The four float values in **pl[4]** represent the plane equation. If you call the four values A,B,C and D, then the plane equation of the plane is  $Ax + By + cZ + D = 0$ .

**Prototype:**

**virtual void FindMtlScreenBox(Rect &sbox, Matrix3\* viewTM=NULL, int mtlIndex=-1)=0;**

**Remarks:**

This method computes the rectangle in screen space of the specified material. This uses the viewTM, but assumes the remaining view params (devWidth, devHeight, devAspect, fov) are the same as the main render. This method is specific to the Flat Mirror material.

**Parameters:****Rect & sbox**

The resulting 2D box.

**Matrix3\* viewTM=NULL**

The view matrix.

**int mtlIndex=-1**

The material index, or -1 if a node level material.

**Prototype:**

**virtual Box3 CameraSpaceBoundingBox()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method computes the bounding box in camera space of the object associated with the reflection or refraction map being built in a call to a map's **BuildMaps()** method.

**Prototype:**

**virtual Box3 ObjectSpaceBoundingBox()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method computes the bounding box in object space of the object associated with the reflection or refraction map being built in a call to a map's **BuildMaps()** method.

**Prototype:**

**virtual Matrix3 ObjectToWorldTM()=0;**

**Remarks:**

This method is available in release 2.0 and later only.

This method returns the object to world transformation.

**Prototype:**

**virtual RenderGlobalContext \*GetGlobalContext();**

**Remarks:**

This method is available in release 2.0 and later only.

Returns a pointer to a class that describes properties of the rendering environment.

**Prototype:**

**virtual int Render(Bitmap \*bm, ViewParams &vp,  
SubRendParams &srp, Point4 \*clipPlanes=NULL, int  
nClipPlanes=0)=0;**

**Remarks:**

Renders the scene and stores in the result into **bm**.

**Parameters:**

**Bitmap \*bm**

The Bitmap to render the result to. The properties of this bitmap define the properties of the render (such as the width and height).

**ViewParams &vp**

The ViewParams.

**SubRendParams &srp**

The SubRendParams.

**Point4 \*clipPlanes=NULL**

This revised parameter is available in release 2.0 and later only.

This is a pointer to an array of Point4s, each of which represents a clip plane. If it is non-null, the renderer will clip all objects against these planes in addition to the normal left/right/top/bottom clipping. This is used by the Mirror material to clip away stuff that is behind the mirror. If not needed this may default to NULL.

**int nClipPlanes=0**

This parameter is available in release 2.0 and later only.

The number of clipping planes above. A maximum of 6 is possible.

**Return Value:**

Nonzero on success; otherwise zero.

**Prototype:**

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0);
```

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:****int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

# Class PStamp

See Also: [Class AnimProperty](#).

class PStamp: public AnimProperty

## Description:

This class is available in release 2.0 and later only.

This is used internally by the Material / Map Browser which supports the display of small and large icon images for material and texture maps. This class is the postage stamp image object.

The width in bytes of the image pixel array is given by the following macro, where w is pixel width.

```
#define ByteWidth(w) (((w*3+3)/4)*4)
```

## Methods:

### Prototype:

```
virtual int Width()=0;
```

### Remarks:

Returns the width of the image in pixels.

### Prototype:

```
virtual int Height()=0;
```

### Remarks:

Returns the height of the image in pixels.

### Prototype:

```
virtual void SetImage(UBYTE *img)=0;
```

### Remarks:

Sets the image for the postage stamp.

### Parameters:

```
UBYTE *img
```

This is an array of RGB triplets.

**Prototype:**

**virtual void GetImage(UBYTE \*img)=0;**

**Remarks:**

Retrieves the image bytes of the postage stamp.

**Parameters:**

**UBYTE \*img**

This is an array of RGB triplets.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

This method is called to delete this instance of the class.

**Prototype:**

**virtual IOResult Load(ILoad \*iload)=0;**

**Remarks:**

This method is used to load the postage stamp image.

**Prototype:**

**virtual IOResult Save(ISave \*isave)=0;**

**Remarks:**

This method is used to save the postage stamp image.

## List of Material Flags

See Also: [Class MtlBase](#).

One or more of the following values:

### **MTL\_IN\_SCENE**

The material is being used in the scene.

### **MTL\_BEING\_EDITED**

The material's parameters are being displayed in the Material Editor.

### **MTL\_SUB\_BEING\_EDITED**

This material OR sub-material texmap is being displayed in the Material Editor.

### **MTL\_TEX\_DISPLAY\_ENABLED**

Interactive texture display is enabled.

### **MTL\_MEDIT\_BACKGROUND**

The material has the background shown in Material Editor.

### **MTL\_MEDIT\_BACKLIGHT**

The material is backlight in Material Editor.

### **MTL\_MEDIT\_OBJTYPE**

The object type displayed in Material Editor.

### **MTL\_MEDIT\_TILING**

The tiling setting of the Material Editor.

### **MTL\_MEDIT\_VIDCHECK**

The video check state of the Material Editor.

### **MTL\_BROWSE\_OPEN1**

This is for internal use only.

### **MTL\_BROWSE\_OPEN2**

This is for internal use only.

## Class IAutoMParamDlg

See Also: [Class ParamDlg](#), [Class IParamMap2](#), [Class IMtlParams](#), [Class ParamMap2UserDlgProc](#).

```
class IAutoMParamDlg : public ParamDlg
```

### Description:

This class is available in release 3.0 and later only.

This is the Auto ParamDlg class for Material Editor auto-UI, instanced by **ClassDesc2::CreateParamDlg()**. It maintains a table of secondary ParamDlgs for master ParamDlgs (eg, the one returned from CreateParamDlg()) and will broadcast appropriate method calls to them as the master receives them.

### Methods:

public:

### Prototype:

```
virtual void InvalidateUI()=0;
```

### Remarks:

This method causes the user interface controls to be re-drawn.

### Prototype:

```
virtual void MtlChanged()=0;
```

### Remarks:

This method may be called to causes the viewports to be redrawn. It should be called when any parameter that affects the look of the material in the viewport has been altered. If the material is not on a visible node in a shaded view, nothing will happen. This method should **not** be called as a spinner is being dragged, but only upon release of the mouse button.

### Prototype:

```
virtual int NumDlgs()=0;
```

### Remarks:

Returns the number of secondary dialogs.

**Prototype:**

**virtual void AddDlg(ParamDlg\* dlg)=0;**

**Remarks:**

Adds the specified dialog as another secondary dialog.

**Parameters:**

**ParamDlg\* dlg**

Points to the parameter dialog to add.

**Prototype:**

**virtual ParamDlg\* GetDlg(int i)=0;**

**Remarks:**

Returns a pointer to the 'i-th' secondary dialog.

**Parameters:**

**int i**

The zero based index of the dialog to return.

**Prototype:**

**virtual void SetDlg(int i, ParamDlg\* dlg)=0;**

**Remarks:**

Sets the 'i-th' dialog to the one passed.

**Parameters:**

**int i**

The zero based index of the dialog to set.

**ParamDlg\* dlg**

Points to the parameter dialog to set.

**Prototype:**

**virtual void DeleteDlg(ParamDlg\* dlg)=0;**

**Remarks:**

This method is used for deleting secondary dialogs from a master **IAutoMParamDlg**. Use this along with **AddDlg()** if you are dynamically changing the set of rollups for the plugin, so that the **P\_AUTO\_UI** system

can correctly manage all current secondary rollups.

**Parameters:**

**ParamDlg\* dlg**

Points to the ParamDlg to delete.

**Prototype:**

**virtual IParamMap2\* GetMap()=0;**

**Remarks:**

Returns a pointer to the parameter map2 of this primary (master) dialog.

**This function is not part of this class but is available for use.**

**Function:**

**IAutoMParamDlg\* CreateAutoMParamDlg(HWND hMedit, IMtlParams \*i, MtlBase\* mtl, IParamBlock2\* pb, ClassDesc2\* cd, HINSTANCE inst, TCHAR\* dlgTemplate, TCHAR\* title, int rollFlags, ParamMap2UserDlgProc\* dlgProc=NULL);**

**Remarks:**

This function may be called to create a parameter map2 for use in the material editor dialog.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **CreateAutoMParamDlg()** with default map ID of 0.

**Parameters:**

**HWND hMedit**

The window handle of the materials editor.

**IMtlParams \*i**

The interface pointer for materials. See [Class IMtlParams](#).

**MtlBase\* mtl**

Points to the material calling this function.

**IParamBlock2\* pb**

Points to the parameter block instance associated with the parameter map.

**ClassDesc2\* cd**

The class descriptor2 for the plug-in creating the parameter map.

**HINSTANCE inst**

The plug-ins DLL instance handle.

**TCHAR\* dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR\* title**

The title displayed in the rollup page title bar.

**int rollFlags**

A set of flags to control settings of the rollup page.

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**HWND hOldRollup=NULL**

An optional window handle to allow supplying an existing rollup that will be **replaced** by the newly created one.

**Return Value:**

A pointer to an interface for managing the parameter map2.

**Function:**

```
IAutoMParamDlg* CreateAutoMParamDlg(MapID map_id,  
HWND hMedit, IMtlParams *i, MtlBase* mtl, IParamBlock2* pb,  
ClassDesc2* cd, HINSTANCE inst, TCHAR* dlgTemplate,  
TCHAR* title, int rollFlags, ParamMap2UserDlgProc*  
dlgProc=NULL);
```

**Remarks:**

This function is available in release 4.0 and later only.

This function may be called to create a parameter map2 for use in the material editor dialog. This overload of **CreateAutoMParamDlg()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

**Function:**

**IAutoEParamDlg\* CreateAutoEParamDlg(IRendParams \*i, Effect\* e, IParamBlock2\* pb, ClassDesc2\* cd, HINSTANCE inst, TCHAR\* dlgTemplate, TCHAR\* title, int rollFlags, ParamMap2UserDlgProc\* dlgProc=NULL);**

**Remarks:**

This function creates an AutoEParamDlg for render effects.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **CreateAutoEParamDlg()** with default map ID of 0.

**Parameters:**

**IRendParams \*i**

An interface pointer for rendering effects.

**Effect\* e**

Points to the rendering effect calling this function.

**IParamBlock2\* pb**

Points to the parameter block instance associated with the parameter map.

**ClassDesc2\* cd**

The class descriptor2 for the plug-in creating the parameter map.

**HINSTANCE inst**

The plug-ins DLL instance handle.

**TCHAR\* dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR\* title**

The title displayed in the rollup page title bar.

**int rollFlags**

A set of flags to control settings of the rollup page.

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**Return Value:**

A pointer to an interface for managing the parameter map2.

**Function:**

```
IAutoEParamDlg* CreateAutoEParamDlg(MapID map_id,  
IRendParams *i, Effect* e, IParamBlock2* pb, ClassDesc2* cd,  
HINSTANCE inst, TCHAR* dlgTemplate, TCHAR* title, int  
rollFlags, ParamMap2UserDlgProc* dlgProc=NULL);
```

**Remarks:**

This function is available in release 4.0 and later only.

This function creates an AutoEParamDlg for render effects. This overload of **CreateAutoEParamDlg()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to be created for this particular parameter block. See original function for the rest of the description.

## Class MtlMakerCallback

See Also: [Class TexHandleMaker](#), [Class TextureInfo](#), [Class MtlBase](#), [Class Texmap](#), [Class Material](#).

class MtlMakerCallback: public TexHandleMaker

### Description:

This class is available in release 4.0 and later only.

This is the callback used with the new multi-texture interface provided by method **MtlBase::SetupGfxMultiMaps()**.

### Methods:

public:

### Prototype:

```
virtual void GetGfxTexInfoFromTexmap(TimeValue t,  
TextureInfo& texinf, Texmap *txm)=0;
```

### Remarks:

This method updates all the fields of the TextureInfo instance except the texture handle and the texture ops.

### Parameters:

#### **TimeValue t**

The time at which the texture is evaluated.

#### **TextureInfo& texinf**

The texture info which is updated.

#### **Texmap \*txm**

Points to the texmap as the source for the texture info update.

### Prototype:

```
virtual BOOL NumberTexturesSupported()=0;
```

### Remarks:

This method returns the number of textures that the hardware+driver supports.

## List of IOResults

One of the following values:

### **IO\_OK**

The result was acceptable - no errors.

### **IO\_END**

This is returned from **ILoad::OpenChunk()** when the end of the chunks at a certain level have been reached. It is used as a signal to terminate the processing of chunks at that level.

### **IO\_ERROR**

This is returned if an error occurred. Note that the plug-in should not put up a message box if a read error occurred. It should simply return the error status. This prevents a excess of messages from appearing.

## Class RefList

See Also: [Class RefListItem](#).

class RefList

### Description:

The method **GetRefList()** returns a list of references to a reference target. This class is linked list of references. All methods of this class are implemented by the system.

### Data Members:

public:

**RefListItem\* first;**

### Methods:

#### Prototype:

**RefList()**

#### Remarks:

Constructor. The list is set to NULL.

#### Prototype:

**RefListItem\* FirstItem()**

#### Remarks:

Returns the first item in the list.

#### Prototype:

**RefResult DeleteItem(RefMakerHandle hmaker, int eval);**

#### Remarks:

Deletes the specified item from the list.

#### Parameters:

**RefMakerHandle hmaker**

The item to delete.

**int eval**

If nonzero then when inside of **NotifyDependents()**, just set **maker** to

NULL.

**Return Value:**

If the item was deleted **REF\_SUCCEED** is returned; otherwise **REF\_INVALID** is returned.

**Prototype:**

**RefResult AddItem(RefMakerHandle hmaker);**

**Remarks:**

Adds the specified item to the list.

**Parameters:**

**RefMakerHandle hmaker**

The item to add.

**Return Value:**

Returns **REF\_SUCCEED**.

**Prototype:**

**void Cleanup();**

**Remarks:**

This method removes null entries from the list.

## Class DependentEnumProc

See Also: [Class ReferenceTarget](#).

class DependentEnumProc

### Description:

This class is a callback object for the

**ReferenceTarget::EnumDependents()** method. The **proc()** method is called by the system.

### Methods:

### Prototype:

```
virtual int proc(ReferenceMaker *rmaker)=0;
```

### Remarks:

Implemented by the Plug-In.

This is the method called by system from

**ReferenceTarget::EnumDependents()**.

### Parameters:

**ReferenceMaker \*rmaker**

A pointer to the reference maker

### Return Value:

One of the following values:

**DEP\_ENUM\_CONTINUE**

This continues the enumeration.

**DEP\_ENUM\_HALT**

The stops the enumeration.

**DEP\_ENUM\_SKIP**

This option is available in release 3.0 and later only.

Reference Targets can have multiple Reference Makers (dependents). In certain instances when EnumDependents() is used to enumerate them you may not want to travel up all of the "branches". By returning DEP\_ENUM\_SKIP from this method you tell the enumerator to not enumerate the current Reference Maker's dependents but not to halt the enumeration completely.

## List of Snap Flags

See [Structure SnapInfo](#).

One or more of the following values:

### **SNAP\_IN\_3D**

Snap to all points.

### **SNAP\_IN\_PLANE**

Snap only to points on the construction (or optionally specified) plane.

### **SNAP\_UNSEL\_OBJS\_ONLY**

Ignore selected nodes when considering snap points.

### **SNAP\_SEL\_OBJS\_ONLY**

Ignore unselected nodes when considering snap points.

### **SNAP\_UNSEL\_SUBOBJ\_ONLY**

Ignore selected sub-object geometry when considering snap points.

### **SNAP\_SEL\_SUBOBJ\_ONLY**

Ignore unselected sub-object geometry when considering snap points.

### **SNAP\_FORCE\_3D\_RESULT**

Override user settings to force snap in 3D.

### **SNAP\_OFF\_PLANE**

This is used internally to snap only to points off the plane.

Many of the objects call **GetCPDisp()** when they want to snap lengths etc.

This method knew nothing about the snapping that was added in 3ds max 2.0

so it had to be retrofited to call **SnapPoint()**. This is where

**SNAP\_OFF\_PLANE** is called. It forces the osnapmanager to only consider points which are OFF the current construction plane.

### **SNAP\_TRANSPARENTLY**

This is used internally to suppresses any display in the viewports. This can be used to provide a way of snapping to arbitray screen points without giving feedback.

### **SNAP\_APPLY\_CONSTRAINTS**

This is used internally to suppresses any display in the viewports.

### **SNAP\_PROJ\_XAXIS**

This is used internally to suppresses any display in the viewports.

**SNAP\_PROJ\_YAXIS**

This is used internally to suppresses any display in the viewports.

**SNAP\_PROJ\_ZAXIS**

This is used internally to suppresses any display in the viewports.

## Class HitData

See Also: [Class HitRecord](#).

class HitData

### Description:

This class is used during sub-object hit testing to identify sub-object components. If the 4 bytes available in the **info** data member of **HitRecord** are insufficient to identify the sub-object component an instance of this class can be created to contain the necessary data.

### Methods:

### Prototype:

```
virtual ~HitData();
```

### Remarks:

Implemented by the Plug-In.

The virtual destructor allows the instance of **HitData** to be deleted when the **HitRecord** that points to it is deleted.

## List of Mouse Callback Flags

One or more of the following values describing the state of the Shift/Ctrl/Alt keys and mouse buttons:

### **MOUSE\_SHIFT**

Indicates the Shift key is pressed.

### **MOUSE\_CTRL**

Indicates the Ctrl key is pressed.

### **MOUSE\_ALT**

Indicates the Alt key is pressed.

### **MOUSE\_LBUTTON**

Indicates the Left button is down.

### **MOUSE\_MBUTTON**

Indicates the Middle button is down.

### **MOUSE\_RBUTTON**

Indicates the Right button is down.

## Class CtrlHitLog

class CtrlHitLog

### Description:

This class provides a data structure for keeping a log of hits during controller gizmo hit-testing. It provides a list of **CtrlHitRecords** that may be added to and cleared. A developer may also request the 'closest' hit record in the list. All methods are implemented by the system.

### Methods:

#### Prototype:

**CtrlHitLog()**

#### Remarks:

Constructor.

#### Prototype:

**void Clear();**

#### Remarks:

Clears the list of hits.

#### Prototype:

**CtrlHitRecord\* First()**

#### Remarks:

Returns the first hit record in the list.

#### Prototype:

**CtrlHitRecord\* ClosestHit();**

#### Remarks:

Returns the **CtrlHitRecord** that was 'closest' to the mouse position when hit testing was performed.

#### Prototype:

**void LogHit(INode \*nr,DWORD dist,ulong info,DWORD infoExtra);**

**Remarks:**

This method is called to log a hit. It creates a new **CtrlHitRecord** object using the data passed and adds it to the hit log.

**Parameters:**

**INode \*nr**

The node whose gizmo was hit.

**DWORD dist**

The 'distance' of the hit. What the distance actually represents depends on the rendering level of the viewport. For wireframe modes, it refers to the distance in the screen XY plane from the mouse to the sub-object component. In a shaded mode, it refers to the Z depth of the sub-object component. In both cases, smaller values indicate that the sub-object component is 'closer' to the mouse cursor.

**ulong hitInfo;**

A general unsigned long value. Most controllers will just need this to identify the sub-object element. The meaning of this value (how it is used to identify the element) is up to the plug-in.

**DWORD infoExtra;**

If the above **hitInfo** data member is not sufficient to describe the sub-object element this data member may be used as well.

# Class PostLoadCallback

See Also: [Class ILoad](#), [References](#).

```
class PostLoadCallback : public InterfaceServer
```

## Description:

This is the callback object used by **ILoad::RegisterPostLoadCallback()**. The **proc()** method will be called when everything has been loaded and all the references are in place. It is assumed that if the callback needs to be deleted, the **proc()** method will do it.

## Methods:

### Prototype:

```
virtual void proc(ILoad *iLoad)=0;
```

### Remarks:

Implemented by the Plug-In.

This method will be called when loading is complete.

### Parameters:

**ILoad \*iLoad**

This class provides methods to load data from disk.

### Prototype:

```
virtual int Priority();
```

### Remarks:

This method is available in release 3.0 and later only.

This method determines the order that the various registered callbacks execute. This method is overridden, for example, by the ParamBlock2 PostLoadCallbacks to return 1 so it can execute before the others.

### Return Value:

The allowable return values are **0** to **10**, with 5 being the default. **0** is reserved for ParamBlock2PLCB and ParamBlockPLCB..

### Default Implementation:

```
{ return 5; }
```

**Prototype:**

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0);
```

**Remarks:**

This method is available in release 3.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

This is reserved for future use.

**Parameters:****int cmd**

The command to execute.

**ULONG arg1=0**

Optional argument 1 (defined uniquely for each **cmd**).

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value (defined uniquely for each **cmd**).

**Default Implementation:**

```
{ return 0; }
```

## Class ClassDirectory

See Also: [Class DllDir](#), [Class SubClassList](#), [Class Class\\_ID](#), [Class ClassDesc](#).

class ClassDirectory

### Description:

This class is available in release 2.0 and later only.

It provides a table of **SubClassList** objects, one for each pluggable super class. Methods are provided for accessing specific sub class lists by specifying a super class ID, and retrieving the class descriptors or class entries of the classes in the lists.

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**int Count();**

#### Remarks:

Returns the number of class lists in the class directory.

#### Prototype:

**SubClassList\* GetClassList(SClass\_ID superClassID);**

#### Remarks:

Returns a pointer to the list of sub-classes derived from the specified super class ID.

#### Parameters:

**SClass\_ID superClassID**

The super class ID whose list of sub-classes are to be returned.

#### Prototype:

**ClassDesc\* FindClass(SClass\_ID superClassID, Class\_ID subClassID);**

#### Remarks:

Returns a pointer to the class descriptor for the class whose super class ID and class ID are passed.

**Parameters:****SClass\_ID superClassID**

The super class ID which specifies which sub class list to search.

**Class\_ID subClassID**

The class ID of the class to find.

**Prototype:****ClassEntry \*FindClassEntry(SClass\_ID superClassID, Class\_ID subClassID);****Remarks:**

Returns a pointer to the class entry for the class whose super class ID and class ID are passed.

**Parameters:****SClass\_ID superClassID**

The super class ID which specifies which sub class list to search.

**Class\_ID subClassID**

The class ID of the class to find.

**Prototype:****void AddSuperClass(SClass\_ID superClassID);****Remarks:**

This method is used internally **only** to add a new Super Class ID to the list maintained by this class.

**Important Note:** It is illegal to create your own Super Class of persistent objects in 3ds max. 3ds max will crash if a scene is saved and later loaded into a system that doesn't have a plug-in added Super Class. The problem is that 3ds max needs a stand-in class for each 'plug-able' Super Class and if a plug-in is loaded that requires (but doesn't have) a stand-in, an assertion is fired.

The only legal Super Classes that you can derive your plug-in from are listed [List of Super Class IDs](#).

**Prototype:****int AddClass(ClassDesc \*cdesc, int dllNum, int index);**

**Remarks:**

This method is used internally to add a class descriptor to the list maintained by this class.

**Operators:****Prototype:**

```
SubClassList& operator[](int i){ return(*cl[i]);}
```

**Remarks:**

Returns a reference to the 'i-th' sub class list.

**Parameters:**

**int i**

Specifies which sub class list to return a reference to. This value is between **0** and **Count()-1**.

## Class PickObjectProc

See Also: [Class IMtlParams](#), [Class INode](#).

class PickObjectProc

### Description:

This class is available in release 2.0 and later only.

This is the callback object passed to **IMtlParams::SetPickMode()**. It gets set so the user can pick objects in the scene. Its methods allow for filtering the picks, changing cursors over valid hits, and allowing multiple picks.

All methods of this class are implemented by the plug-in.

### Methods:

#### Prototype:

**virtual BOOL Pick(INode \*node)=0;**

#### Remarks:

This method is called when the user picks something.

#### Parameters:

**INode \*node**

This is the node that was selected.

#### Return Value:

Return TRUE to end the pick mode; FALSE to ignore the pick and to continue to allow the user to pick.

#### Prototype:

**virtual BOOL Filter(INode \*node)=0;**

#### Remarks:

This method is called to allow the callback to filter hits. It should return TRUE if this is an acceptable hit; FALSE otherwise.

#### Parameters:

**INode \*node**

This is the node that was selected.

**Prototype:****virtual void EnterMode();****Remarks:**

This method is called when the pick mode has been entered. The developer may provide any pre-processing here.

**Default Implementation:****{}****Prototype:****virtual void ExitMode();****Remarks:**

This method is called when the pick mode has been exited. The developer may provide any post-processing here.

**Default Implementation:****{}****Prototype:****virtual HCURSOR GetDefCursor();****Remarks:**

This method is called to get the default (non-hit) cursor to use.

**Return Value:**

The handle of the non-pickable object cursor or NULL if the default cursor should be used.

**Default Implementation:****{return NULL;}****Prototype:****virtual HCURSOR GetHitCursor();****Remarks:**

This method is called to get the pickable object cursor to use.

**Return Value:**

The handle of the pickable object cursor or NULL if the default cursor should

be used.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual BOOL AllowMultiSelect();
```

**Remarks:**

This method determines if multiple objects can be picked. Returning TRUE allows the user to pick more than one thing. In this case the **Pick()** method may be called more than once.

**Default Implementation:**

```
{return FALSE;}
```

## Class LayerProperty

See Also: [Class ReferenceTarget](#), [Class ILayerManager](#).

```
class LayerProperty : public ReferenceTarget
```

### Description:

This class is available in release 3.0 and later only.

The methods of this class are currently unused and reserved for internal use as of release 3.0.

```
class LayerProperty : public ReferenceTarget
```

### Methods:

```
public:
```

### Prototype:

```
    LayerProperty();
```

### Remarks:

Constructor.

This will set the layer ID to -1 and an empty name string.

### Default Implementation:

```
{ }
```

### Prototype:

```
    LayerProperty(const TSTR & name, int id);
```

### Remarks:

Constructor.

This will initialize the layer property with the name and ID of the property.

### Default Implementation:

```
{ }
```

### Prototype:

```
    virtual ~LayerProperty();
```

### Remarks:

Destructor.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void SetProperty(const int d) = 0;
```

**Remarks:**

This method allows you to set the integer property.

**Parameters:**

**const int d**

The property to set.

**Prototype:**

```
virtual void SetProperty(const float d) = 0;
```

**Remarks:**

This method allows you to set the floating point property.

**Parameters:**

**const float d**

The property to set.

**Prototype:**

```
virtual void SetProperty(const Point3 & d) = 0;
```

**Remarks:**

This method allows you to set the Point3 property.

**Parameters:**

**const Point3 & d**

The property to set.

**Prototype:**

```
virtual void SetProperty(const TSTR & d) = 0;
```

**Remarks:**

This method allows you to set the string property.

**Parameters:**

**const TSTR & d**

The property to set.

**Prototype:**

**virtual void SetProperty(void \* d) = 0;**

**Remarks:**

This method allows you to set the property.

**Parameters:**

**void \* d**

A pointer to the property data to set.

**Prototype:**

**virtual bool GetProperty(int & i) const = 0;**

**Remarks:**

This method returns the layer property.

**Parameters:**

**int & i**

The property data.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual bool GetProperty(float & f) const = 0;**

**Remarks:**

This method returns the layer property.

**Parameters:**

**float & f**

The property data.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual bool GetProperty(Point3 & p) const = 0;**

**Remarks:**

This method returns the layer property.

**Parameters:**

**Point3 & p**

The property data.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual bool GetProperty(TSTR & n) const = 0;**

**Remarks:**

This method returns the layer property.

**Parameters:**

**TSTR & n**

The property data.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual bool GetProperty(void \* v) const = 0;**

**Remarks:**

This method returns the layer property.

**Parameters:**

**void \* v**

The property data.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**int GetID() const;**

**Remarks:**

This method returns the property ID.

**Prototype:**

**void SetID(int id);**

**Remarks:**

This method allows you to set the property ID.

**Parameters:**

**int id**

The ID to set.

**Prototype:**

**TSTR GetName() const;**

**Remarks:**

This method returns the property name.

**Prototype:**

**void SetName(const TSTR & name);**

**Remarks:**

This method allows you to set the property name.

**Parameters:**

**const TSTR & name**

The property name to set.

## Class GenericNamedSelSetList

See Also: [Class BitArray](#).

### Description:

This class is available in release 3.0 and later only.

This class is a tool for manipulating lists of named selection sets. This class is used by modifiers such as the edit mesh, mesh select, spline select and edit patch.

All methods of this class are implemented by the system.

### Data Members:

public:

**Tab<TSTR\*> names;**

A table of names, one for each selection set.

**Tab<BitArray\*> sets;**

The bit array pointers for the selection sets.

**Tab<DWORD> ids;**

A table of Ids, one for each selection set.

### Methods:

public:

### Prototype:

**~GenericNamedSelSetList();**

### Remarks:

Destructor. The names and sets are deleted.

### Prototype:

**BitArray \*GetSet(TSTR name);**

### Remarks:

Returns a pointer to the bit array corresponding to the specified name. If the set is not found NULL is returned.

### Parameters:

**TSTR name**

The name of the selection set to retrieve.

**Prototype:**

**BitArray \*GetSet(DWORD id);**

**Remarks:**

Returns a pointer to the bit array corresponding to the specified ID. If the set is not found NULL is returned.

**Parameters:**

**DWORD id**

The id of the selection set to retrieve.

**Prototype:**

**BitArray \*GetSetByIndex(int index);**

**Remarks:**

Returns a pointer to the bit array corresponding to the specified index in the list. If the set is not found NULL is returned.

**Parameters:**

**int index**

The zero based index of the selection set to retrieve ( $\geq 0$  and  $< \text{sets.Count}()$ ).

**Prototype:**

**int Count();**

**Remarks:**

Returns the number of selection sets.

**Prototype:**

**void AppendSet(BitArray &nset, DWORD id=0, TSTR name=\_T(""));**

**Remarks:**

Appends the named selection set data to the list of sets maintained by this class.

**Parameters:**

**BitArray &nset**

The selection set data to append.

**DWORD id=0**

An ID for the selection set.

**TSTR name=\_T("")**

The name for the selection set.

**Prototype:**

```
void InsertSet(int pos, BitArray &nset, DWORD id=0, TSTR  
&name=TSTR(""));
```

**Remarks:**

Inserts the named selection set data into the list of sets maintained by this class.

**Parameters:****int pos**

The position in the list where this named selection set should be inserted. If pos >= Count(), AppendSet() is automatically used instead.

**BitArray &nset**

The selection set data to insert.

**DWORD id=0**

An ID for the selection set.

**TSTR &name=TSTR("")**

The name for the selection set.

**Prototype:**

```
int InsertSet(BitArray &nset,DWORD id=0,TSTR  
&name=TSTR(""));
```

**Remarks:**

This method is similar to **InsertSet()** above, however instead of accepting an explicit location this method inserts the new set alphabetically in the list. (Of course, this requires an alphabetized list to work properly, although there's no problem if the list is not alphabetized.)

**Parameters:**

**BitArray &nset**

The selection set data to insert.

**DWORD id=0**

An ID for the selection set.

**TSTR &name=TSTR("")**

The name for the selection set.

**Return Value:**

The position where the set was inserted.

**Prototype:**

**BOOL RemoveSet(TSTR name);**

**Remarks:**

Removes the selection set whose name is passed.

**Parameters:****TSTR name**

The name of the selection set to remove.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL RemoveSet(DWORD id);**

**Remarks:**

Removes the selection set whose ID is passed.

**Parameters:****DWORD id**

The id of the selection set to retrieve.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**void SetSize(int size);**

**Remarks:**

Resizes the selection set bit arrays to the specified number of bits. The old selection set data is preserved.

**Parameters:**

**int size**

The new size for the bit arrays in bits.

**Prototype:**

**void DeleteSetElements(BitArray &set,int m=1);**

**Remarks:**

This method is not currently used. What it does however, is go through all of the named selection sets and deletes array elements according to which bits are set in the given bit array. It could be used to keep the named selection set bit arrays in line with the vertex array (for example).

**Prototype:**

**void DeleteSet(int i);**

**Remarks:**

Deletes the named selection set whose index is passed.

**Parameters:**

**int i**

The zero based index of the set to delete ( $\geq 0$  and  $< \text{sets.Count}()$ ).

**Prototype:**

**BOOL RenameSet(TSTR &oldName, TSTR &newName);**

**Remarks:**

This locates the named selection set **oldName** and renames it to **newName**.

**Parameters:**

**TSTR &oldName**

The old name of the set.

**TSTR &newName**

The new name for the set.

**Return Value:**

TRUE if the operation succeeded; otherwise FALSE. It will only fail if the **oldName** set is not present.

**Prototype:**

**void Alphabetize();**

**Remarks:**

Alphabetizes the list of names.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

This method is used internally to load the selection sets from disk.

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

This method is used internally to save the selection sets to disk.

**Prototype:**

**BitArray &operator[](int i);**

**Remarks:**

Returns a reference to the 'i-th' selection set.

**Parameters:**

**int i**

The zero based index of the selection set to return.

**Prototype:**

**GenericNamedSelSetList& operator=(GenericNamedSelSetList& from);**

**Remarks:**

Assignment operator. This list of sets is emptied and then copied from the list passed.

**Parameters:**

**GenericNamedSelSetList& from**

The list of selection sets to copy.

## List of Parameter Types

See Also: [Class IParamBlock](#), [Class ParamBlockDesc](#), [Class ParamBlockDescID](#).

The Parameter Type - One of the following types may be used:

- **TYPE\_INT** - Integers values.
- **TYPE\_FLOAT** - Floating point value.
- **TYPE\_POINT3** - Point values.
- **TYPE\_RGBA** - Colors values - Red, Green, Blue and Alpha.
- **TYPE\_BOOL** - Boolean values.
- **TYPE\_MATRIX3** - A Matrix3 data type value. This type is available in 4.0 and later only.
- **TYPE\_MATRIX3\_TAB** - A table of Matrix3 data type values. is available in 4.0 and later only.

## Class ParamBlockDesc2

See Also: [Class ClassDesc2](#), [Class ParamMap2UserDlg](#), [Class Mtl](#), [Class Texmap](#), [Class PBBitmap](#), [Class PBAccessor](#), [Class INode](#), [Class Color](#), [Class Point3](#), [Class ReferenceTarget](#), [Structure ParamDef](#), [List of ParamType Choice](#), [List of ParamTags Choices](#).

```
class ParamBlockDesc2 : public BaseInterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

In the Parameter Block2 scheme there is one ParamBlockDesc2 object per entire Parameter Block2.

This class is used when adding a paramblock descriptor for each parameter block2. This is usually done in the form of a static instance of this class. The constructor takes a number of fixed, block-related arguments and then a varargs-based variable list of arguments that define the block and its parameters.

All methods of this class are implemented by the System.

### Method / Specification Groups:

The hyperlinks below take you to the start of groups of related methods or constructor vararg specifications within the class:

[Constructors](#)

[<required block specs>](#)

[<auto construct block refno>](#)

[<auto ui parammap specs>](#)

[<required param specs>](#)

[<optional tagged param specs>](#)

### Data Members:

public:

**ParamDef\* paramdefs;**

Array of parameter definitions.

**ClassDesc2\* cd;**

This is the class descriptor of the class which owns this parameter block descriptor.

**TCHAR\* int\_name;**

This is the internal name of this parameter descriptor. This name is not localized. Internal names are meant to be parsable as identifiers. As such they should begin with an alpha character, have only alphanumerics, and have no spaces, punctuations, etc. The convention for multi-word names is to use study-caps, eg, paintRadius.

**int local\_name;**

This is the string table resource ID for the localized (sub-anim) name.

**BlockID ID;**

The permanent parameter block ID.

**USHORT count;**

The number of parameters in block.

**ULONG version;**

This is the parameter block version.

**BYTE flags;**

One or more of the following values (see the constructor argument **flags** below for details):

**P\_CLASS\_PARAMS**

**P\_AUTO\_CONSTRUCT**

**P\_AUTO\_UI**

**P\_USE\_PARAMS**

**P\_INCLUDE\_PARAMS**

**P\_SCRIPTED\_CLASS**

**P\_TEMPORARY**

**P\_HASCATEGORY**

**P\_CALLSETS\_ON\_LOAD**

**P\_TEMPLATE\_UI**

Note: The following data members are optional and used by the auto-construct code:

**int ref\_no;**

The reference number for the auto-constructed parameter block.

**int dlg\_template;**

The rollout dialog template resource.

**int title;**

String table resource ID of the rollout title.

**int test\_flags;**

The **ClassDesc2::Begin/EndEditParams()** test flags. See the **flag\_mask** parameter description below for details.

**int rollup\_flags;**

This is used to control rollup creation. See the **rollup\_flags** parameter description below for details.

**ParamMap2UserDlgProc\* dlgProc;**

Points to the parameter map user dialog proc (if used).

**MSPluginClass\* pc;**

If this ParamBlockDesc2 belongs to a scripted plug-in this points to the scripted class (or NULL otherwise). See [MAXScript SDK](#).

**Value\* rollout;**

If this ParamBlockDesc2 belongs to a scripted plug-in this points to rollout name. See [MAXScript SDK](#).

**IParamBlock2\* class\_params;**

Pointer to class parameter block if the **CLASS\_PARAM** flag is specified for the block. See the flags descriptions below for details.

**Prototype:**

**void AddParam(ParamID id, ...);**

**Remarks:**

This method is used for building descriptors incrementally. It adds a parameter to an existing descriptor.

Note that you must not modify a descriptor with this function once it has been used to construct a ParamBlock2 (for instance in object creation); there is no version control in place and crashes or unpredictable results can occur.

**Parameters:**

**ParamID id, ...**

This function takes a single parameter definition in exactly the same varargs format as the ParamBlockDesc2 constructor. See [Constructors](#).

**Prototype:**

**void ReplaceParam(ParamID id, ...);**

**Remarks:**

This method is used for modifying a descriptors incrementally. It overrides an existing parameter definition of same ID passed. Note: You must not modify a descriptor with this method once it has been used to construct a ParamBlock2.

**Parameters:**

**ParamID id, ...**

This function takes a single parameter definition in exactly the same varargs format as the ParamBlockDesc2 constructor. See [Constructors](#).

**Prototype:**

**void DeleteParam(ParamID id);**

**Remarks:**

Deletes the specified parameter from the descriptor.

**Parameters:**

**ParamID id**

The permanent ID of the parameter to delete.

**Prototype:**

**void ParamOption(ParamID id, int option\_tag, ...);**

**Remarks:**

This method is used for modifying a descriptor incrementally. It alters a parameter definition optional information tag of an existing descriptor. Note: You must not modify a descriptor with this method once it has been used to construct a ParamBlock2.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**int option\_tag, ...**

This method takes a single tagged option in the same varargs format as the tagged parameter options in the ParamBlockDesc2 constructor. See

[<optional\\_tagged\\_param\\_specs>](#)

**Prototype:**

```
void SetClassDesc(ClassDesc2* cd);
```

**Remarks:**

This method sets the ClassDesc2 pointer maintained by this class. You can only call this method once on a descriptor and then only if it has been constructed initially with a NULL **cd**. See the notes in the constructor.

**Parameters:**

**ClassDesc2\* cd**

Points to the ClassDesc to set.

**Constructors****Prototype:**

```
ParamBlockDesc2(BlockID ID, TCHAR* int_name, int  
local_name, ClassDesc2* cd, BYTE flags, ...);
```

**Remarks:**

This constructor takes a number of fixed, block-related arguments and then a varargs-based variable list of arguments that define the block and its parameters.

The format of the definitions in this shown below, but basically is in the form of a sequence of fixed specs followed by a variable number of tagged optional specs for each parameter.

**Parameters:**

The generic form for the parameters of this constructor is shown below:

```
ParamBlockDesc2(<required_block_specs>,  
[<auto_construct_block_refno>],  
[<auto_ui_parammap_specs>],  
{<required_param_specs>,  
 {<optional_tagged_param_specs>},  
end  
,}  
end);
```

That is:

1. Required block specs & per-block flags, followed by,
2. Optional owning object reference number for the block if auto-construct, followed by,
3. Optional parameter map specs if auto-ui, followed by,
4. Zero or more parameter specs, comprising:
  - a. Required parameter specs, followed by,
  - b. Zero or more optional parameter specs, each with its own leading tag, the list terminated by an 'end' tag, followed by,
5. an 'end' tag

### <required\_block\_spec>

The following required first arguments to the constructor form the "required\_block\_spec" and "per block flags":

#### **BlockID ID**

The permanent block ID for the parameter block2.

#### **TCHAR\* int\_name**

The internal name string. This name is not localized. Internal names are meant to be parsable as identifiers. As such they should begin with an alpha character, have only alphanumerics, and have no spaces, punctuations, etc. The convention for multi-word names is to use studly-caps, eg, paintRadius.

#### **int local\_name**

The resource ID of the localized (sub-anim) name string.

#### **ClassDesc2\* cd**

Points to the class descriptor2 of the owning class. This is used to add this descriptor to the ClassDesc2's table of block descriptors for the class. Note: This value may be passed as NULL for situations where the blocks owning ClassDesc2 is not available for static initializations (such as in a separate file). Before using the descriptor for any block construction, the ClassDesc2\* must be initialized with the method:

```
void ParamBlockDesc2::SetClassDesc(ClassDesc2* cd);
```

You can only call this method once on a descriptor and then only if it has been constructed initially with a NULL **cd**.

#### **BYTE flags**

Per block/descriptor flags. One or more of the following values (they may be added together as in **P\_AUTO\_CONSTRUCT + P\_AUTO\_UI**).

## **P\_CLASS\_PARAMS**

Indicates this block holds class-level parameters which are attached to the **ClassDesc2** for the plug-in. Such class level parameters are shared by each instance of the class. The block is automatically allocated by and stored in the descriptor. You get at its parameters via **GetValue()/SetValue()** calls on the descriptor.

## **P\_AUTO\_CONSTRUCT**

Indicates the parameter block will be constructed and referenced automatically to its owner in the call to **ClassDesc2::MakeAutoParamBlocks()**. If this flag is set, the parameter block's reference number in the owning object should be given immediately following the flag word in the descriptor constructor. See [<auto\\_construct\\_block\\_refno>](#).

## **P\_AUTO\_UI**

Indicates this block supports automatic UI rollout management in calls to **ClassDesc2::BeginEditParams()**, **ClassDesc2::EndEditParams()**, **ClassDesc2::CreateParamDlg()**, **ClassDesc2::CreateParamDialog()**, etc.

If set, the [<auto\\_ui\\_parammap\\_specs>](#) must be supplied in the descriptor constructor.

## **P\_HASCATEGORY**

This parameter is available in release 4.0 and later only.

The category field that can be used to order rollups (see [Class IRollupWindow](#)) to various Parameter Map creation methods. In order to use the category field with AutoUI, this flag has to be declared together with **P\_AUTO\_UI** in the ParamBlockDesc2. An additional int, that describes the category has to be appended to the parameter list after the **ParamMap2UserDlgProc\* proc** parameter. The same thing is true for multimaps. The **P\_HASCATEGORY** field can only be declared for the whole pblock. That means, that every multimap has to have the category parameter. To use the standard value

**ROLLUP\_CAT\_STANDARD** can be used. In the example below 4900 is used as the integer describing the category;

```
static ParamBlockDesc2 std2_shader_blk ( std2_shader,
```

```

    _T("shaderParameters"), 0, &stdmtl2CD,
    P_AUTO_CONSTRUCT + P_AUTO_UI +
P_HASCATEGORY, SHADER_PB_REF,
    //rollout
    IDD_DMTL_SHADER4, IDS_KE_SHADER, 0, 0,
&shaderDlgProc, 4900,
    // params
    std2_shader_type, _T("shaderType"), TYPE_INT, 0,
IDS_JW_SHADERTYPE,

```

### **P\_USE\_PARAMS**

Indicates that this block shares (exactly) the paramdefs from another descriptor. This is used to specify an already established ParamBlockDesc2 instance whose parameters you wish to share. This effectively gives the referring descriptor a pointer to the established descriptors 'paramdefs' array. In this case no other parameters definition can be supplied to referencing descriptors constructor. See the sample code below.

### **P\_INCLUDE\_PARAMS**

Indicates that this block loads in a copy the paramdefs from another descriptor. This is used to take a copy of an already established descriptor's parameters, to which you can add extra parameter definitions in the referencing descriptors constructor. This provides a kind of poor-man's factoring of common parameters, but note it is a copy; any subsequent changes to the established descriptor's parameter definitions are not reflected in the referencing descriptor.

You give the pointer to the ParamBlockDesc2 supplying the existing parameter definitions following the other optional block-level parameters: following the flag word is the block's reference number if

**P\_AUTO\_CONSTRUCT** is specified, then the rollout dialog template info if **P\_AUTO\_UI** is specified, then the pointer to the sourcing descriptor if **P\_USE\_PARAMS** or **P\_INCLUDE\_PARAMS** is specified. For example:

```

    static ParamBlockDesc2 metal2_param_blk ( shdr_params,

```

```

    _T("shaderParameters"), 0, &metalCD,
    P_AUTO_CONSTRUCT + P_USE_PARAMS,
    // pblock refno
    0,
    // use params from existing descriptor
    &const_param_blk
);

```

### **P\_SCRIPTED\_CLASS**

This is for internal use only. It means that the descriptor was defined on the fly as a side-effect of writing a scripted plug-in.

### **P\_TEMPORARY**

This is for internal use only.

### **P\_TEMPLATE\_UI**

This indicates that dialog templates will be provided or constructed.

### **P\_CALLSETS\_ON\_LOAD**

Signals that this block should have **CallSets()** called on it during post-load processing on scene loads & merges. This effectively ensures that all **PBAccessor::Set()** methods will be called after the flagged pblock2 is fully loaded, so that they can track loaded param values, for example. This allows a single point of param value tracking in the **PBAccessor::Set()** and precludes the need for individual objects to implement PLCBs to do this tracking themselves.

### **P\_MULTIMAP**

This flag is available in release 4.0 and later only.

Indicates that the block being described will have more than one rollup/map. If you specify this flag, the constructor interprets the rollup template and parameter definitions arguments in a modified syntax. Here's a sample rework of the main pblock in GeoSphere into two rollups:

```

enum { geo_map_1, geo_map_2 }; // enum IDs for the 2
parammaps

```

```

static ParamBlockDesc2 geo_param_blk ( geo_params,
    _T("GeosphereParameters"), 0, &gsphereDesc,
    P_AUTO_CONSTRUCT +
    P_AUTO_UI + P_MULTIMAP, PBLOCK_REF_NO,
    // map rollups
    2,
    geo_map_1, IDD_GSPHERE_1, IDS_PARAMETERS_1, 0, 0,
    NULL,
    geo_map_2, IDD_GSPHERE_2, IDS_PARAMETERS_2, 0, 0,
    NULL,
    // params
    geo_hemi, _T("hemisphere"), TYPE_BOOL,
    P_ANIMATABLE, IDS_HEMI,
    p_default, FALSE,
    p_ui, geo_map_2, TYPE_SINGLECHECKBOX, IDC_HEMI,
    end,
    geo_segs, _T("segs"), TYPE_INT, P_ANIMATABLE,
    IDS_RB_SEGS,
    p_default, 4,
    p_range, MIN_SEGMENTS, MAX_SEGMENTS,
    p_ui, geo_map_1, TYPE_SPINNER, EDITTYPE_INT,
    IDC_SEGMENTS, IDC_SEGSPINNER, 0.05f,
    end,
    geo_radius, _T("radius"), TYPE_FLOAT, P_ANIMATABLE +
    P_RESET_DEFAULT, IDS_RB_RADIUS,
    p_default, 0.0,
    p_ms_default, 25.0,
    p_range, MIN_RADIUS, MAX_RADIUS,
    p_ui, geo_map_1, TYPE_SPINNER, EDITTYPE_UNIVERSE,
    IDC_RADIUS, IDC_RADSPINNER, 1.0,
    p_uix, geo_map_2,

```

**end,**

...

First, there is an enum to provide IDs for the two maps in the main block, **geo\_map\_1** and **geo\_map\_2**. The **P\_MULTIMAP** flag is added to the block flags in the main descriptor constructor arguments to indicate multiple pmaps present. If **P\_AUTO\_UI** is specified, the usual single rollup template spec is replaced by a count (of rollups) followed by that many sets of rolup specs, each beginning with the associated mapID. The auto UI mechanism will add the rollups in the order given in this list.

In the parameter definition section, the only change is to the **p\_ui** option, which now requires a map ID before the rest of the UI specification to say which rollup/map the spec relates to. In this case, we've put the hemisphere checkbox in the 2nd rollup and the segs and radius spinners in the first.

There is also a new option, **p\_uix**, which is used to say that the parameter is to appear in more than one rollup. In this case, the radius spinner also shows up in the 2nd rollup, **geo\_map\_2**. When you do this, all the controls that connect to this parameter are ganged together; they all change when any one of them changes and all show keyframe highlights and so on. The current limitations on this are that the type of UI control and its various dialog template item IDs must be the same in each rollup in which it appears.

<auto\_construct\_block\_refno>

If **P\_AUTO\_CONSTRUCT** is specified in the required per block / descriptor flags (**BYTE flags** above) then the integer reference number of the parameter block2 in the plug-in needs to be specified:

**int ref\_no**

This is the same number that the plug-in would use to get and set the parameter block reference in **GetReference()** and **SetReference()**.

<auto\_ui\_parammap\_specs>

If **P\_AUTO\_UI** is specified in the required per block / descriptor flags (**BYTE flags** above) then the following arguments to the constructor are required:

**int dialog\_template\_ID, int dialog\_title\_res\_ID, int flag\_mask, int**

**rollup\_flags, ParamMap2UserDlgProc\* proc**

Each of these is described below:

**int dialog\_template\_ID**

The ID of the dialog template (eg IDD\_something).

**int dialog\_title\_res\_ID**

The string table resource ID for the title of the dialog.

**int flag\_mask**

This is used by **ClassDesc2::BeginEditParams()** and **ClassDesc2::EndEditParams()** to determine whether the ParamMap2 should be created/deleted on this call. All the bits in the supplied mask must be on in the Begin/EndEditParams flag longword for the action to take place.

**int rollup\_flags**

This flag is used to control rollup creation. You may pass:

**APPENDROLL\_CLOSED**

to have the rollup added in the closed (rolled up) state. Otherwise pass 0.

**ParamMap2UserDlgProc\* proc**

If there are controls in the dialog that require special processing this user dialog proc can be implemented to process them. See [Class ParamMap2UserDlgProc](#). If not used then NULL should be passed.

**<required\_param\_specs>**

The required parameter spec is formatted as shown below. There is one of these for each of the controls managed by the parameter map. They are followed by an [<optional\\_tagged\\_param\\_spec>](#).

**ParamID id, TCHAR\* internal\_name, ParamType type, [int table\_size,] int flags, int local\_name\_res\_ID,**

Each of these is described below:

**ParamID id**

The permanent, position-independent ID for the parameter.

**TCHAR\* internal\_name**

The internal name for the parameter.

**ParamType type**

The type of parameter. See [List of ParamType Choices](#).

## **[int table\_size]**

If the type is one of the Tab<> types, you need to supply an initial table size which can be 0.

## **int flags**

The per parameter flag bits. This should be a bitwise OR of one or more of the following:

### **P\_ANIMATABLE**

Indicates the parameter is animatable.

### **P\_TRANSIENT**

Indicates the parameter should not be saved in the scene file. One might do this, for example, for virtual parameters that don't actually hold data but reflect data held elsewhere in the object (and accessed via a PBAccessor) which is saved explicitly by it. This might save file space. In some cases, parameters might be provided that are derived from other state in the object that is computed each time you load the object and made available say as a help to script authors. In these cases also, you might decide not to take up file space.

### **P\_NO\_INIT**

This is obsolete.

### **P\_COMPUTED\_NAME**

Indicates to call a compute name function to get a dynamically-computed name. This allows a plug-in to provide a dynamically-created local name for a parameter or Tab<> parameter entry. If you specify this parameter flag, you also need to supply a **p\_accessor PBAccessor** instance pointer that has the **GetLocalName()** method implemented.

### **P\_COMPUTED\_DIM**

Indicates to call a compute dimension function to get a dynamically-computed dimension. See the [optional tagged param spec p\\_dim](#).

### **P\_RESET\_DEFAULT**

Indicates to not make creation parameters sticky, rather always reset to default values.

### **P\_SUBANIM**

Indicates this is a non-animatable reference target parameter to be

published as a sub-anim (which makes it visible in Track View)

### **P\_TV\_SHOW\_ALL**

This is used for Tab<> animatables, and indicates to show all entries in Track View even if no controller assigned.

### **P\_NO\_REF**

For reference target parameters, this indicates to not maintain the reference automatically (rather simply keep a copy of the pointer).

### **P\_OWNERS\_REF**

Indicates this is a reference target parameter owned by the parameter block's owner not the block itself. Make sure to supply the owner's reference number in a [p\\_refno](#) specification. If neither **P\_REF\_NO** or **P\_OWNERS\_REF** is set, the parameter block owns and maintains the reference.

### **P\_SUBTEX**

Indicates a texmap parameter is kept by the owner using the `MtlBase::xSubTexmap` methods. Provide the integer index of the sub-texmap using the param tag [p\\_subtexno](#).

You would use this flag in materials or texmaps that contain other texmaps as parameters, in which these 'sub' maps are not stored in the `ParamBlock2` and are **not** accessible as direct references on the parent map or material, but are accessible via the `MtlBase::GetSubTexmap()/SetSubTexmap()` protocol.

Specifying this **P\_SUBTEX** flag signals this situation and looks for a `p_subtexno` parameter option to give the sub-texmap number for the map. For example, the Standard material stores all its maps in a separate structure that appears as a single reference in the Standard material. In this case, the individual texmaps are not direct references on the material, but are accessible on the material via `GetSubTexmap()/SetSubTexmap()`.

Note that **P\_OWNERS\_REF** and **P\_SUBTEX** are both primarily intended for use when re-coding existing plug-ins in which the sub-maps are already managed by the owner in some way. If you are implementing a new plug-in, you should just let the `ParamBlock2` host them for you and then you don't have to bother with either of these flags.

### **P\_CAN\_CONVERT**

Indicates the **p\_classID** validator is used in a **CanConvertTo()** call, rather than as exact classID match.

### **P\_VARIABLE\_SIZE**

Indicates a Tab<> parameter is variable in size. This allows scripted changes.

### **P\_NO\_AUTO\_LABELS**

Disables the automatic setting of button text for texmaps, materials, files, etc. You can use the method **IParamMap2::SetText()** to set it by hand.

### **P\_SHORT\_LABELS**

This is for use with **TYPE\_TEXMAP**, **TYPE\_MTL** and **TYPE\_BITMAP** parameters that are associated with ParamMap2 picker buttons. If you specify this flag, a shortened form of the object name is installed in the picker button label. For texmaps and materials, the **MtlBase::GetName()** rather than **MtlBase::GetFullName()** is used, and for bitmaps just the filename rather than the full pathname is used.

### **int local\_name\_res\_ID**

The localized name for the parameter. This is a ID for the resource in the string table.

### **<optional\_tagged\_param\_specs>**

There may be zero or more optional tagged parameter specs, each with its own leading tag, with the entire list terminated by an 'end' tag. This list of tagged arguments has the following form:

**<tag>**, **<optional\_param\_spec>**,

For the possible tags and the specification arguments see [List of ParamTags Choices](#).

### **Methods:**

public:

### **Prototype:**

**USHORT Count();**

### **Remarks:**

Returns the number of parameters in the block.

**Prototype:**

**DWORD Version();**

**Remarks:**

Returns the version of the parameter block.

**Prototype:**

**int IDtoIndex(ParamID id);**

**Remarks:**

Returns the index into the parameter definition array of the parameter whose ID passed.

**Parameters:**

**ParamID id**

The permanent parameter ID.

**Prototype:**

**ParamID IndextoID(int i);**

**Remarks:**

Returns the permanent parameter ID of the parameter whose index is passed.

**Parameters:**

**int i**

The zero based index of the parameter in the **paramdefs** array.

**Prototype:**

**ParamDef& GetParamDef(ParamID id);**

**Remarks:**

This method is used for accessing a parameter's ParamDef structure.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**Prototype:**

**BOOL SetValue(ParamID id, TimeValue t, float v, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.  
Sets the floating point value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**float v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL SetValue(ParamID id, TimeValue t, int v, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.  
Sets the integer value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**int v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the

value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL SetValue(ParamID id, TimeValue t, Point3& v, int  
tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Sets the Point3 value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Point3& v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL SetValue(ParamID id, TimeValue t, Color& v, int  
tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Sets the Color value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Color& v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
BOOL SetValue(ParamID id, TimeValue t, TCHAR* v, int  
tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Sets the string value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**TCHAR\* v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
BOOL SetValue(ParamID id, TimeValue t, Mtl* v, int  
tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.  
Sets the Mtl\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Mtl\*v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
BOOL SetValue(ParamID id, TimeValue t, Texmap* v, int  
tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.  
Sets the Texmap\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Texmap\* v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
BOOL SetValue(ParamID id, TimeValue t, PBBitmap* v, int  
tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Sets the PBBitmap\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**PBBitmap\* v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
BOOL SetValue(ParamID id, TimeValue t, INode* v, int  
tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Sets the INode\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**INode\* v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL SetValue(ParamID id, TimeValue t, ReferenceTarget\* v, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Sets the ReferenceTarget\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**ReferenceTarget\*v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL SetValue(ParamID id, TimeValue t, Matrix3& v, int tabIndex=0);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used with static class parameter blocks only.

Sets the Matrix3 value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to set the value.

**Matrix3& v**

The value to set.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to set.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
BOOL GetValue(ParamID id, TimeValue t, float& v, Interval  
&ivalid, int tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the floating point value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**float& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved

parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL GetValue(ParamID id, TimeValue t, int& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the integer value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**int& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL GetValue(ParamID id, TimeValue t, Point3& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the Point3 value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Point3& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
BOOL GetValue(ParamID id, TimeValue t, Color& v, Interval  
&ivalid, int tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the Color value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Color& v**

The value to retrieve is returned here.

### **Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

### **int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

### **Return Value:**

TRUE on success; otherwise FALSE.

### **Prototype:**

**BOOL GetValue(ParamID id, TimeValue t, TCHAR\*& v, Interval &ivalid, int tabIndex=0);**

### **Remarks:**

This method is used with static class parameter blocks only.

Retrieves the string value of the specified parameter at the specified time.

### **Parameters:**

#### **ParamID id**

The permanent ID of the parameter.

#### **TimeValue t**

The time at which to get the value.

#### **TCHAR\*& v**

The value to retrieve is returned here.

#### **Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

#### **int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

### **Return Value:**

TRUE on success; otherwise FALSE.

### **Prototype:**

**BOOL GetValue(ParamID id, TimeValue t, Mtl\*& v, Interval**

**&ivalid, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the Mtl\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Mtl\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL GetValue(ParamID id, TimeValue t, Texmap\*& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the Texmap\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Texmap\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL GetValue(ParamID id, TimeValue t, PBBitmap\*& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the PBBitmap\* value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**PBBitmap\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL GetValue(ParamID id, TimeValue t, INode\*& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the INode\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**INode\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**BOOL GetValue(ParamID id, TimeValue t, ReferenceTarget\*& v, Interval &ivalid, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the ReferenceTarget\* value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**ReferenceTarget\*& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

```
BOOL GetValue(ParamID id, TimeValue t, Matrix3& v, Interval  
&ivalid, int tabIndex=0);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is used with static class parameter blocks only.

Retrieves the Matrix3 value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t**

The time at which to get the value.

**Matrix3& v**

The value to retrieve is returned here.

**Interval &ivalid**

This is the validity interval which is updated by the validity of the retrieved parameter.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

TRUE on success; otherwise FALSE.

**Prototype:**

**Color GetColor(ParamID id, TimeValue t=0, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Returns the Color value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

The Color value of the parameter.

**Prototype:**

**Point3 GetPoint3(ParamID id, TimeValue t=0, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Retrieves the value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

The Point3 value of the parameter

**Prototype:**

```
int GetInt(ParamID id, TimeValue t=0, int tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

The integer value of the parameter.

**Prototype:**

```
float GetFloat(ParamID id, TimeValue t=0, int tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

The floating point value of the parameter.

**Prototype:**

```
TimeValue GetTimeValue(ParamID id, TimeValue t=0, int  
tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

The TimeValue value of the parameter.

**Prototype:**

```
TCHAR* GetStr(ParamID id, TimeValue t=0, int tabIndex=0);
```

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the

value to get.

**Return Value:**

The TimeValue value of the parameter.

**Prototype:**

**Mtl\* GetMtl(ParamID id, TimeValue t=0, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

A pointer to the Mtl object.

**Prototype:**

**Texmap\* GetTexmap(ParamID id, TimeValue t=0, int  
tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

A pointer to the Texmap.

**Prototype:**

**PBBitmap\* GetBitmap(ParamID id, TimeValue t=0, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

A pointer to the PBBitmap object.

**Prototype:**

**INode\* GetINode(ParamID id, TimeValue t=0, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

A pointer to the INode.

**Prototype:**

**ReferenceTarget\* GetReferenceTarget(ParamID id, TimeValue t=0, int tabIndex=0);**

**Remarks:**

This method is used with static class parameter blocks only.

Returns the value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

A pointer to the ReferenceTarget.

**Prototype:**

**Matrix3 GetMatrix3(ParamID id, TimeValue t=0, int tabIndex=0);**

**Remarks:**

This method is available in release 4.0 and later only.

This method is used with static class parameter blocks only.

Retrieves the value of the specified parameter at the specified time.

**Parameters:**

**ParamID id**

The permanent ID of the parameter.

**TimeValue t=0**

The time at which to get the value.

**int tabIndex=0**

If the parameter is a Tab<> this is the zero based index into the table of the value to get.

**Return Value:**

The Matrix3 value of the parameter

**Prototype:**

```
TCHAR* GetString(int id);
```

**Remarks:**

Returns a string resource from plug-in module's resource.

**Parameters:****int id**

The permanent ID of the parameter.

**Prototype:**

```
void InvalidateUI();
```

**Remarks:**

This method invalidates any current parameter map2 user interface currently open for this descriptor.

**Prototype:**

```
void InvalidateUI(ParamID id, int tabIndex=-1);
```

**Remarks:**

This method invalidates the control whose parameter ID is specified.

**Parameters:****ParamID id**

The permanent ID of the parameter.

**int tabIndex=-1**

If the parameter is a Tab<> this is the zero based index of parameter whose associated control is to be redrawn.

**Prototype:**

```
void SetUserDlgProc(ParamMap2UserDlgProc* proc=NULL);
```

**Remarks:**

This method allows for special handling for a set of controls. The developer provides a dialog proc object to process the message from the controls. This method is used to tell the parameter map that the developer defined method should be called. The given proc will be called after default processing is done. Note that if the proc is non-NULL when the ParamMap is deleted its DeleteThis() method will be called.

Note, in version 4.0 and later, this actually maps to a call on the explicit map ID overload of **SetUserDlgProc()** with default map ID of 0.

**Parameters:**

```
ParamMap2UserDlgProc* proc=NULL
```

A pointer to the user dialog proc object to process the control.

**Prototype:**

```
void SetUserDlgProc(MapID map_id, ParamMap2UserDlgProc*  
proc=NULL);
```

**Remarks:**

This method is available in release 4.0 and later only.

This overload of **SetUserDlgProc()** has a new parameter, **map\_id**, that specifies the ID of the parameter map/rollup to set the user dialog proc for. See original function for the rest of the description.

**Prototype:**

```
ParamMap2UserDlgProc* GetUserDlgProc(MapID map_id = 0);
```

**Remarks:**

Returns the user dialog proc for the parameter map associated with this descriptor.

**Parameters:**

```
MapID map_id
```

This parameter is available in release 4.0 and later only.

Specifies the ID of the map/rollout to get the user dialog proc for.

**Prototype:**

**void SetOwnerRefNo(ParamID id, int refno);**

**Remarks:**

This method allows dynamic setting of the **P\_OWNERS\_REF** reference number for given Reference Target parameter.

**Parameters:**

**ParamID id**

The permanent ID for the parameter.

**int refno**

The reference number to set.

**Prototype:**

**int GetOwnerRefNo(ParamID id);**

**Remarks:**

This method returns the **P\_OWNERS\_REF** reference number for given Reference Target parameter.

**Parameters:**

**ParamID id**

The permanent ID for the parameter.

**Prototype:**

**void SetSubTexNo(ParamID id, int texno);**

**Remarks:**

Sets the sub-texture number for the specified texmap parameter. You can use this to dynamically change a parameter's sub object number.

**Parameters:**

**ParamID id**

The parameter ID for the texmap.

**int texno**

The sub-texture number to set.

**Prototype:**

**void SetSubMtlNo(ParamID id, int mtlno);**

**Remarks:**

Sets the sub-material number for the specified texmap parameter. You can use this to dynamically change a parameter's sub object number.

**Parameters:**

**ParamID id**

The parameter ID for the material.

**int mtlno**

The sub-material number to set.

**Prototype:**

**int GetSubTexNo(ParamID id);**

**Remarks:**

Returns the sub-texture number for the specified parameter.

**Parameters:**

**ParamID id**

The ID of the parameter.

**Prototype:**

**int GetSubMtlNo(ParamID id);**

**Remarks:**

Returns the sub-material number for the specified parameter.

**Parameters:**

**ParamID id**

The ID of the parameter.

**Prototype:**

**void SetInitFile(ParamID id, TCHAR\* s);**

**Remarks:**

This method allows dynamic setting of the **TYPE\_OPEN/SAVEFILEBUTTON p\_init\_file** field.

**Parameters:**

**ParamID id**

The permanent ID for the parameter.

**TCHAR\* s**

The string to set.

**Prototype:**

**TCHAR\* GetInitFile(ParamID id);**

**Remarks:**

This method returns the **TYPE\_OPEN/SAVEFILEBUTTON**  
**p\_init\_file** field.

**Parameters:**

**ParamID id**

The permanent ID for the parameter.

## List of ParamType2 Choices

See Also: [Class ParamBlockDesc2](#), [Template Class Tab](#), [Class ParamDimension](#), [Class Mtl](#), [Class Texmap](#), [Class INode](#), [Class PBBitmap](#), [Class ReferenceTarget](#).

The following are the valid types for parameters in parameter blocks.

The type is passed to the **ParamBlockDesc2** constructor as the **ParamType type** argument of the <required\_param\_spec>.

The first group are single parameters while the second are tables of the first set of parameters.

### **TYPE\_FLOAT**

A single floating point value.

### **TYPE\_INT**

A single integer value.

### **TYPE\_RGBA**

A Point3 value with an implied **stdColor255Dim** dimension.

### **TYPE\_POINT3**

A Point3 data type value.

### **TYPE\_BOOL**

An integer used as a boolean value.

### **TYPE\_ANGLE**

A floating point value with an implied **stdAngleDim** dimension.

### **TYPE\_PCNT\_FRAC**

A floating point with an implied **stdPercentDim** dimension.

### **TYPE\_WORLD**

Specify that a parameter represents world distance units. This implies a parameter dimension of **stdWorldDim**.

### **TYPE\_STRING**

A character string (**TCHAR\***). The string has a local copy made and managed by the paramblock.

### **TYPE\_FILENAME**

This is the same as **TYPE\_STRING**, but is used with

**TYPE\_FILEOPENBUTTON** and **TYPE\_FILESAVEBUTTON**  
parameter map controls.

### **TYPE\_HSV**

This option is obsolete.

### **TYPE\_COLOR\_CHANNEL**

A single floating point value with an implied **stdColor255Dim** dimension.

### **TYPE\_TIMEVALUE**

A single integer value used as a TimeValue -- implies a **stdTimeDim** dimension.

### **TYPE\_RADIOBTN\_INDEX**

This is currently unused but intended to allow specification of state names to make scripter access symbolic.

### **TYPE\_MTL**

A pointer to a material object (**Mtl\***). This can be one of three types: a reference owned by the parameter block, a reference owned by the block owner, or no reference management (just a copy of the pointer).

### **TYPE\_TEXMAP**

A pointer to a texmap object (**Texmap\***). This can be one of three types: a reference owned by the parameter block, a reference owned by the block owner, or no reference management (just a copy of the pointer).

### **TYPE\_BITMAP**

A pointer to a Bitmap/BitmapInfo object (**PBBitmap\***). This can be one of three types: a reference owned by the parameter block, a reference owned by the block owner, or no reference management (just a copy of the pointer).

### **TYPE\_INODE**

A pointer to a node (**Inode\***). This can be one of three types: a reference owned by the parameter block, a reference owned by the block owner, or no reference management (just a copy of the pointer).

### **TYPE\_REFTARG**

A pointer to a Reference arget (**ReferenceTarget\***), all the RefTarg types in this group can be one of three types: Reference owned by parameter block, Reference owned by block owner, No reference management (just a copy of the pointer).

## **TYPE\_INDEX**

This is used for parameters that are 0-based, but exposed to MAXScript as 1-based. For example a vertex index.

## **TYPE\_MATRIX3**

A standard max Matrix3

## **TYPE\_PBLOCK2**

A pointer to an IParamBlock2 object.

The following are tables of the above data types:

## **TYPE\_FLOAT\_TAB**

A table of floating point values.

## **TYPE\_INT\_TAB**

A table of integer values.

## **TYPE\_RGBA\_TAB**

A table of Point3 values with an implied **stdColor255Dim** dimension.

## **TYPE\_POINT3\_TAB**

A table of Point3 data type values.

## **TYPE\_BOOL\_TAB**

A table of integers used as a set of boolean values.

## **TYPE\_ANGLE\_TAB**

A table of floating point value with an implied **stdAngleDim** dimension.

## **TYPE\_PCNT\_FRAC\_TAB**

A table of same as **TYPE\_STRING**, but is used with **TYPE\_FILEOPENBUTTON** and **TYPE\_FILESAVEBUTTON** parameter map controls.

## **TYPE\_WORLD\_TAB**

A table of parameters that represents world distance units. This implies a parameter dimension of **stdWorldDim**.

## **TYPE\_STRING\_TAB**

A table of character strings (**TCHAR\***).

## **TYPE\_FILENAME\_TAB**

A table of filenames (**TYPE\_FILENAME** -- see notes above).

### **TYPE\_HSV\_TAB**

This option is obsolete.

### **TYPE\_COLOR\_CHANNEL\_TAB**

A table of floating point values with an implied **stdColor255Dim** dimension.

### **TYPE\_TIMEVALUE\_TAB**

A table of integer value used as a TimeValue -- implies **stdTimeDim** dimension.

### **TYPE\_RADIOBTN\_INDEX\_TAB**

This is currently unused.

### **TYPE\_MTL\_TAB**

A table of material object pointers (see **TYPE\_MTL** above).

### **TYPE\_TEXMAP\_TAB**

A table of texmap object pointers (see **TYPE\_TEXMAP** above).

### **TYPE\_BITMAP\_TAB**

A table of **TYPE\_BITMAP** values (see above).

### **TYPE\_INODE\_TAB**

A table of **TYPE\_INODE** values (see above).

### **TYPE\_REFTARG\_TAB**

A table of **TYPE\_REFTARG** values (see above).

### **TYPE\_MSFLOAT = 255**

This option is obsolete.

### **TYPE\_UNSPECIFIED = -255**

This option is obsolete.

## Structure ParamDef

See Also: [Class ParamBlockDesc2](#), [Structure PB2Value](#), [Class PBValidator](#), [Class PBBitmap](#), [List of ParamType2 Choices](#), [Class ParamDimension](#), [Template Class Tab](#).

```
struct ParamDef
```

### Description:

This structure is available in release 3.0 and later only.

The data members of this class provide a definition of a parameter. An array of these parameter definitions is a data member of class **ParamBlockDesc2**.

### struct ParamDef

```
{
```

#### **ParamID ID;**

This is the permanent, position independent ID of the parameter.

#### **TCHAR\* int\_name;**

This is a fixed internal name of the parameter. This name is not localized. Internal names are meant to be parsable as identifiers. As such they should begin with an alpha character, have only alphanumeric, and have no spaces, punctuations, etc. The convention for multi-word names is to use study-caps, eg, paintRadius.

#### **ParamType2 type;**

This is the type of the parameter. See [List of ParamType2 Choices](#).

#### **int flags;**

They are the per-parameter constructor flags (**P\_ANIMATABLE**, **P\_TRANSIENT**, etc.) Normally, the flags are set up as a result of things you specify in the **ParamBlockDesc2** constructor and should generally be read-only at runtime.

#### **int local\_name;**

This is the string table resource ID for the localized (sub-anim) name.

#### **ParamDimension\* dim;**

This is the parameter dimension. See [Class ParamDimension](#).

#### **PB2Value def;**

The default value for the parameter.

**PB2Value ms\_def;**

This is the default value for MAXScript and the MacroRecorder.

**PB2Value cur\_def;**

This is the current 'sticky' default value, used to maintain creation defaults within a session.

**int description;**

This is a one sentence description. Use a string resource ID.

**PB2Value range\_low;**

This indicates the low allowable range used in MAXScript validation and spinner setup.

**PB2Value range\_high;**

This indicates the high allowable range used in MAXScript validation and spinner setup.

**PBValidator\* validator;**

Points to an instance of the validator object. This object has a **Validate()** method used to check if the parameter is valid.

**PBAccessor\* accessor;**

Points to an instance of an accessor object. Any parameter in a block can have an accessor callback object that has its **Get()** or **Set()** method called whenever the parameter is accessed. This may be used to provide access to dynamically-computed virtual parameters and sometimes to allow parameter-specific processing by the class as the parameter in the block is modified.

**short tab\_size;**

If the parameter is a table (Tab<>) this is the initial table size.

**short ref\_no;**

This is a block-owner's reference number for non-hosted ReferenceTargets parameters.

**short subtex\_no;**

This is a block-owner's SubTex index for Texmap parameters in Mtl owners.

**Class\_ID class\_ID;**

This is the Class\_ID validator for reference targets.

**SClass\_ID sclass\_ID;**

This is the SClass\_ID validator for reference targets (similar to above).

**ControlType2 ctrl\_type;**

This is the type of user interface control.

**EditSpinnerType spin\_type;**

This is the spinner type if the associated UI control is a spinner. One of the following values may be used:

**EDITTYPE\_INT**

Any integer value.

**EDITTYPE\_FLOAT**

Any floating point value.

**EDITTYPE\_UNIVERSE**

This is a value in world space units. It respects the system's unit settings (for example feet and inches).

**EDITTYPE\_POS\_INT**

Any integer  $\geq 0$

**EDITTYPE\_POS\_FLOAT**

Any floating point value  $\geq 0.0$

**EDITTYPE\_POS\_UNIVERSE**

This is a positive value in world space units. It respects the system's unit settings (for example feet and inches) .

**EDITTYPE\_TIME**

This is a time value. It respects the system time settings (SMPTE for example).

**int\* ctrl\_IDs;**

This is the array of control IDs for this parameter.

**short ctrl\_count;**

This is the number of controls in the **ctrl\_IDs** array above.

**int\* val\_bits;**

These are radio button vals or bit numbers for int bits controlled by multiple checkboxes. The numbers in the int array are used to indicate which bit to flip in the TYPE\_INT parameter depending on the state of the associated (by order) checkbox. This is not yet implemented.

**float scale;**

This is the scale given to the ISpinnerControl, as is used in SetupFloatSpinner(), for example.

**int numSegs;**

This is the slider segments count.

**ParamID\* enable\_ctrls;**

The array of which other parameters have their UI controls automatically enabled by this parameter.

**short enable\_count;**

This is the number of parameter IDs in the **enable\_ctrls** array above.

**int prompt;**

The status line prompt string resource ID for various picker buttons.

**int caption;**

The caption string resource ID for open/save file dialogs.

**TCHAR\* init\_file;**

The initial filename for open/save file dialogs.

**int file\_types;**

The file types string resource ID for open/save file dialogs (in MAXScript type: form)

**void DeleteThis();**

This function deletes this instance of the structure.

## Class PBValidator

See Also: [Structure ParamDef](#), [List of Type 2 Params](#), [Class ParamBlockDesc2](#).

```
class PBValidator : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

A pointer to an instance of this class is a data member of struct **ParamDef** and is also used by the **p\_validator** tag in a **ParamBlockDesc2** constructor. Any parameter can have a custom validator. This is used by the scripter and node pick button filter for example. If you want to validate a parameter block 2 value create an instance of this class and implement the **Validate()** method.

### Methods:

public:

### Prototype:

```
virtual BOOL Validate(PB2Value& v)=0;
```

### Remarks:

Returns TRUE if the given PB2Value is valid; otherwise FALSE.

### Parameters:

**PB2Value& v**

The value to check.

### Prototype:

```
virtual BOOL Validate(PB2Value& v, ReferenceMaker* owner,  
ParamID id, int tabIndex);
```

### Remarks:

This method is available in release 4.0 and later only.

A variant of **Validate()** method to **PBValidate** which supplies more context than the original **Validate()**, effectively giving the same context information as the **Set()** & **Get()** methods in a **PBAccessor**.

### Parameters:

**PB2Value& v**

The value to check.

**ReferenceMaker\* owner**

Points to the owner of the parameter.

**ParamID id**

The permanent ID of the parameter.

**int tabIndex**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

**Prototype:**

**virtual void DeleteThis();**

**Remarks:**

This method that can be used to destroy dynamically allocated instances of this class.

**Default Implementation:**

```
{ }
```

## Class PBAccessor

See Also: [Structure ParamDef](#), [List of ParamType2 Choices](#), [Class ReferenceMaker](#), [Structure PB2Value](#).

```
class PBAccessor : public InterfaceServer
```

### Description:

This class is available in release 3.0 and later only.

Any parameter in a block can have an accessor callback object that has its **Get()** or **Set()** method called whenever the parameter is accessed. This may be used to provide access to dynamically-computed virtual parameters and sometimes to allow parameter-specific processing by the class as the parameter in the block is modified (such as keeping object data members up-to-date).

The **Get()** and **Set()** methods are called at all times when a parameter is accessed, including parameters that are animated. The **Get()** method is called after the controller is accessed, so the current controller value is seen and can be optionally overridden in the **Get()** method. Note that the controller is accessed whenever the 3ds max time is changed (such as a frame slider move) and so the **Get()** method will be called each frame as this happens.

A pointer to an instance of this class is a data member of the **ParamDef** structure.

### Methods:

public:

### Prototype:

```
virtual void Get(PB2Value& v, ReferenceMaker* owner, ParamID id, int tabIndex, TimeValue t, Interval &valid);
```

### Remarks:

This method is called when the specified owner object accesses (gets) the value v. This is called after the controller is accessed, so the current controller value is seen and can be optionally overridden in the this method. Note that the controller is accessed whenever the 3ds max time is changed (such as a slider move) and so this method will be called each frame as this happens.

### Parameters:

**PB2Value& v**

The value being accessed.

**ReferenceMaker\* owner**

Points to the owner of the parameter.

**ParamID id**

The permanent ID of the parameter.

**int tabIndex**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

**TimeValue t**

The current time the get is taking place.

**Interval &valid**

The validity interval for the value.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void Set(PB2Value& v, ReferenceMaker* owner, ParamID id, int tabIndex TimeValue t);
```

**Remarks:**

This method is called when the specified owner objects sets the value v. This is called just before calling **SetValue()** on the parameters controller, so it can take note of the value going in and change it if desired.

**Parameters:**

**PB2Value& v**

The value being set.

**ReferenceMaker\* owner**

Points to the owner of the parameter.

**ParamID id**

The permanent ID of the parameter.

**int tabIndex**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

**TimeValue t**

The current time the set is taking place.

**Default Implementation:**

```
{
```

**Function:**

```
virtual void TabChanged(tab_changes changeCode,  
Tab<PB2Value>* tab, ReferenceMaker* owner, ParamID id, int  
tabIndex, int count);
```

**Remarks:**

This method is available in release 4.0 and later only.

This method is called when a **Tab**<> parameter has a change made to its table structure.

**Parameters:**

**tab\_changes changeCode**

Describes the change that has just occurred to the **Tab**<> parameter. One of the following enumerations:

```
enum tab_changes { tab_insert, tab_append, tab_delete,  
tab_ref_deleted, tab_setcount, tab_sort };
```

**Tab<PB2Value>\* tab**

Points to the actual **Tab**<> in the pblock parameter.

**ReferenceMaker\* owner**

Points to the owner of the parameter.

**ParamID id**

The permanent ID of the parameter.

**int tabIndex**

The start index of the change (for **tab\_insert**, **tab\_append**, **tab\_delete**, **tab\_ref\_deleted**)

**int count**

The number of elements changed (for **tab\_insert**, **tab\_append**, **tab\_delete**).

**Prototype:**

**virtual BOOL KeyFrameAtTime(ReferenceMaker\* owner,  
ParamID id, int tabIndex, TimeValue t);**

**Remarks:**

Checks to see if a keyframe exists for the given parameter at the given time. Returns TRUE if a keyframe exists at the specified time; otherwise FALSE. For parameters not directly hosted in the parameter block that are internally animatable, this provides a keyframe query callback so that any ParamMap2 spinners associated with these 'virtual' parameters can show keyframe status for the underlying parameter. In these cases, developers should implement this method for the parameter usually asking the underlying parameter for its keyframe state.

**Parameters:**

**ReferenceMaker\* owner**

Points to the owner of the parameter.

**ParamID id**

The permanent ID of the parameter.

**int tabIndex**

**TimeValue t**

The current time.

**Default Implementation:**

```
{ return FALSE; }
```

**Prototype:**

**virtual TSTR GetLocalName(ReferenceMaker\* owner, ParamID  
id, int tabIndex);**

**Remarks:**

This allows a plug-in to provide a dynamically-created local name for a parameter or Tab<> parameter entry. If you specify the **P\_COMPUTED\_NAME** parameter flag, you also need to supply a **p\_accessor PBAccessor** instance pointer that has this method implemented.

**Parameters:**

**ReferenceMaker\* owner**

Points to the owner of the parameter.

**ParamID id**

The permanent ID of the parameter.

**int tabIndex**

If the parameter is a Tab<> this is the zero based index of the parameter in the table.

**Default Implementation:**

```
{ return _T(""); }
```

**Prototype:**

```
virtual void DeleteThis();
```

**Remarks:**

This method that can be used to destroy dynamically allocated instances of this class.

**Default Implementation:**

```
{ }
```

## Class PBitmap

See Also: [Class IParamBlock2](#), [Class Bitmap](#), [Class BitmapInfo](#), [Class BitmapManager](#).

```
class PBitmap
```

### Description:

This class is available in release 3.0 and later only.

This is a Bitmap/BitmapInfo wrapper class. It is used by ParamBlock2s to store bitmap information. The class has two public data members that hold the BitmapInfo and the Bitmap itself.

### Data Members:

public:

**BitmapInfo bi;**

Stores the BitmapInfo for the bitmap.

**Bitmap \*bm;**

Points to the Bitmap itself.

### Methods:

public:

### Prototype:

**PBitmap(BitmapInfo &bi);**

### Remarks:

Implemented by the System

Constructor. The BitmapInfo data member is initialized to the BitmapInfo passed. The Bitmap pointer is set to NULL.

### Prototype:

**PBitmap();**

### Remarks:

Implemented by the System

Constructor. The Bitmap pointer is set to NULL.

**Prototype:**

**~PBBitmap();**

**Remarks:**

Implemented by the System

Destructor. The Bitmap, if allocated, is deallocated.

**Prototype:**

**void Load();**

**Remarks:**

Implemented by the System

The BitmapManager is used to Load the bitmap as specified by the BitmapInfo.

**Prototype:**

**PBBitmap\* Clone();**

**Remarks:**

Implemented by the System

Makes a copy of the bitmap and returns a pointer to it.

## Class ParamBlock2PLCB

See Also: [Class PostLoadCallback](#), [Class ILoad](#).

```
class ParamBlock2PLCB : public PostLoadCallback
```

### Description:

This class is available in release 3.0 and later only.

This is a post load call back for fixing up parameter block2s. This callback handles conversion of pre-ParamBlock2 versions of an object to a ParamBlock2 version. NOTE: this thing deletes itself when it's done.

### Data Members:

All data members are public.

#### **ParamVersionDesc\* versions;**

This is an array of ParamVersionDesc2s.

#### **int count;**

This is the number in the array specified above.

#### **ParamBlockDesc2\* curdesc;**

This is a pointer to the current version of the description.

#### **ReferenceTarget\* targ;**

This is a pointer to a reference target. This is usually the this pointer of the object.

#### **int pbRefNum;**

This is the reference index of the parameter block.

### Methods:

public:

### Prototype:

```
ParamBlock2PLCB(ParamVersionDesc *v, int cnt,  
ParamBlockDesc2* pd, ReferenceTarget *t, int refNum);
```

### Remarks:

Constructor. The data members are initialized to the values passed.

### Prototype:

**void proc(ILoad \*iload);**

**Remarks:**

This methods handles the conversion of edition 1 ParamBlocks to the new ParamBlock2s.

**Parameters:**

**ILoad \*iload**

An interface for loading files.

## Structure ParamAlias

See Also: [Class IParamBlock2](#), [Class ParamBlockDesc2](#).

### Description:

This structure is available in release 3.0 and later only.

This structure provides information about a parameter alias. Aliases allow individual parameters or Tab<> parameter elements to be named. See the methods **IParamBlock2::DefineParamAlias**, **FindParamAlias**, etc.

### typedef struct

{

**TCHAR\* alias;**

The name of the alias.

**ParamID ID;**

The permanent ID of the parameter.

**int tabIndex;**

If the parameter is a Tab<> this is the zero based index into the table of the parameter. If the parameter is not a table this is -1.

**} ParamAlias;**

## Class IAutoEParamDlg

See Also: [Class ParamDlg](#), [Class IParamMap2](#), [Class IRendParams](#), [Class ParamMap2UserDlgProc](#).

```
class IAutoEParamDlg : public EffectParamDlg
```

### Description:

This class is available in release 3.0 and later only.

Auto ParamDlg class for Effects auto-UI, instanced by

**ClassDesc2::CreateParamDialog()**. It maintains a table of secondary EffectParamDlg for master EffectParamDlg (e.g., the one returned from CreateParamDialog()) and will broadcast appropriate method calls to them as the master receives them.

### Methods:

public:

### Prototype:

```
virtual void InvalidateUI()=0;
```

### Remarks:

This method causes the user interface controls to be re-drawn.

### Prototype:

```
virtual int NumDlgs()=0;
```

### Remarks:

Returns the number of secondary dialogs.

### Prototype:

```
virtual void AddDlg(SFXParamDlg* dlg)=0;
```

### Remarks:

Adds the specified dialog as another secondary dialog.

### Parameters:

**SFXParamDlg\* dlg**

Points to the parameter dialog to add.

**Prototype:**

**virtual SFXParamDlg\* GetDlg(int i)=0;**

**Remarks:**

Returns a pointer to the 'i-th' secondary dialog.

**Parameters:**

**int i**

The zero based index of the dialog to return.

**Prototype:**

**virtual void SetDlg(int i, SFXParamDlg\* dlg)=0;**

**Remarks:**

Sets the 'i-th' dialog to the one passed.

**Parameters:**

**int i**

The zero based index of the dialog to set.

**SFXParamDlg\* dlg**

Points to the parameter dialog to set.

**Prototype:**

**virtual void DeleteDlg(SFXParamDlg\* dlg)=0;**

**Remarks:**

This method is used for deleting secondary dialogs from a master **IAutoEParamDlg**. Use this along with **AddDlg()** if you are dynamically changing the set of rollups for the plugin, so that the **P\_AUTO\_UI** system can correctly manage all current secondary rollups.

**Parameters:**

**SFXParamDlg\* dlg**

Points to the ParamDlg to delete.

**Prototype:**

**virtual IParamMap2\* GetMap()=0;**

**Remarks:**

Returns a pointer to the parameter map2 of this primary (master) dialog.

**The following function is not a method of this class but is available for use:**

**Function:**

**IAutoEParamDlg\* CreateAutoEParamDlg(IRendParams \*i, Effect\* e, IParamBlock2\* pb, ClassDesc2\* cd, HINSTANCE inst, TCHAR\* dlgTemplate, TCHAR\* title, int rollFlags, ParamMap2UserDlgProc\* dlgProc=NULL);**

**Remarks:**

This function creates an AutoEParamDlg for render effects.

**Parameters:**

**IRendParams \*i**

An interface pointer for rendering effects.

**Effect\* e**

Points to the rendering effect calling this function.

**IParamBlock2\* pb**

Points to the parameter block instance associated with the parameter map.

**ClassDesc2\* cd**

The class descriptor2 for the plug-in creating the parameter map.

**HINSTANCE inst**

The plug-ins DLL instance handle.

**TCHAR\* dlgTemplate**

Dialog template for the rollup page (created using the resource editor)

**TCHAR\* title**

The title displayed in the rollup page title bar.

**int rollFlags**

A set of flags to control settings of the rollup page.

**APPENDROLL\_CLOSED**

Starts the page in the rolled up state.

**ParamMap2UserDlgProc\* dlgProc=NULL**

If there is some custom handling required by a particular control, the client

can derive a class from ParamMap2UserDlgProc and set it as the parameter map's user callback.

**Return Value:**

A pointer to an interface for managing the parameter map2.

## Class ParamMap2UserDlgProc

See Also: [Class ParamBlockDesc2](#), [Class ClassDesc2](#).

class ParamMap2UserDlgProc

### Description:

This class is available in release 3.0 and later only.

This class is used with the release 3 parameter map system. If there are controls which require custom handling you can create an object from this class and set it as the parameter map's user callback (usually using **SetUserDialogProc()**).

### Methods:

public:

### Prototype:

**virtual BOOL DlgProc(TimeValue t, IParamMap2 \*map, HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)=0;**

### Remarks:

Implemented by the Plug-In.

This is the dialog proc that will be called to process the control messages. This proc will be called after the default processing is complete.

### Parameters:

#### **TimeValue t**

This is the current time.

#### **IParamMap2 \*map**

This is a pointer to the parameter map.

#### **HWND hWnd**

This is the handle to the dialog.

#### **UINT msg**

This is the message that should be processed by the dialog proc.

#### **WPARAM wParam**

This is a parameter that holds message specific information.

#### **LPARAM lParam**

This is a parameter that holds message specific information.

### Return Value:

This is essentially the equivalent of a normal Windows dialog proc, so it should return whatever value a normal dialog proc returns for the message. An exception is that the value **REDRAW\_VIEWS** may be returned to cause the viewports to be redrawn.

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

Implemented by the Plug-In.

This method is called to delete this object. If the DlgProc is non-NULL when the ParamMap is deleted this method will be called.

**Prototype:**

**virtual void SetThing(ReferenceTarget \*m);**

**Remarks:**

Implemented by the Plug-In.

This gets called if the **DlgProc** is registered with a parameter map associated with one of the **IAutoXXXParamDlg** instances and that dialog has **SetThing()** called on it. This provides a simple way for the user dialog proc to track changes to the underlying object in the parameter map.

**Parameters:**

**ReferenceTarget \*m**

The item that was set.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void Update(TimeValue t, Interval& valid, IParamMap2\* pmap);**

**Remarks:**

This method is available in release 4.0 and later only.

This is a variant of **Update()** method in **ParamMap2UserDlgProc** used to

supply more context, including the pmap itself and its validity interval.  
Changes you make to the validity interval passed in affect the validity interval of the whole parammap.

**Parameters:**

**TimeValue t**

The time at which the update is taking place.

**Interval& valid**

The validity interval of the parameter map.

**IParamMap2 \*pmap**

The parameter map the user dialog proc is associated with.

## Bitmap Flags

See Also: [Class BitmapStorage](#).

The following flag bits describe properties of the bitmap:

### **MAP\_READY**

This bitmap has had memory allocated to it, or is accessible directly.

### **MAP\_HAS\_ALPHA**

The bitmap has an alpha channel. This flag can be checked from a **BitmapInfo** instance (**bi.Flags() & MAP\_HAS\_ALPHA**), but not from a **Bitmap** instance (**bmap->Flags() & MAP\_HAS\_ALPHA**). Therefore if you have a **Bitmap**, use the **Bitmap's HasAlpha()** method to see if the map has an alpha channel.

### **MAP\_ALPHA\_PREMULTIPLIED**

The bitmap has pre-multiplied alpha.

### **MAP\_PALETTED**

The bitmap uses a palette (not true color).

### **MAP\_DITHERED**

The bitmap is dithered.

### **MAP\_FLIPPED**

The bitmap is flipped horizontally.

### **MAP\_INVERTED**

The bitmap is inverted vertically.

### **MAP\_USE\_SCALE\_COLORS**

This option is available in release 4.0 and later only.

The bitmap should scale colors when high dynamic range values are out of gamut.

### **MAP\_LEGAL\_DELETE**

This flag is for internal use only.

### **MAP\_VIEW\_FILTERED**

This flag is used for testing internally.

### **MAP\_FRAME\_SYSTEM\_LOCKED**

This flag is no longer used.

### **MAP\_NOFLAGS**

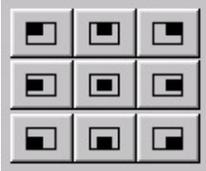
The bitmap has none of the characteristics below.

**MAP\_ALL\_FLAGS**

Indicates ALL the flags are set.

## List of Bitmap Alignment Positions

One of the following nine values define the position of the bitmap:



1 | 2 | 3

----+----+----

4 | 5 | 6

----+----+----

7 | 8 | 9

**BMM\_CUSTOM\_POSNW**

Top Left Position (1)

**BMM\_CUSTOM\_POSN**

Top Center Position (2)

**BMM\_CUSTOM\_POSNE**

Top Right Position (3)

**BMM\_CUSTOM\_POSW**

Middle Left Position (4)

**BMM\_CUSTOM\_POSCN**

Middle Center Position (5)

**BMM\_CUSTOM\_POSE**

Middle Right Position (6)

**BMM\_CUSTOM\_POSSW**

Bottom Left Position (7)

**BMM\_CUSTOM\_POSS**

Bottom Center Position (8)

**BMM\_CUSTOM\_POSSE**

Bottom Right Position (9)

## List of Custom Bitmap Flags

The Custom Bitmap Flags. These may be ORed together as in:

**BMM\_CUSTOM\_GAMMA | BMM\_CUSTOM\_SIZE**

One or more of the following flags may be used.

**BMM\_CUSTOM\_GAMMA**

Custom gamma setting.

**BMM\_CUSTOM\_SIZE**

Custom size setting.

**BMM\_CUSTOM\_RESFIT**

Bitmap is to be resized.

**BMM\_CUSTOM\_POS**

Bitmap has a custom positioning.

**BMM\_CUSTOM\_FILEGAMMA**

Bitmap has a custom file gamma setting.

# Class CropCallback

See Also: [Class Bitmap](#).

class CropCallback

## Description:

This class is available in release 2.0 and later only.

This class is a callback for interactive adjustment of bitmap "Cropping rectangle", passed in as an argument to the **Bitmap::Display()** method. See `\MAXSDK\SAMPLES\MATERIALS\BMTEX.CPP` for sample code.

All methods of this class are implemented by the plug-in.

## Methods:

### Prototype:

**virtual float GetInitU()=0;**

### Remarks:

Returns the initial U value.

### Prototype:

**virtual float GetInitV()=0;**

### Remarks:

Returns the initial V value.

### Prototype:

**virtual float GetInitW()=0;**

### Remarks:

Returns the initial W value.

### Prototype:

**virtual float GetInitH()=0;**

### Remarks:

Returns the initial H value.

**Prototype:**

**virtual BOOL GetInitMode()=0;**

**Remarks:**

Returns TRUE for place mode; FALSE for crop mode. In crop mode the image is clipped at the edges. In place mode, the image is resized or moved.

**Prototype:**

**virtual void SetValues(float u, float v, float w, float h, BOOL md)=0;**

**Remarks:**

This method is called to set the values as the user is making adjustments. If the parameters may be animated, use **Interface::GetTime()** to retrieve the time they are being set for.

**Parameters:**

**float u**

The U parameter to set.

**float v**

The V parameter to set.

**float w**

The W parameter to set.

**float h**

The H parameter to set.

**BOOL md**

The mode being set. TRUE is place mode; FALSE is crop.

**Prototype:**

**virtual void OnClose()=0;**

**Remarks:**

This method is called when the cropping adjustment window is closed.

# Class BitmapNotify

See Also: [Class Bitmap](#), [Class BitmapStorage](#).

```
class BitmapNotify
```

## Description:

This class is available in release 2.0 and later only.

This class is a callback for notifying bitmaps that their storage has changed, and if any on screen displays need to be refreshed. This is installed as a callback via the method **Bitmap::SetNotify()**.

All methods of this class are implemented by the system.

## Methods:

### Prototype:

```
virtual int Changed(ULONG flags)=0;
```

### Remarks:

This method is called when the storage for the Bitmap has changed.

### Parameters:

**ULONG flags**

One of the following:

**BMNOTIFY\_FLAG\_STORAGE\_CHANGE**, notifies that the storage (the contents of the bitmap) has changed.

**BMNOTIFY\_FLAG\_FILE\_CHANGE**, notifies that that bitmap file has changed, probably by an external program. The bitmap should be reloaded. Note that by the time this call is made, the API has already checked to see if the user has set the global preferences asking for these changes to be automatically reloaded.

### Prototype:

```
virtual void VFBClosed();
```

### Remarks:

This method is called when Virtual Frame Buffer is closed.

### Default Implementation:

```
{}
```

## Class GWinSetup

See Also: [Class GraphicsWindow](#).

class GWinSetup

### **Description:**

This class is the graphics window setup structure. An instance of this class is passed to the function **createGW()** to create a new graphics window.

Note: This is no longer available for use by plug-ins in 3ds max 2.0 and later.

## List of Rendering Limits

See [Class GraphicsWindow](#).

These are the flags for specifying rendering limits. One or more of the following values:

### **GW\_NO\_ATTS**

Not attributes are specified.

### **GW\_WIREFRAME**

Wireframe rendering mode.

### **GW\_ILLUM**

This indicates that you have colors per vertex in your polygons and that they should be used. If you had colors per vertex but this flag was not set, the colors would be ignored.

### **GW\_PERSPECTIVE\_CORRECT**

In this mode textures are corrected for perspective display.

### **GW\_POLY\_EDGES**

This option is available in release 2.0 and later only.

This mode causes polygon edges (Edged Faces) to be on.

### **GW\_FLAT**

Flat (facet) shading mode.

### **GW\_SPECULAR**

This enables specular highlight display.

### **GW\_TEXTURE**

This enables texture display.

### **GW\_Z\_BUFFER**

When coordinates are specified for drawing primitives they have x, y, and z values. Sometimes when drawing entities in the viewports it is desirable to ignore the z values. For example in the 3ds max viewports the text that display the type of viewport (Front, Left, ...) are drawn without z values. So are the arc-rotate circle control and the axis tripods. These items are drawn without this flag being set so they always show up in front.

### **GW\_BACKCULL**

Backface culling is used. Entities whose surface normal face away from the view direction are not drawn.

## **GW\_TWO\_SIDED**

Faces are displayed regardless of their surface normal orientation.

## **GW\_COLOR\_VERTS**

This option is available in release 2.0 and later only.

This turns on color-per-vertex display.

## **GW\_SHADE\_CVERTS**

This option is available in release 2.0 and later only.

This modifies **GW\_COLOR\_VERTS**. If set, then lighting is enabled and the vertex colors are used to modulate the colors that result from lighting. If off, then the colors on each vertex are used directly to shade the triangle. (Note that when 3ds max uses **GW\_SHADE\_CVERTS** mode, it puts a white diffuse-only material on the object so that it appears that the colors are shaded without distortion.)

Described further, when shading is OFF, then the vertex colors are used directly. This is equivalent to saying that they are modulated by a pure white self-illuminated material (i.e. the color values are "modulated (multiplied) by 1" -- so they don't change).

When shading is ON, the diffuse white material is illuminated by the scene lighting, resulting in shades ranging from black to white (0 to 1), with most vertices being some shade of pure gray. When the vertex colors are modulated by the material color, they get multiplied (in general) by a number less than 1, which makes them appear darker.

The RGB components of the colors are modulated uniformly, so that there is no shift from, say, red to green. That would happen if the underlying material was not evenly weighted (i.e. a pure gray lying between black and white). Said another way, only the intensity of the vertex colors is changed when shading is on, not luminance, chrominance, etc.

## **GW\_PICK**

This indicates hit testing will be performed (not rendering).

## **GW\_BOX\_MODE**

Objects are shown using their bounding box.

## **GW\_ALL\_EDGES**

All edges of the item are shown (including hidden ones).

## **GW\_VERT\_TICKS**

This option is available in release 2.0 and later only.

This mode is really a pseudo-mode, in that it doesn't actually cause GFX do anything differently, but rather is tested by the Mesh class, which sends down vertex markers (+) if the mode is on.

### **GW\_LIGHTING**

This is the same as (**GW\_ILLUM** | **GW\_SPECULAR**)

### **GW\_TRANSPARENCY**

Specifies to use Transparency

### **GW\_TRANSPARENT\_PASS**

Specifies a Second pass to perform Blended Transparency

## Clip Flags

View volume clip flags. Flags are provided for each of the six planes bounding the viewing frustum. One or more of the following values:

**GW\_LEFT\_PLANE**

**GW\_RIGHT\_PLANE**

**GW\_BOTTOM\_PLANE**

**GW\_TOP\_PLANE**

**GW\_FRONT\_PLANE**

**GW\_BACK\_PLANE**

## List of Mouse Callback Messages

This message describes the type of event that occurred.

### **MOUSE\_ABORT**

When the user aborts a mouse procedure, for example when they are dragging the mouse and they right click, this message is sent.

### **MOUSE\_IDLE**

This message is used internally.

### **MOUSE\_POINT**

This message is sent when the user has clicked a point.

### **MOUSE\_MOVE**

This message is sent when the mouse input is captured and the user moved the mouse. When mouse input is captured all mouse events continue to go to the current window even when the mouse is move outside the limits of the window. This is when the user is 'dragging'.

### **MOUSE\_DBLCLICK**

This is sent when the user has double clicked the mouse.

### **MOUSE\_INIT**

This is sent when the mouse proc is plugged in as the current mouse proc. If a plug-in needed to perform some kind of initialization when it was first became current this message could be processed.

### **MOUSE\_UNINIT**

This is sent when the mouse proc is un-plugged as the current mouse proc.

### **MOUSE\_FREEMOVE**

This message is similar to a **MOUSE\_MOVE** message except it is not called when the mouse is in a 'drag' session. This means that mouse input is not captured. If mouse input is not captured and the mouse is moved outside the current window, the current window will no longer receive the mouse messages.

### **MOUSE\_KEYBOARD**

This is not used. Keyboard input is processed by registering an accelerator table. See [Class Interface](#) for the methods to register a keyboard accelerator. Also see the section [Keyboard Accelerators and Dialog Messages](#).

### **MOUSE\_PROPCLICK**

This message is sent on a right click, when nothing is selected, and the user is not over any selectable object. For example, this is how the unfreeze-by-hit pick mode knows to abort if the user presses the right mouse button. Note that this is different than if you right click while dragging - in that case you get a **MOUSE\_ABORT** message.

## Class ICustStatusEdit

See Also: [Class ICustomControl](#).

```
class ICustStatusEdit : public ICustomControl
```

### Description:

This class is available in release 3.0 and later only.

This control mimics the edit control as well as a status control. It may be set to 'read-only' so the user can read but cannot edit the displayed string.

The value to use in the Class field of the Custom Control Properties dialog is:

**CustStatusEdit**

### Function:

```
ICustStatusEdit *GetICustStatusEdit(HWND hCtrl);
```

### Remarks:

This global function is available in release 3.0 and later only.

Used to initialize and return a pointer to the control.

### Parameters:

**HWND hCtrl**

The window handle of the control.

### Function:

```
void ReleaseICustStatusEdit(ICustStatusEdit *ice);
```

### Remarks:

This global function is available in release 3.0 and later only.

Used to release the control when finished.

### Parameters:

**ICustStatusEdit \*ice**

Points to the control to release.

### Methods:

public:

### Prototype:

```
virtual void GetText(TCHAR *text, int ct)=0;
```

**Remarks:**

Retrieves the text entered into the control.

**Parameters:**

**TCHAR \*text**

Storage for the text to retrieve.

**int ct**

Specifies the maximum length of the string returned.

**Prototype:**

**virtual void SetText(TCHAR \*text)=0;**

**Remarks:**

This method places the text into the control for editing.

**Parameters:**

**TCHAR \*text**

The text to place in the control.

**Prototype:**

**virtual void SetText(int i)=0;**

**Remarks:**

This method allows you to pass in an integer value to the control. The integer is converted to a string and displayed in the control.

**Parameters:**

**int i**

This value is converted to a string and displayed in the control.

**Prototype:**

**virtual void SetText(float f, int precision=3)=0;**

**Remarks:**

This method allows you to pass in a floating point value to the control. The float is converted to a string and displayed in the control.

**Parameters:**

**float f**

This value is converted to a string and displayed in the control.

### **int precision=3**

The precision argument is simply the number of decimal places that get represented in the string that appears in the edit field. So if the arguments were (1.0f/3.0f, 3) then the string "0.333" would appear in the edit field.

#### **Prototype:**

**virtual int GetInt(BOOL \*valid=NULL)=0;**

#### **Remarks:**

This method parses and returns an integer value from the control.

#### **Parameters:**

**BOOL \*valid=NULL**

This pointer, if passed, is set to TRUE if the input is 'valid'; otherwise FALSE. FALSE indicates that something caused the parsing of the input to terminate improperly. An example is a non-numeric character. So for example, if the user entered "123jkfk sdf" into the field the valid pointer would be set to FALSE.

#### **Prototype:**

**virtual float GetFloat(BOOL \*valid=NULL)=0;**

#### **Remarks:**

This method parses and returns a floating point value from the control.

#### **Parameters:**

**BOOL \*valid=NULL**

This pointer, if passed, is set to TRUE if the input is 'valid'; otherwise FALSE. FALSE indicates that something caused the parsing of the input to terminate improperly. An example is a non-numeric character. So for example, if the user entered "123jkfk sdf" into the field this pointer would be set to FALSE.

#### **Prototype:**

**virtual void SetLeading(int lead)=0;**

#### **Remarks:**

A developer doesn't normally need to call this method. This offsets the text

vertically in the edit control.

**Parameters:**

**int lead**

This parameter specifies the number of pixels to offset.

**Prototype:**

**virtual void WantReturn(BOOL yesNo)=0;**

**Remarks:**

This method allows custom handling of the RETURN key. If you pass TRUE to this method an EN\_CHANGE message will be sent to the control when the RETURN key is pressed. The EN\_CHANGE message is sent when the user has taken any action that may have altered text in an edit control so developer need to also call GotReturn() (documented below) to see if it was indeed a RETURN keypress.

**Parameters:**

**BOOL yesNo**

If TRUE, then when the user presses the RETURN key in that control, the edit field will send an EN\_CHANGE message to the owner, and calling GotReturn() will return TRUE.

**Prototype:**

**virtual BOOL GotReturn()=0;**

**Remarks:**

This method should be called on receipt of an EN\_CHANGE message. It return TRUE if pressing the RETURN key generated the message; otherwise FALSE.

**Prototype:**

**virtual void GiveFocus()=0;**

**Remarks:**

Calling this method gives the control the focus to receive input.

**Prototype:**

**virtual BOOL HasFocus()=0;**

**Remarks:**

Returns TRUE if the control has the focus to receive input; otherwise FALSE.

**Prototype:**

**virtual void WantDlgNextCtl(BOOL yesNo)=0;**

**Remarks:**

Determines whether the TAB key may be used to jump to the next control in the tab sequence.

**Parameters:**

**BOOL yesNo**

TRUE to enable the TAB key to move to the next control; FALSE to disable the TAB key from moving the focus.

**Prototype:**

**virtual void SetNotifyOnKillFocus(BOOL onOff)=0;**

**Remarks:**

Normally when a user exits an edit field the notification WM\_CUSTEDIT\_ENTER is sent. Many plug-ins key off this message to finalize the input of values. For instance, if the user is entering a value into an edit field and they hit the TAB key to leave the field the value should be entered. Normally this is the desired behavior. However, as a special case condition, if a developer does not want to update the value, this method may be called so the WM\_CUSTEDIT\_ENTER notification won't be sent when the edit control loses focus.

**Parameters:**

**BOOL onOff**

TRUE to turn on; FALSE to turn off.

**Prototype:**

**virtual void SetBold(BOOL onOff)=0;**

**Remarks:**

Sets the text font in the edit control to display in a bold format or normal.

**Parameters:**

**BOOL onOff**

TRUE to turn bolding on; FALSE to turn off.

**Prototype:**

**virtual void SetReadOnly(BOOL onOff)=0;**

**Remarks:**

Sets if the control is 'read-only'. That is, the string is displayed but cannot be edited.

**Parameters:**

**BOOL onOff**

TRUE for read-only; FALSE to allow editing.

# Class ToolItem

See Also: [Custom Controls](#), [Class ToolImageItem](#).

class ToolItem

## Description:

This class describes the properties of an item in a 3ds max custom toolbar.

## Data Members:

public:

**ToolItemType type;**

See [List of Tool Item Types](#).

**int id**

The ID for the control.

**DWORD helpID**

For plug-in developers this id should be set to 0. Basically, the main 3ds max help file contains help tags that are tied to various parts of the 3ds max UI, allowing the right help page to come up when UI help is requested. In particular, if you press the ? button on the toolbar, then press another toolbar button, you'll get help on that button's functionality. This is because internally pressing the button yields a help ID that indexes into the help file. But since the same help ID must be compiled into the help file and into MAX, and since the main 3ds max helpfile can not be rebuilt by developers, they cannot use this functionality.

**int w**

The width of the button image.

**int h**

The height of the button image.

**int orient;**

This data member is available in release 3.0 and later only.

The orientation of the item. One of the following values:

**CTB\_HORIZ**

**CTB\_VERT**

**CTB\_FLOAT**

**Methods:**

**Prototype:**

**virtual ~ToolItem()**

**Remarks:**

Destructor.

## Class ToolButtonItem

See Also: [Class ToolItem](#), [Class MAXBmpFileIcon](#), [Custom Controls](#).

```
class ToolButtonItem : public ToolItem
```

### Description:

This class describes the properties of a 3ds max custom toolbar button.

Note, you can use MaxBmpFileIcons on a toolbar button using the new constructor in custcont.h:

```
ToolButtonItem(ToolItemType t,  
               MaxBmpFileIcon* pIcon,  
               int iW, int iH, int wd,int ht,  
               int ID, DWORD hID=0, TCHAR *lbl = NULL,  
               int or = CTB_HORIZ|CTB_VERT|CTB_FLOAT)
```

### Data Members:

public:

The following four data members (**iOutEn**, **iInEn**, **iOutDis**, **iInDis**) are indices into the image list. They indicate which images to use for each of the four possible button states. You may specify a unique image for each one of these states by passing a different index for each state. Or you may supply a single image to be used for all the states by specifying the same index four times.

**int iOutEn**

Out&Enabled.

**int iInEn**

In&Enabled.

**int iOutDis**

Out&Disabled.

**int iInDis**

In&Disabled.

**int iw**

The width of the button image.

**int ih**

The height of the button image.

**TCHAR \*label;**

This data member is available in release 4.0 and later only.

The label describing the tool button item.

**MaxBmpFileIcon \*mpIcon;**

This data member is available in release 4.0 and later only.

A pointer to the icon image associated with the button.

**MaxBmpFileIcon \*mpInIcon;**

This data member is available in release 4.0 and later only.

A pointer to the in icon image associated with the button.

**Methods:**

**Prototype:**

**ToolButtonItem(ToolItemType t, int iOE, int iIE, int iOD, int iID, int iW, int iH, int wd, int ht, int ID, DWORD hID=0, TCHAR \*lbl = NULL, int or = CTB\_HORIZ|CTB\_VERT|CTB\_FLOAT)**

**Remarks:**

Constructor.

**Parameters:**

**ToolItemType t**

See [List of Tool Item Types](#).

**int iOE**

The Out&Enabled index.

**int iIE**

The In&Enabled index.

**int iOD**

The Out&Disabled index.

**int iID**

The In&Disabled index.

**int iW**

The image width (size of the bitmap in the ImageList).

**int iH**

The image height (size of the bitmap in the ImageList).

**int wd**

The width of the button.

**int ht**

The height of the button.

**int ID**

The ID of the control.

**DWORD hID=0**

The help ID. For plug-in developers this id should be set to 0.

**TCHAR \*lbl = NULL**

The label of the button.

**int or = CTB\_HORIZ|CTB\_VERT|CTB\_FLOAT**

The allowable orientation of the item. This may be one or more of the following:

**CTB\_HORIZ** - Horizontal

**CTB\_VERT** - Vertical

**CTB\_FLOAT** - Floating (not docked)

**Prototype:**

**ToolButtonItem(ToolItemType t, MaxBmpFileIcon\* pIcon, int iW, int iH, int wd, int ht, int ID, DWORD hID=0, TCHAR \*lbl = NULL, int or = CTB\_HORIZ|CTB\_VERT|CTB\_FLOAT);**

**Remarks:**

This method is available in release 4.0 and later only.

Constructor.

**Parameters:**

**ToolItemType t**

See [List of Tool Item Types](#).

**MaxBmpFileIcon\* pIcon**

A pointer to the icon associated with the button.

**int iW**

The image width (size of the bitmap in the ImageList).

**int iH**

The image height (size of the bitmap in the ImageList).

**int wd**

The width of the button.

**int ht**

The height of the button.

**int ID**

The ID of the control.

**DWORD hID=0**

The help ID. For plug-in developers this id should be set to 0.

**TCHAR \*lbl = NULL**

The label of the button.

**int or = CTB\_HORIZ|CTB\_VERT|CTB\_FLOAT**

The orientation of the button item.

#### **Prototype:**

```
ToolButtonItem(ToolItemType t, MaxBmpFileIcon* pIcon,  
MaxBmpFileIcon* pInIcon, int iW, int iH, int wd,int ht, int ID,  
DWORD hID=0, TCHAR *lbl = NULL, int or =  
CTB_HORIZ|CTB_VERT|CTB_FLOAT);
```

#### **Remarks:**

This method is available in release 4.0 and later only.

Constructor.

#### **Parameters:**

**ToolItemType t**

See [List of Tool Item Types](#).

**MaxBmpFileIcon\* pIcon**

A pointer to the icon associated with the button.

**MaxBmpFileIcon\* pInIcon**

A pointer to the in icon associated with the button.

**int iW**

The image width (size of the bitmap in the ImageList).

**int iH**

The image height (size of the bitmap in the ImageList).

**int wd**

The width of the button.

**int ht**

The height of the button.

**int ID**

The ID of the control.

**DWORD hID=0**

The help ID. For plug-in developers this id should be set to 0.

**TCHAR \*lbl = NULL**

The label of the button.

**int or = CTB\_HORIZ|CTB\_VERT|CTB\_FLOAT**

The orientation of the button item.

## Class IRollupPanel

class IRollupPanel : public InterfaceServer

### Description:

This class is available in release 4.0 and later only.

This class represents the interface for a rollup panel and describes the properties of that panel (which is one rollup). You can obtain a pointer to the IRollupPanel class for any given specified window by calling

**IRollupWindow::IRollupPanel \*GetPanel(HWND hWnd);**

### Methods:

public:

### Prototype:

**virtual HINSTANCE GetHInst()=0;**

### Remarks:

This method returns a handle to the rollup panel instance.

### Prototype:

**virtual DWORD\_PTR GetResID()=0;**

### Remarks:

This method returns the resource ID of the rollup panel.

### Prototype:

**virtual BOOL operator==(const IRollupPanel& id)=0;**

### Remarks:

Equality test operator.

### Prototype:

**virtual int GetCategory()=0;**

### Remarks:

This method returns the rollup panel category identifier.

**Prototype:**

**virtual void SetCategory(int cat)=0;**

**Remarks:**

This method allows you to set the category identifier for the rollup panel.

**Parameters:**

**int cat**

The category identifier to set.

**Prototype:**

**virtual HWND GetHwnd()=0;**

**Remarks:**

This method returns a handle to the rollup window.

**Prototype:**

**virtual HWND GetRollupWindowHWND()=0;**

**Remarks:**

This method returns a handle to the actual panel in the rollup window.

**Prototype:**

**virtual HWND GetTitleWnd()=0;**

**Remarks:**

This method returns a handle to the window from which you can get the title through the **GWLP\_USERDATA**.

## Class BitArrayCallback

See Also: [Class BitArray](#).

class BitArrayCallback

### Description:

This class is available in release 3.0 and later only.

This is the callback object for the method **BitArray::EnumSet()**. The proc method is called for each "1" in the BitArray.

### Methods:

public:

### Prototype:

```
virtual void proc(int n)=0;
```

### Remarks:

This method is called for each "1" in the BitArray.

### Parameters:

**int n**

This is the zero based index into the BitArray of the element which is "1".

## Class GamConvert16

See Also: [Class GammaMgr](#), [Class GamConv8](#).

class GamConvert16

### Description:

A temporary table for converting 16->16. A developer may define an instance of this class and it will build a gamma correction table. The constructor will build the table with the specified gamma setting and the destructor will free the table. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**GamConvert16(float gam=1.0f);**

#### Remarks:

Constructor. The gamma table is built using the specified gamma setting.

#### Prototype:

**~GamConvert16();**

#### Remarks:

Destructor. The gamma table is deleted.

#### Prototype:

**void SetGamma(float gam);**

#### Remarks:

Sets the gamma setting to the value specified and builds the gamma table.

#### Parameters:

**float gam**

The gamma value to set.

#### Prototype:

**UWORD Convert(UWORD v);**

#### Remarks:

Gamma corrects the specified color.

**Parameters:**

**UWORD v**

The color to gamma correct.

**Return Value:**

The gamma corrected value.

## Class GamConvert8

See Also: [Class GammaMgr](#), [Class GamConvert16](#).

class GamConvert8

### Description:

A temporary table for converting 8->16. A developer may define an instance of this class and it will build a gamma correction table. The constructor will build the table with the specified gamma setting and the destructor will free the table. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**GamConvert8(float gam=1.0f);**

#### Remarks:

Constructor. The gamma table is built using the specified gamma setting.

#### Prototype:

**void SetGamma(float gam);**

#### Remarks:

Sets the gamma setting to the value specified and builds the gamma table.

#### Parameters:

**float gam**

The gamma value to set.

#### Prototype:

**UWORD Convert(UBYTE v);**

#### Remarks:

Gamma corrects the specified color.

#### Parameters:

**UBYTE v**

The color to gamma correct.

#### Return Value:

The gamma corrected value.

# Class FlagUser

See Also: [Class MNMesh](#).

Class FlagUser

## Description:

This class is available in release 2.0 and later only.

This is a handy class to subclass off of when you're designing a class with flags. It contains one private data member, a DWORD, with the flag info. It then implements a bunch of handy flag-related functions. All methods of this class are implemented by the system.

## Data Members:

private:

**DWORD FlagUserFlags;**

Stores the flags.

## Methods:

### Prototype:

**FlagUser();**

### Remarks:

Constructor. Sets FlagUserFlags to 0, clearing all flag bits.

### Prototype:

**void SetFlag(DWORD fl, bool val=TRUE);**

### Remarks:

Sets flags. Each bit that is set in **fl** is assigned the value **val**.

### Prototype:

**void ClearFlag(DWORD fl);**

### Remarks:

Clears flags. Each bit that is set in **fl** is cleared.

**Prototype:**

**bool GetFlag(DWORD fl) const;**

**Remarks:**

Checks if flags are set.

**Return Value:**

TRUE if any of the bits set in fl are set in this FlagUser. FALSE if none of them are.

**Prototype:**

**void ClearAllFlags();**

**Remarks:**

Sets FlagUserFlags to 0, clearing all flag bits.

**Prototype:**

**void CopyFlags(DWORD fl);**

**Remarks:**

This method is available in release 2.0 and later only.  
Copies all flag bits over from **fl**.

**Prototype:**

**void CopyFlags(const FlagUser & fu);**

**Remarks:**

Copies all flag bits over from **fu**.

**Prototype:**

**void CopyFlags(const FlagUser \* fu);**

**Remarks:**

Copies all flag bits over from **\*fu**.

**Prototype:**

**void CopyFlags(DWORD fl, DWORD mask);**

**Remarks:**

Copies from **fl** only those bits set in **mask**.

**Prototype:**

**void CopyFlags(const FlagUser &fu, DWORD mask);**

**Remarks:**

Copies from **fu** only those bits set in **mask**.

**Prototype:**

**void CopyFlags(const FlagUser \*fu, DWORD mask);**

**Remarks:**

Copies from **\*fu** only those bits set in **mask**.

**Prototype:**

**void OrFlags(const FlagUser & fu);**

**Remarks:**

Sets all flags that are set in **fu**.

**Prototype:**

**void OrFlags(const FlagUser \* fu);**

**Remarks:**

Sets all flags that are set in **\*fu**.

**Prototype:**

**void AndFlags(const FlagUser & fu);**

**Remarks:**

Clears all flags that are clear in **fu**.

**Prototype:**

**void AndFlags(const FlagUser \* fu);**

**Remarks:**

Clears all flags that are clear in **\*fu**.

**Prototype:**

**bool FlagMatch(DWORD fmask, DWORD fl) const;**

**Remarks:**

Checks whether all the bits that are set in fmask are the same in this FlagUser and in fl.

**Prototype:**

**bool FlagMatch(DWORD fmask, const FlagUser & fu) const;**

**Remarks:**

Checks whether all the bits that are set in fmask are the same in this FlagUser and in fu.

**Prototype:**

**bool FlagMatch(DWORD fmask, const FlagUser \* fu) const;**

**Remarks:**

Checks whether all the bits that are set in fmask are the same in this FlagUser and in \*fu.

**Prototype:**

**DWORD ExportFlags() const;**

**Remarks:**

Returns a DWORD equal to FlagUserFlags.

**Prototype:**

**void ImportFlags(DWORD fl);**

**Remarks:**

Sets FlagUserFlags equal to fl. Same as CopyFlags (fl), but it's included for "linguistic completeness".

## Class MNVert

See Also: [Class FlagUser](#), [Class MNMesh](#), [Template Class Tab](#).

```
class MNVert : public FlagUser
```

### Description:

This class is available in release 2.0 and later only.

MNVert is the vertex class used with the MNMesh mesh. MNVerts have not only a Point3 location, but also some flags.

All methods of this class are implemented by the system.

### Data Members:

public:

#### **Point3 p**

The location of this MNVert.

#### **int orig**

This data member is obsolete and should not be used.

### Flags:

For more information on flags, see [Class FlagUser](#).

#### **MN\_SEL**

Indicates that the vertex is selected.

#### **MN\_TARG**

Indicates that the vertex is targeted. (See the MNMesh methods starting with the words TargetBy.)

#### **MN\_DEAD**

Indicates that the vertex is not used and should be ignored. Vertices with the MN\_DEAD flag are deleted in the next MNMesh call to CollapseDeadVerts ().

#### **MN\_BACKFACING**

Indicates that the vertex faces "backwards" in the current viewport. (Changes often.)

#### **MN\_WHATEVER**

Developers should not use this flag and should restrict themselves to

**MN\_USER** and higher bits.

#### **MN\_VERT\_DONE**

Set in algorithms that may accidentally revisit the same vertex twice, to keep them from processing it the second time.

### **MN\_VERT\_HIDDEN**

Different from MN\_HIDDEN - means a vert is on a face's hvtx list, rather than existing independently of faces.

### **MN\_USER(1<<16)**

Flag bits at or above MN\_USER are reserved in all MNMesh components for the plug-in developer, if needed. Since FlagUser-derived classes have 32 flag bits, this allows for up to 16 user-defined flags.

### **Methods:**

#### **Prototype:**

**MNVert();**

#### **Remarks:**

Constructor.

This method initializes the MNVert.

#### **Prototype:**

**MNVert & operator=(MNVert & from);**

#### **Remarks:**

Assignment operator. Copies over all data from "from".

## Class MNEdge

See Also: [Class FlagUser](#), [Class MNMesh](#).

```
class MNEdge : public FlagUser
```

### Description:

This class is available in release 2.0 and later only.

MNEdge is the edge class used with the MNMesh mesh. MNEdges are "winged-edge" structures, which means they keep track of a start vertex, an end vertex, and the (unique) face that uses the start and end vertices in that order. If there is a face that uses the end and start vertices in that order, i.e. that travels this edge in the other direction, it is also recorded.

All methods of this class are implemented by the system.

### Data Members:

public:

**int v1, v2**

The start and end vertices. These values are indices into the parent MNMesh's list of MNVerts.

**int f1**

The (unique) face that references this edge in the forward direction. This value is an index into the parent MNMesh's list of MNFaces.

**int f2**

The face (if any) that references this edge in the backward direction. Faces with f2=-1 are considered "one-sided", and lie on the boundary of a hole in the mesh. This value is an index into the parent MNMesh's list of MNFaces.

**int track**

This data member is obsolete and should not be used.

### Flags:

For more information on flags, see [Class FlagUser](#).

**MN\_SEL**

Indicates that the edge is selected.

**MN\_TARG**

Indicates that the edge is targeted. (See the MNMesh methods starting with the words TargetBy.)

## **MN\_DEAD**

Indicates that the edge is not used and should be ignored. Edges with the MN\_DEAD flag are deleted in the next MNMesh call to CollapseDeadEdges ().

## **MN\_EDGE\_INVIS**

Both faces using this edge consider it invisible.

## **MN\_EDGE\_HALFINVIS**

One face using this edge considers it invisible.

## **MN\_EDGE\_NOCROSS**

This edge should not be crossed in algorithms like MNMesh::SabinDoo that can "mix" faces across edges.

## **MN\_EDGE\_TV\_SEAM**

This edge lies on a mapping coordinate "seam". You must use MNMesh::SetTVSeamFlags on a mesh with mapping coordinates in order to set these flags correctly.

## **MN\_EDGE\_WHATEVER**

Developers should not use this flag and should restrict themselves to MN\_USER and higher bits.

## **MN\_USER(1<<16)**

Flag bits at or above MN\_USER are reserved in all MNMesh components for the plug-in developer, if needed. Since FlagUser-derived classes have 32 flag bits, this allows for up to 16 user-defined flags.

### **Methods:**

#### **Prototype:**

**MNEdge();**

#### **Remarks:**

Constructor. Initializes both faces to -1 and both vertices to 0.

#### **Prototype:**

**MNEdge(int vv1, int vv2, int fc);**

#### **Remarks:**

Constructor. Initializes edge to run from vv1 to vv2 with f1 set to fc.

**Prototype:**

**void Init();**

**Remarks:**

This method is available in release 3.0 and later only.  
Initializes v1, v2 and f1 to 0, f2 to -1 and track to -1.

**Prototype:**

**int OtherFace(int ff);**

**Remarks:**

Assuming that ff is one of the faces using this edge, OtherFace will return the other. If the edge is one-sided, -1 will be returned. If ff is not one of the faces, f2 (which may be -1) will be returned.

**Prototype:**

**int OtherVert(int vv);**

**Remarks:**

Assuming that vv is one of the vertices on this edge, OtherVert will return the other. If vv is not one of the faces, v2 will be returned.

**Prototype:**

**void Invert();**

**Remarks:**

Flips the edge around, so that it now goes from what was v2 to what was v1. f1 and f2 are also switched. This should not be called on one-sided edges.

**Prototype:**

**void ReplaceFace(int of, int nf, int vv1=-1);**

**Remarks:**

This method is available in release 2.5 or later only.

Replaces face **of** with face **nf** in the edge records. NOTE that this method causes an assertion failure if face of is not currently used by the edge. If **of** is on both sides of the edge, which is possible on some valid NONTRI meshes, a nonnegative **vv1** is used to specify which side is replaced. **Vv1** should be the

"starting vertex" for the edge on face of. Assertion failures will also result if **vv1** is nonnegative and is not either of the edge's verts, or if **vv1** indicates that of should be the edge's f1, but it is not, etc.

**Prototype:**

**void ReplaceVert(int ov, int nv);**

**Remarks:**

This method is available in release 2.5 or later only.

Replaces vertex ov in the edge records with vertex nv. NOTE that this method causes an assertion failure if vertex ov is not used by this edge

**Prototype:**

**bool Uncrossable();**

**Remarks:**

If this edge has the **MN\_EDGE\_NOCROSS** flag set, or if it has no second face, this method returns TRUE. Otherwise, it returns FALSE. It's a shorthand equivalent for **(GetFlag(MN\_EDGE\_NOCROSS) || (f2<0))**.

**Prototype:**

**void MNDebugPrint();**

**Remarks:**

This method is available in release 2.5 and later only.

Uses DebugPrint to print out edge information to the Debug Results window in DevStudio. The information consists of the vertices and faces using this edge. It is generally a good idea to put in a DebugPrint immediately before this with the index of the edge, so you know which one is being printed out:

**DebugPrint("Edge %d: ", eid);**

**E(eid)->MNDebugPrint();**

**Prototype:**

**MNEdge & operator=(const MNEdge & from);**

**Remarks:**

Assignment operator. Copies over all data.

**Prototype:**

```
int& operator[](int i);
```

**Remarks:**

This method is available in release 4.0 and later only.

Vertex access operator.

**Default Implementation:**

```
{ return i ? v2 : v1; }
```

**Prototype:**

```
const int& operator[](int i) const;
```

**Remarks:**

This method is available in release 4.0 and later only.

Vertex access operator.

**Default Implementation:**

```
{ return i ? v2 : v1; }
```

## Class MNMeshBorder

See Also: [Class MNMesh](#), [Template Class Tab](#), [Class BitArray](#).

class MNMeshBorder

### Description:

This class is available in release 2.0 and later only.

MNMeshBorder is a small class used to hold boundary information for an MNMesh mesh. The principal data contained is a table of tables of int's, which represent edge lists in the MNMesh. These edge lists form closed loops of one-sided edges: boundaries, or borders, of the mesh. These edges are stored in order such that  $E(\text{loop}(i)[j]) \rightarrow v1$  is the same as  $E(\text{loop}(i)[j+1]) \rightarrow v2$ ,  $E(\text{loop}(i)[j+1]) \rightarrow v1 == E(\text{loop}(i)[j+2]) \rightarrow v2$ , and so on. (This is the right-hand, counterclockwise order when looking down on the hole from outside the mesh.)

Most 3ds max primitives have no borders, but the Patch Grid, when converted to a mesh, is an example of one with a single border.

All methods of this class are implemented by the system.

### Data Members:

private:

**Tab<IntTab \*> bdr**

IntTab is itself a Tab<int>. We use a table of Tab<int> pointers (rather than a table of Tab<int>'s) for clean memory allocation & freeing. However, this is an unwieldy structure, so it's kept private.

**BitArray btarg**

This is simply an array of targeting bits for each of the boundary loops in bdr.

### Methods:

#### Prototype:

**~MNMeshBorder();**

#### Remarks:

Destructor; frees all reserved memory. There is no Constructor for this class, since both data members have their own, adequate constructors.

#### Prototype:

**void Clear();**

**Remarks:**

Frees all reserved memory and reinitializes the data, producing an empty border.

**Prototype:**

**int Num();**

**Remarks:**

Returns the number of border loops for the MNMesh analyzed.

**Prototype:**

**IntTab \*Loop(int i);**

**Remarks:**

Returns a pointer to the i'th border loop.

**Prototype:**

**bool LoopTarg(int i);**

**Remarks:**

Indicates whether border loop **i** is targeted or not.

**Prototype:**

**MNDebugPrint(MNMesh \*m);**

**Remarks:**

This method is available in release 2.5 and later only.

Uses DebugPrint to print out the MNMesh borders to the Debug Results window in DevStudio. This can be useful for tracking down bugs. Be careful not to leave MNDebugPrint calls in your final build; they will slow down your effect to no purpose.

**Parameters:**

**MNMesh \*m**

The MNMesh to which this MNMeshBorder refers is required to give more details about the border.

## MNMesh Note on Edge Characteristics.

See Also: [Class MNMesh](#), [Class Mesh](#).

Regular 3ds max Meshes consist of faces and vertices; they have no real edges as such. Edge characteristics such as visibility and selection are referenced by face. For example, the edgeSel data member of a Class Mesh has  $3 * \text{numFaces}$  members. Each set of three values `edgeSel[i*3]`, `edgeSel[i*3+1]`, and `edgeSel[i*3+2]` represent the selection of the three edges (going from vertex 0 to vertex 1, vertex 1 to vertex 2, and vertex 2 to vertex 0, respectively) of face `i`.

There are similar issues in MNMesh for a number of reasons. Edge selection and visibility data must initially be stored in the MNFace members, since until we call `FillInMesh` there are no MNEdges to store them in. There are a number of methods in the library (and many more the developer could design) which would invalidate the topological data, rendering the MNEdge list invalid. Also, eventually, we're going to want to convert each MNMesh back into a regular Mesh using `OutToTri`, so it's convenient to store edge selection and visibility information in the faces for this purpose.

Thus, although the MNEdges contain `MN_SEL` and `MN_EDGE_INVIS` flags, the more fundamental location for this data is in the MNFaces using that edge. When altering an MNMesh to make edges visible or invisible, or to change their selection status, be sure to make the changes in the MNFaces using the edge, otherwise the information can be easily overwritten, and generally won't get passed to the output Mesh.

With 3.0, there are simple new methods to do this:

```
void MNMesh::SetEdgeVis (int ee, BOOL vis=TRUE);
```

```
void MNMesh::SetEdgeSel (int ee, BOOL sel=TRUE);
```

Otherwise, here's an example of how to set all the information yourself. I want to make edge `ee` visible and selected in the following code:

```
void MakeEdgeVisAndSel(MNMesh & mm, int ee) {  
    assert(mm.GetFlag(MN_MESH_FILLED_IN));  
    MNEdge *me = mm.E(ee);  
    MNFace *mf1 = mm.F(me->f1);  
    MNFace *mf2 =(me->f2>-1) ? mm.F(me->f2) : NULL;  
  
    // Change the edge as desired
```

```
me->ClearFlag(MN_EDGE_INVIS |
MN_EDGE_HALF_INVIS);
me->SetFlag(MN_SEL);

// Make the corresponding changes in face 1
int i;
  i = mf1->EdgeIndex(ee);
  mf1->visedg.Set(i);
  mf1->edgsel.Set(i);

// Make the corresponding changes in face 2
if(mf2) {
  i = mf2->EdgeIndex(ee);
  mf2->visedg.Set(i);
  mf2->edgsel.Set(i);
}
}
```

## MNMesh Notes on Debugging

See Also: [Class MNMesh](#).

MNDebugPrint:

Please familiarize yourself with the new **MNDebugPrint** methods in **MNVert**, **MNEdge**, **MNFace**, **MNMesh**, and **MNMeshBorder**. These functions print out detailed ASCII summaries of the component or mesh, which can be an invaluable aid in tracking bugs. Also, during your debugging phase, frequent calls to **MNMesh::CheckAllData** can be very useful. But, since the **MNDebugPrint** and **CheckAllData** functions are highly time intensive and serve no useful purpose for a released plug-in, you should check to remove them all before release.

As an example, here is the **MNMesh::MNDebugPrint** output from a standard 3ds max box, after applying **MakePolyMesh**. Mapping coordinates, vertex colors, and face triangulation may also be included.

**MNMesh Debug Output: 8 verts, 12 edges, 6 faces**

**Vertex 0:( -17.185, -26.798, 0.000) Edges:(0 3 9 ) Faces:(0 2 5 )**

**Vertex 1:( 17.185, -26.798, 0.000) Edges:(2 3 8 ) Faces:(0 2 3 )**

**Vertex 2:( -17.185, 26.798, 0.000) Edges:(0 1 11 ) Faces:(0 4 5 )**

**Vertex 3:( 17.185, 26.798, 0.000) Edges:(1 2 10 ) Faces:(0 3 4 )**

**Vertex 4:( -17.185, -26.798, 24.984) Edges:(4 7 9 ) Faces:(1 2 5 )**

**Vertex 5:( 17.185, -26.798, 24.984) Edges:(4 5 8 ) Faces:(1 2 3 )**

**Vertex 6:( -17.185, 26.798, 24.984) Edges:(6 7 11 ) Faces:(1 4 5 )**

**Vertex 7:( 17.185, 26.798, 24.984) Edges:(5 6 10 ) Faces:(1 3 4 )**

**Edge 0: verts(0 2), face 0, rev-face 5**

**Edge 1: verts(2 3), face 0, rev-face 4**

**Edge 2: verts(3 1), face 0, rev-face 3**

**Edge 3: verts(1 0), face 0, rev-face 2**

**Edge 4: verts(4 5), face 1, rev-face 2**

**Edge 5: verts(5 7), face 1, rev-face 3**

**Edge 6: verts(7 6), face 1, rev-face 4**

**Edge 7: verts(6 4), face 1, rev-face 5**

**Edge 8: verts(1 5), face 2, rev-face 3**

**Edge 9: verts(4 0), face 2, rev-face 5**

**Edge 10: verts(3 7), face 3, rev-face 4**  
**Edge 11: verts(2 6), face 4, rev-face 5**  
**Face 0: verts(0 2 3 1 ) edges(0v 1v 2v 3v )**  
**Face 1: verts(4 5 7 6 ) edges(4v 5v 6v 7v )**  
**Face 2: verts(0 1 5 4 ) edges(3v 8v 4v 9v )**  
**Face 3: verts(1 3 7 5 ) edges(2v 10v 5v 8v )**  
**Face 4: verts(3 2 6 7 ) edges(1v 11v 6v 10v )**  
**Face 5: verts(2 0 4 6 ) edges(0v 9v 7v 11v )**

Assertion failures:

MNMeshes have a complex, interlinked topology. Without frequent assertions, a mistake in one part of the code can result in a failure much further down the line. Since an assertion failure in itself provides little diagnostic information, each AF in MNMath.dll is accompanied by a DebugPrint message giving more information about the problem. If this information does not immediately lead to a solution, try using assert (**CheckAllData()**) between each of your MNMesh operations.

If you call an **MNMesh** function that produces an inexplicable error after your MNMesh has successfully had all its data checked with **MNMesh::CheckAllData**, please contact SDK support.

To give a simple example, suppose you had an application which created an MNMesh from a Mesh, made it into a polygonal mesh, split a single face, and returned.

```
SplitUp(Mesh & mesh) {  
  MNMesh mm(mesh);  
  mm.MakePolyMesh();  
  mm.SplitTriEdge(0, .5f);  
  mm.OutToTri(mesh);  
}
```

You try it out on a geosphere, and it works fine. But then you try it out on a box, and for some reason you get an assertion failure somewhere in the MNMath.dll source. By running 3ds max under the debugger, you can reproduce the error. If you check the Debug output in Visual Studio, you'll find the message:

**MNMesh::SplitTriEdge error: edge's face 1(0) not a triangle.**

So you check with the documentation and realize that `SplitTriEdge` is an edge-splitting function that only works on triangulated meshes. To do the same thing to a `PolyMesh`, you need to use one of the variants of `SplitEdge`. So you can fix your code by either removing `mm.MakePolyMesh()`, triangulating with a call to `mm.Triangulate()`, or just calling `mm.SplitEdge (0, .5f)` instead of `SplitTriEdge`.

Imagine a more complicated case: in the middle of a series of complex `MNMesh` modifications made by your program, you need to recompute the triangulation of one of the higher-degree faces with a call to `MNMesh::RetriangulateFace`. For some reason, this generates an assertion failure. Checking the Debug output, you find the cryptic message:

**`MNMesh::RetriangulateFace` internal error.**

This could mean one of two things. First of all, the `MNMesh` could have had some of its data scrambled before the call to `RetriangulateFace`. You can check this by adding an assert (`mm.CheckAllData ()`), where `mm` is your `MNMesh`, immediately before the call to `RetriangulateFace`. `CheckAllData` performs a time-intensive series of checks on the mesh structure, checking each mesh component against the components it's supposed to be linked to, checking that nothing references a dead component, and so on. If any errors are found, the check will fail, and `CheckAllData` will print diagnostic information to the Debug output.

If `CheckAllData` gives your mesh a clean bill of health, but you're still getting an "internal error", or just an "error" with no further info, you may have found a bug. Please contact SDK support.

## List of Edge Data Index Options

See Also: [Class MNMesh](#), [Class PerData](#).

The following are the edge data channel index values for use with the edge data methods of class **MNMesh** and class **EPoly**.

**EDATA\_KNOT**: The edge knot data. This is index 0.

**EDATA\_CREASE**: The crease data. This is index 1.

**1** through **MAX\_EDGEDATA-1**: Developer defined data. Note: **#define MAX\_EDGEDATA 10**

## Class MNTempData

See Also: [Class BaseInterfaceServer](#), [Class MNMesh](#), [Class MNFaceClusters](#), [Class MNEdgeClusters](#), [Class MNFaceElement](#), [Class MNChamferData](#)

```
class MNTempData : public BaseInterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This is a class for caching face and edge clusters, vertex normals, and other derived data about an **MNMesh**. There is a **SetMesh()** method to set the current mesh that the TempData is based on, then there's a series of methods to update the cache and return some sort of derived data. All of these methods follow the form:

**DerivedData \*MNTempData::DData(parameters);**

DerivedData is the container for the derived data requested (often a simple table, though there are some specialized classes returned from some methods). If the data has already been computed, the parameters are ignored and the cached data is returned. Otherwise, the data is computed from the parameters and the current mesh.

There are no procedures in place to detect changes in parameters or the mesh since the last time a method was called, so it's the calling routine's responsibility to free invalid structures. If you know that only certain pipeline channels, such as **GEOM\_CHANNEL**, have changed, you can use the **Invalidate(DWORD partsChanged)** method. (**GEOM\_CHANNEL** would free the distances-to-selected-vertices, for example, but not the Edge Clusters.)

In particular, there is no way for the **MNTempData** to know when its mesh pointer is no longer valid, so it's vital that the calling routine clear the mesh (with **SetMesh(NULL)**) or stop using the MNTempData when this happens.

All data members are private. They basically consist of a series of pointers which are initialized to NULL and then filled with allocated derived data as requested. There is also a NULL-initialized, private mesh pointer which is set with **SetMesh()**. Editable Poly uses this class to hold all the varieties of

temporary, cached data it creates -- examples are vertex normals and face clusters.

To use MNTempData, just set it to your mesh and start asking for stuff:

```
MyAlgorithm (MNMesh *m) {  
    MNTempData mtd(m);  
    // Get Edge Clusters.  
    MNEdgeClusters *eclust = mtd.EdgeClusters ();  
}
```

**Methods:**

public:

**Prototype:**

```
MNTempData ();
```

**Remarks:**

Constructor. Sets all data members to NULL.

**Prototype:**

```
MNTempData (MNMesh *m);
```

**Remarks:**

Constructor. Sets the internal mesh pointer to the mesh passed. (Sets all other data members to NULL.)

**Parameters:**

```
MNMesh *m
```

The internal mesh pointer to set.

**Prototype:**

```
~MNTempData ();
```

**Remarks:**

Destructor.

**Prototype:**

```
void SetMesh (MNMesh *m);
```

**Remarks:**

Sets the internal mesh pointer to **m**.

**Parameters:**

```
MNMesh *m
```

The internal mesh pointer to set.

**Default Implementation:**

```
{ mesh = m; }
```

**Prototype:**

```
MNFaceClusters *FaceClusters (DWORD  
clusterFlags=MN_SEL);
```

**Remarks:**

This method returns a face cluster list, which groups selected faces into "clusters" for transformation. See [Class MNFaceClusters](#) for more information. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh.

**Parameters:**

```
DWORD clusterFlags=MN_SEL
```

The face flags to cluster the faces by. For instance, with the default value, faces are clustered by their selection.

**Prototype:**

```
MNEdgeClusters *EdgeClusters (DWORD  
clusterFlags=MN_SEL);
```

**Remarks:**

Returns an edge cluster list, which groups selected edges into "clusters" for applying transforms. See [Class MNEdgeClusters](#) for more information. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh.

**Parameters:**

### **DWORD clusterFlags=MN\_SEL**

The edge flags to cluster the edges by. For instance, with the default value, edges are clustered by their selection.

#### **Prototype:**

```
Tab<int> *VertexClusters (int sl, DWORD  
clusterFlags=MN_SEL);
```

#### **Remarks:**

This method returns an index of which cluster, if any, each vertex is in. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameter.

#### **Parameters:**

##### **int sl**

The selection level. This should be either **MNM\_SL\_EDGE** or **MNM\_SL\_FACE**, to indicate whether the vertex cluster information should be based on edge or face clusters. Note that this parameter is ignored if there's already a vertex cluster cache.

##### **DWORD clusterFlags=MN\_SEL**

The edge or face flags to cluster the edges or faces by. For instance, with the default value, edges or faces are clustered by their selection. Note that this parameter is ignored if there's already a vertex cluster cache.

#### **Return Value:**

A table of DWORD's is returned, one for each vertex. If **(VertexClusters(sl))[i]** is **UNDEFINED**, vertex *i* is not in any cluster. Otherwise, the value for vertex *i* is the cluster index.

#### **Prototype:**

```
Tab<Point3> *ClusterNormals (int sl, DWORD  
clusterFlags=MN_SEL);
```

#### **Remarks:**

This method returns average normals for each cluster. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameter. Note that cluster centers and normals are computed and cached at the same time, when you call either method.

**Parameters:****int sl**

The selection level. This should be either **MNM\_SL\_EDGE** or **MNM\_SL\_FACE**, to indicate whether the vertex cluster information should be based on edge or face clusters. Note that this parameter is ignored if there's already a cluster normal cache.

**DWORD clusterFlags=MN\_SEL**

The edge or face flags to cluster the edges or faces by. For instance, with the default value, edges or faces are clustered by their selection. Note that this parameter is ignored if there's already a cluster normal cache.

**Return Value:**

A table of Point3's is returned, one for each cluster. The values are already normalized to length 1.

**Prototype:**

```
Tab<Point3> *ClusterCenters (int sl, DWORD  
clusterFlags=MN_SEL);
```

**Remarks:**

This method returns mean centers for each cluster. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameter. Note that cluster centers and normals are computed and cached at the same time, when you call either method.

**Parameters:****int sl**

Selection level. This should be either **MNM\_SL\_EDGE** or **MNM\_SL\_FACE**, to indicate whether the clusters we're talking about are the edge or face clusters. Note that this parameter is ignored if there's already a cluster center cache.

**DWORD clusterFlags=MN\_SEL**

The edge or face flags to cluster the edges or faces by. For instance, with the default value, edges or faces are clustered by their selection. Note that this parameter is ignored if there's already a cluster center cache.

**Return Value:**

A table of Point3's is returned, one for each cluster.

**Prototype:**

**Matrix3 ClusterTM (int clust);**

**Remarks:**

This method uses the current cluster center and normal caches to return the "objectspace to clusterspace" transform. This is the transform of the "local" axis in moving edge or face clusters in Editable Poly. If the cluster centers and normals have not been cached, the identity matrix is returned; thus the control over whether this is an edge or face cluster is handled by the last call to ClusterCenters or ClusterNormals.

**Parameters:**

**int clust**

The cluster you want the transform for.

**Prototype:**

**Tab<Point3> \*VertexNormals ();**

**Remarks:**

This method returns a table of local average normals for vertices. These normals are computed using the **MNMesh::GetVertexNormal()** method, wherein each face's contribution to the vertex normal is weighted by the face angle at the vertex.

**Prototype:**

**Tab<float> \*VSWeight (BOOL useEdgeDist, int edgeIts, BOOL ignoreBack, float falloff, float pinch, float bubble, DWORD selFlags=MN\_SEL);**

**Remarks:**

This method returns Vertex Selection weights (for soft selection). If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters. Weights are based on the standard soft selection falloff from the currently selected vertices.

Note: If **useEdgeDist** is FALSE, this is an n-log-n algorithm: it compares

every vertex not in the cluster with every vertex in it. If **useEdgeDist** is TRUE, the time it takes is proportional to the number of **verts** in the cluster times **edgeIts**.

**Parameters:**

**BOOL useEdgeDist**

If set to TRUE, the distance between vertices is computed along edges. If set to FALSE, it's computed directly through space.

**int edgeIts**

This indicates the maximum number of edges the algorithm may travel along in finding the distance between vertices. (Maximum path length.)

**BOOL ignoreBack**

If set to TRUE, vertices with a normal (as computed in VertexNormals) that points more than 90 degrees away from the average normal of the selection are not given any partial selections. They're either 1 if selected or 0 otherwise.

**float falloff**

The limit distance of the effect. If distance > falloff, the function will always return 0.

**float pinch**

Use this to affect the tangency of the curve near distance=0. Positive values produce a pointed tip, with a negative slope at 0, while negative values produce a dimple, with positive slope.

**float bubble**

Use this to change the curvature of the function. A value of 1.0 produces a half-dome. As you reduce this value, the sides of the dome slope more steeply. Negative values lower the base of the curve below 0.

**DWORD selFlags=MN\_SEL**

Indicates what flag defines the hard selection we're basing this soft selection on.

**Return Value:**

A table of float values, one per vertex, that are 1.0 if the vertex is in the current selection, 0.0 if it's more than falloff distance (or more than edgeIts edges, if (useEdgeDist)), and AffectRegionFunction((\*SelectionDist(useEdgeDist, edgeIts)), falloff, pinch, bubble) otherwise.

**Prototype:**

```
Tab<float> *SelectionDist (BOOL useEdgeDist, int edgeIts,  
    DWORD selFlags=MN_SEL);
```

**Remarks:**

This method computes the current distance of each vertex from the current selection. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters. The term "Selected verts" below refers to the vertices that are selected in the mesh's current selection level. See the **Mesh** method **GetTempSel()** for details.

NOTE: If **useEdgeDist** is FALSE, this is an n-log-n algorithm: it compares every nonselected vertex with every selected one. If **useEdgeDist** is TRUE, the time it takes is proportional to the number of selected vertices times **edgeIts**.

**Parameters:****BOOL useEdgeDist**

If set to TRUE, the distance between vertices is computed along edges. If set to FALSE, it's computed directly through space.

**int edgeIts**

This indicates the maximum number of edges the algorithm may travel along in finding the distance between vertices. (Maximum path length.)

**DWORD selFlags=MN\_SEL**

Indicates what flag defines the hard selection we're basing this soft selection on.

**Return Value:**

A table consisting of one float value per vertex. If this value is 0, the vertex is either selected or on top of a selected vertex. Otherwise it represents the distance to the closest selected vertex. If **useEdgeDist** is TRUE, values of -1.0 are returned for vertices with no **edgeIts**-length path to a selected vertex.

**Prototype:**

```
Tab<float> *ClusterDist (int sl, DWORD clusterFlags, int clustId,  
    BOOL useEdgeDist, int edgeIts);
```

**Remarks:**

This method computes the current distance of each vertex from the specified cluster. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters.

NOTE: If **useEdgeDist** is FALSE, this is an n-log-n algorithm: it compares every vertex not in the cluster with every vertex in it. If **useEdgeDist** is TRUE, the time it takes is proportional to the number of verts in the cluster times **edgeIts**.

**Parameters:**

**int sl**

Indicates whether we should use edges (**MNM\_SL\_EDGE**) or faces (**MNM\_SL\_FACE**) to construct the clusters, if needed.

**DWORD clusterFlags**

The edge or face flags to cluster the edges or faces by. For instance, if **clusterFlags==MN\_SEL**, edges or faces are clustered by selection.

**int clustId**

The ID of the cluster we're measuring distance from.

**BOOL useEdgeDist**

If set to TRUE, the distance between vertices is computed along edges. If set to FALSE, it's computed directly through space.

**int edgeIts**

This indicates the maximum number of edges the algorithm may travel along in finding the distance between vertices. (Maximum path length.)

**Return Value:**

A table consisting of one float value per vertex. If this value is 0, the vertex is either selected or on top of a vertex in the cluster. Otherwise it represents the distance to the closest selected vertex. If **useEdgeDist** is TRUE, values of -1.0 are returned for vertices with no **edgeIts**-length path to a vertex in the cluster.

**Prototype:**

```
Tab<Point3> *OutlineDir (int extrusionType, DWORD  
clusterFlags=MN_SEL);
```

**Remarks:**

This method produces the "Outline" direction of all vertices, based on the current face selection. "Outlining" is the direction vertices move to move edges of the current face selection outward at a constant rate. They are not set to length 1, but rather to whatever "rate" best makes the outline edges move most consistently, without changing their angles.

**Parameters:**

**int extrusionType**

This is one of **MESH\_EXTRUDE\_CLUSTER** or **MESH\_EXTRUDE\_LOCAL**, to indicate whether vertices should move according to cluster or local face normals.

**DWORD clusterFlags=MN\_SEL**

The face flags to cluster the faces by. For instance, if left at the default value, faces are clustered by selection. Note that this parameter is ignored if the **extrusionType** is **LOCAL**.

**Prototype:**

**MNChamferData \*ChamferData();**

**Remarks:**

This method returns the cache of a **ChamferData** for use in the **MNMesh** Chamfer methods. Unlike other **MeshTempData** methods, this method makes no calculations based on the current mesh, but merely supplies a memory cache. (Computing this information is left to the **MNMesh** methods **GetExtrudeDirection**, **ChamferVertices**, and **ChamferEdges**.)

**Prototype:**

**void Invalidate (DWORD part);**

**Remarks:**

This method invalidates all data based on the specified part of the mesh. In the following chart, the columns represent the channels **GEOM\_CHANNEL** (G), **TOPO\_CHANNEL** (T), **SELECT\_CHANNEL** (S), and **SUBSEL\_TYPE\_CHANNEL** (U). X's indicate dependency of the specified data cache on the given channel.

Method to get cache	G	T	S	U
---------------------	---	---	---	---

FaceClusters		X	X	
EdgeClusters		X	X	
VertexClusters		X	X	X
ClusterCenters	X	X	X	X
ClusterNormals	X	X	X	X
VertexNormals	X	X		
SelectionDist	X	X	X	X
ClusterDist	X	X	X	X
VSWeight	X	X	X	X

NOTE: **ChamferData** and the Outline direction info are handled separately in **freeChamferData** and **freeBevellInfo**.

**Parameters:**

**DWORD part**

One or more of the following channels: **GEOM\_CHANNEL**, **TOPO\_CHANNEL**, **SELECT\_CHANNEL**, **SUBSEL\_TYPE\_CHANNEL**.

**Prototype:**

**void InvalidateDistances ();**

**Remarks:**

This method Uncaches (frees) the distance dependent data returned by **VSWeight**, **SelectionDist**, and **ClusterDist**.

**Prototype:**

**void InvalidateSoftSelection ();**

**Remarks:**

This method frees the **VSWeight** data (but not the underlying distance-from-selection info). This is useful, e.g., if the mesh has not changed, but you wish to change the falloff, pinch, or bubble parameters to get new vertex selection weights.

**Prototype:**

**void freeClusterDist ();**

**Remarks:**

This method is mainly for internal use, this frees just the cluster distance data.

**Prototype:**

**void freeBevelInfo ();**

**Remarks:**

This method frees only the outlining data.

**Prototype:**

**void freeChamferData();**

**Remarks:**

This method frees only the chamfer data structure.

**Prototype:**

**void freeAll ();**

**Remarks:**

This method frees all cached data of any kind.

## **Global Functions**

The following global methods are also available for use, though the related methods in the **MNTempData** class are more highly recommended.

**Prototype:**

**void SelectionDistance (MNMesh & mesh, float \*selDist, DWORD selFlags);**

**Remarks:**

This function computes the current distance of each vertex from the current selection. NOTE: This is an n-log-n algorithm: it compares every non-selected vertex with every selected one.

**Parameters:**

## **MNMesh & mesh**

The mesh we're computing distances in.

### **float \*selDist**

A pointer to an array of floats of size **mesh.VNum()**. This array is filled in with one float value per vertex. If this value is 0, the vertex is either selected or "on top of" a selected vertex. Otherwise it represents the distance to the closest selected vertex.

### **DWORD selFlags**

Indicates what flag defines the hard selection we're basing this soft selection on. (**MN\_SEL** is generally best.)

## **Prototype:**

```
void SelectionDistance (MNMesh & mesh, float *selDist, int iters,  
DWORD selFlags);
```

## **Remarks:**

This function computes the current distance of each vertex from the current selection, along paths of edges. NOTE: This is an n-log-n algorithm: it compares every non-selected vertex with every selected one.

## **Parameters:**

### **MNMesh & mesh**

The mesh we're computing distances in.

### **float \*selDist**

A pointer to an array of floats of size **mesh.VNum()**. This array is filled in with one float value per vertex. If this value is 0, the vertex is either selected or "on top of" a selected vertex. Otherwise it represents the distance to the closest selected vertex. Values of -1 are used to indicate vertices that are more than **iters** edges away from any selected vertex.

### **int iters**

This indicates the maximum number of edges the algorithm may travel along in finding the distance between vertices. (Maximum path length.)

### **DWORD selFlags**

Indicates what flag defines the hard selection we're basing this soft selection on. (**MN\_SEL** is generally best.)

**Prototype:**

```
void ClustDistances (MNMesh & mesh, int numClusts, int *vclust,  
Tab<float> **clustDist);
```

**Remarks:**

Computes the current distance of each vertex from the specified cluster. If cached, the cache is returned. Otherwise a cache is allocated and computed from the current mesh and the parameters.

NOTE: This is an n-log-n algorithm for each cluster: it compares every vertex not in the cluster with every vertex in it.

**Parameters:****MNMesh & mesh**

The MNMesh these cluster distances are based on.

**int numClusts**

The number of clusters available.

**int \*vclust**

A pointer into the vertex cluster table.

**Tab<float> \*\*clustDist**

An array of pointers to tables which will be filled with one float value per vertex. The table pointed to by **clustDist[i]** contains the cluster distances for cluster i. Note that **clustDist** must be allocated, for instance by **clustDist = new (Tab<float>\*)[numClusts]**, and must have its members allocated, for instance by **clustDist[i] = new Tab<float>**, by the calling routine. If a value in a table is 0, the vertex is either selected or on top of a vertex in the cluster. Otherwise it represents the distance to the closest selected vertex. If **useEdgeDist** is TRUE, values of -1.0 are returned for vertices with no **edgeIts**-length path to a vertex in the cluster.

**Prototype:**

```
void ClustDistances (MNMesh & mesh, int numClusts, int *vclust,  
Tab<float> **clustDist, int iters);
```

**Remarks:**

Computes the current distance of each vertex from the specified cluster. If cached, the cache is returned. Otherwise a cache is allocated and computed

from the current mesh and the parameters.

NOTE: This algorithm takes time proportional to the number of **verts** in each cluster times **iters** times the number of clusters.

**Parameters:**

**MNMesh & mesh**

The MNMesh these cluster distances are based on.

**int numClusts**

The number of clusters available.

**int \*vclust**

A pointer into the vertex cluster table.

**Tab<float> \*\*clustDist**

An array of pointers to tables which will be filled with one float value per vertex. The table pointed to by **clustDist[i]** contains the cluster distances for cluster i. Note that **clustDist** must be allocated, for instance by **clustDist = new (Tab<float> \*)[numClusts]**, and must have its members allocated, for instance by **clustDist[i] = new Tab<float>**, by the calling routine. If a value in a table is 0, the vertex is either selected or on top of a vertex in the cluster. Otherwise it represents the distance to the closest selected vertex. Values of -1.0 are returned for vertices with no **iters**-length path to a vertex in the cluster.

**int iters**

The maximum edge path length to compute distance along (in number of edges).

## Class MNChamferData

See Also: [Class MNTempData](#), [Class MNMesh](#)

class MNChamferData

### Description:

This class is available in release 4.0 and later only.

This class contains all the data needed to move points (and map vertices) as the user drags a chamfer or extrude. It's created by the topological change that happens at the start of the chamfer or extrude. The strategy is this: The chamfer/extrude operation is divided into two parts, the topological change and a later geometric change. (This works well for Editable Poly, where the topology change is completed first, then apply a series of geometry changes as the user spins a spinner or drags a mouse. Each geometry change is undone before the next is applied, but the topology change only happens once.)

This class is first initialized to a mesh. Then its data is filled in by the topological change. This data is used to find "directions" for all the geometric and mapping vert changes over the course of the geometric modification.

For convenient caching, it is recommended that you use this class through the **MNTempData** class.

### Data Members:

private:

**Tab<UVVert> hmdir[NUM\_HIDDENMAPS];**

The direction vectors for mapping vertices in any active "hidden" mapping channels.

public:

**Tab<Point3> vdir;**

The related direction vectors for mapping vertices in all active mapping channels.

**Tab<Point3> vmax;**

The maximum amount each vector may be applied (before vertices start crossing over each other).

**Tab<UVVert> \*mdir;**

The related direction vectors for mapping vertices in all active mapping channels.

**Methods:**

public:

**Prototype:**

```
MNChamferData ();
```

**Remarks:**

Constructor.

**Default Implementation:**

```
{ mdir=NULL; }
```

**Prototype:**

```
MNChamferData (const MNMesh & m);
```

**Remarks:**

Constructor.

This constructor initializes to the mesh passed (allocates mapping channels, etc.)

**Parameters:**

```
const MNMesh & m
```

The specified mesh.

**Default Implementation:**

```
{ mdir=NULL; InitToMesh(m); }
```

**Prototype:**

```
~MNChamferData ();
```

**Remarks:**

Destructor.

**Default Implementation:**

```
{ if (mdir) delete [] mdir; }
```

**Prototype:**

```
void InitToMesh (const MNMesh & m);
```

**Remarks:**

This method sets up the **MNChamferData** based on a given mesh, allocating the vertex and mapping vertex tables as appropriate.

**Parameters:**

**const MNMesh & m**

The Mesh to initialize from.

**Prototype:**

**void setNumVerts (int nv, bool keep=TRUE, int resizer=0);**

**Remarks:**

This method simply allocates the **vdir** and **vmax** tables, and initializes the new members of **vmax** to 0. (Note: this method can be applied to an existing **MNChamferData** to reflect an increase in vertices in the **MNMesh** as topological changes occur.)

**Parameters:**

**int nv**

The number of vertices.

**bool keep=TRUE**

TRUE to keep old data if resized; FALSE to discard old data.

**int resizer=0**

The number of extra elements the **vdir** and **vmax** tables are resized beyond their current size. (Extra allocation space to prevent excessive reallocation.)

**Prototype:**

**void ClearLimits ();**

**Remarks:**

Clears all the **vmax** limits to -1 (no limit).

**Prototype:**

**void GetDelta (float amount, Tab<Point3> & delta);**

**Remarks:**

Uses vectors and limits to obtain the offsets corresponding to a certain extrude

or chamfer amount.

**Parameters:**

**float amount**

The amount of the extrude or chamfer.

**Tab<Point3> & delta**

A table (with size set equal to the number of vertices, aka **vdir.Count()**) containing the geometric offset for each vertex in the mesh.

**Prototype:**

**bool GetMapDelta (MNMesh & mm, int mapChannel, float amount, Tab<UVVert> & delta);**

**Remarks:**

Uses map vectors and limits to obtain the mapping offsets corresponding to a certain extrude or chamfer amount.

**Parameters:**

**MNMesh & mm**

The mesh this **MNChamferData** is based on.

**int mapChannel**

The index of the map channel (from **-NUM\_HIDDENMAPS** to **mm.MNum()**).

**float amount**

The amount of the extrude or chamfer.

**Tab<UVVert> & delta**

The offsets for each mapping vertex in this map in the mesh.

**Prototype:**

**Tab<UVVert> & MDir (int mp);**

**Remarks:**

Data accessor. This method returns the appropriate map info. If **mp**  $\geq 0$ , it returns the member of the **mdir** array. If **mp**  $< 0$ , it returns the member of the **hmdir** array (in keeping with "hidden map channel" indexing conventions).

**Default Implementation:**

```
{ return (mp<0) ? hmdir[-1-mp] : mdir[mp]; }
```

**The following are not part of the class but are useful for debugging**

**Prototype:**

```
DllExport void MNChamferDataDebugPrint (MNChamferData &  
mcd, int mapNum);
```

**Description**

This function uses calls to DebugPrint() to output all the data in the specified MNChamferData to the DebugPrint buffer during debug runs. It is available for programmers' use, providing easy access to MNChamferData during development. It ought to be removed for release builds.

**Parameters:**

**MNChamferData & mcd**

The MNChamferData we want to investigate.

**int mapNum**

The number of map channels in the MNMesh associated with this MNChamferData . (For historical reasons, this information is not kept in the MNChamferData class.) Generally this is retrieved with a call to MNMesh::MNum().

## Class FrameRange

See Also: [Class ImageFilterInfo](#).

class FrameRange

### Description:

This class describes a range of frames and provides methods to access the first, last, current and number of frames in the range. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**virtual int First()**

#### Remarks:

Returns the first frame number of this range.

#### Prototype:

**virtual int Last()**

#### Remarks:

Returns the last frame number of this range.

#### Prototype:

**virtual int Count()**

#### Remarks:

Returns the number of frames in this range.

#### Prototype:

**virtual int Current()**

#### Remarks:

Returns the current frame of this range.

#### Prototype:

**virtual int Elapsed()**

**Remarks:**

Returns the elapsed time of this range.

**Prototype:**

**virtual void SetFirst(int u)**

**Remarks:**

Sets the first frame number for this range to the specified value.

**Parameters:**

**int u**

Specifies the new first frame number.

**Prototype:**

**virtual void SetLast(int u)**

**Remarks:**

Sets the last frame number for this range to the specified value.

**Parameters:**

**int u**

Specifies the new last frame number.

**Prototype:**

**virtual void SetCurrent(int u)**

**Remarks:**

Sets the current frame number for this range to the specified value.

**Parameters:**

**int u**

Specifies the new current frame number.

## Class TVNodeNotify

See Also: [Class ITrackViewNode](#), [Class UndoNotify](#).

```
class TVNodeNotify
```

### Description:

This class is available in release 2.0 and later only.

This class is the callback object for

**ITrackViewNode::RegisterTVNodeNotify()**. Developers should derive their class from this class and implement the **NotifyRefChanged()** method. This allows the Track View Node to intercept reference notifications when they use a Track View Node.

For an example of this class in use by **ImageFilter** plug-ins see class **UndoNotify** in `\MAXSDK\INCLUDE\FILTERS.H`. It is sub-classed from this class and provides an implementation of **NotifyRefChanged()**.

All methods of this class are implemented by the plug-in.

### Methods:

#### Prototype:

```
virtual RefResult NotifyRefChanged(Interval changeInt,  
RefTargetHandle hTarget, PartID& partID, RefMessage  
message)=0;
```

### Remarks:

A plug-in which makes references must implement this method to receive and respond to messages broadcast by its dependents.

### Parameters:

#### Interval changeInt

This is the interval of time over which the message is active. Currently, all plug-ins will receive **FOREVER** for this interval.

#### RefTargetHandle hTarget

This is the handle of the reference target the message was sent by. The reference maker uses this handle to know specifically which reference target sent the message.

#### PartID& partID

This contains information specific to the message passed in. Some messages don't use the **partID** at all. See the section [List of Reference Messages](#) for more information about the meaning of the **partID** for some common messages.

### **RefMessage message**

The **message** parameters passed into this method is the specific message which needs to be handled. See [List of Reference Messages](#).

### **Return Value:**

The return value from this method is of type **RefResult**. This is usually **REF\_SUCCEED** indicating the message was processed. Sometimes, the return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

## DLL Setup (SDK)

MAXScript plug-in extensions are packaged as Win32 DLL's in almost exactly the same way as standard MAX plug-ins. They are named with a specific suffix to fit in with the MAX plug-in naming scheme (.DLX) and should be installed in one of the MAX plug-in directories. MAXScript looks for and loads all the **\*.DLX** files it finds in the MAX plug-in directories.

You should be familiar with the section on MAX plug-in construction in the MAX SDK documentation because the following notes primarily describe what is different about setting up MAXScript plug-ins.

The plug-in's external interface is defined with a **.DEF** file, as are MAX plug-ins, but in this case there are only three entry points:

```
LIBRARY <plugin_name>
EXPORTS
LibDescription @1
LibInit @2
LibVersion @3
SECTIONS
.data READ WRITE
```

The **LibDescription** and **LibVersion** entry points are the same as for MAX plug-ins. The **LibInit** entry is called by MAX after MAXScript has loaded itself and all its .DLX plug-ins so that each can perform any runtime initialization. MAXScript is fully functional at the time these init functions are called.

The following code shows a typical DLX entry point implementation (this is from **sample.cpp** in the SDK). All of the required headers are included by the master header file, **MAXscript.h**, including **Max.h** for MAX SDK-related code.

```

#include "MAXScript.h"

HMODULE hInstance;

BOOL APIENTRY
DllMain(HMODULE hModule, DWORD ul_reason_for_call,
LPVOID lpReserved)
{
static BOOL controlsInit = FALSE;
switch (ul_reason_for_call)
{
case DLL_PROCESS_ATTACH:
// Hang on to this DLL's instance handle.
hInstance = hModule;
if (!controlsInit)
{
controlsInit = TRUE;
// Initialize Win95 controls
InitCommonControls();
}
break;
}
return(TRUE);
}

__declspec( dllexport ) void
LibInit()
{
// do any setup here. MAXScript is fully up at this point.
}

__declspec( dllexport ) const TCHAR *
LibDescription()

```

```
{  
return _T("Sample MAXscript DLX");  
}
```

```
__declspec( dllexport ) ULONG  
LibVersion()  
{  
return VERSION_3DSMAX;  
}
```

If any Win32 common controls or MAX custom controls are to be used in the DLL, you should initialize them during process attach, in the same way you would in a MAX DLL. For the **LibVersion** entry point, you should return the current 3DSMAX version as shown.

## **Libraries (SDK)**

The MAXScript SDK API is defined in the various headers and the import library, **Maxscript.lib**, supplied with the SDK. You should add this library to your project along with any MAX SDK import libraries needed to support MAX SDK use. The header files are described in the [MAXScript Header Files](#) topic.

### **Build Configurations (SDK)**

For the moment, there is no Debug version of the SDK. You should configure VC++ builds as you would **Hybrid** MAX SDK plug-in builds (that is, exactly as **Debug** with code generation set for Multi-threaded DLL rather than Debug Multi-threaded DLL).

## MAXScript Value Constructors (SDK)

In the course of implementing new extensions to MAXScript, you will encounter the need to create MAXScript values, perhaps to pass back results or to work with other parts of the MAXScript API. This topic describes the MAXScript value classes, their constructors and how to cooperate with the automatic garbage collector.

All computable values in MAXScript are instances of classes derived from Value. MAXScript is polymorphic across all Value subclasses. These values are allocated in a separate, garbage collected heap, and carry their own runtime typing info.

All non-static instances of the Value class family should be heap-allocated with the 'new' operator or one of the following intern() functions so that garbage collection will work.

Certain classes with immutable instances provide interning functions in place of C++ constructors that may implement caches or other interning mechanisms. In particular, Name instances are always be created with Name::intern() to ensure pointer equality on names.

```
static Value* Float::intern(float init_val);
static Value* Integer::intern(int init_val);
static Value* Name::intern(char* str);
static Value* Name::find_intern(char* str);
static Value* MStime::intern(TimeValue t);
```

```
AngAxisValue (const AngAxis& iaa);
AngAxisValue (const Matrix3& m);
AngAxisValue (const Quat& q);
AngAxisValue (float angle, Point3 axis);
AngAxisValue (float* angles);
```

AngAxisValue (Value\* angle, Value\* axis);  
AngAxisValue (Value\*);  
Array (int init\_size);  
ColorValue (AColor col);  
ColorValue (BMM\_Color\_64& col);  
ColorValue (Color col);  
ColorValue (COLORREF col);  
ColorValue (float r, float g, float b, float a = 1.0f);  
ColorValue (Point3 col);  
ColorValue (Point3Value\* col);  
EulerAnglesValue (const AngAxis&);  
EulerAnglesValue (const Matrix3&);  
EulerAnglesValue (const Quat&);  
EulerAnglesValue (float ax, float ay, float az);  
Matrix3Value (const AngAxis& aa);  
Matrix3Value (const Matrix3& im);  
Matrix3Value (const Point3& row0, const Point3& row1,  
const Point3& row2, Point3& row3);  
Matrix3Value (const Quat& q);  
Matrix3Value (float\* angles);  
Matrix3Value (int i);  
MSInterval (Interval i);  
MSInterval (TimeValue s, TimeValue e);  
Point2Value (float x, float y);  
Point2Value (POINT ipoint);  
Point2Value (Point2 ipoint);  
Point2Value (Value\* x, Value\* y);  
Point3Value (float x, float y, float z);  
Point3Value (Point3 init\_point);  
Point3Value (Value\* x, Value\* y, Value\* z);  
QuatValue (AngAxis& aa);  
QuatValue (const Quat& init\_quat);  
QuatValue (float w, float x, float y, float z);

```
QuatValue (float* angles);  
QuatValue (Matrix3& m);  
QuatValue (Value* val);  
QuatValue (Value* w, Value* x, Value* y, Value* z);  
RayValue (Point3 init_origin, Point3 init_dir);  
RayValue (Ray init_ray);
```

## Distinguished Values (SDK)

There are several distinguished values in MAXScript that are implemented as static instances of their classes, as follows:

```
Undefined undefined;  
Ok ok;  
Unsupplied unsupplied;  
Boolean true_value;  
Boolean false_value;
```

Because these are static instances, you need to remember to work with their addresses:

```
return &ok; // return the ok value  
  
if (val == &>true_value) // value true?  
...
```

## Coercion to C++ Types (SDK)

The Value class defines the following virtual functions for converting selected Value instances to appropriate C++ types. Value subclasses implement the converters that make sense for them. Attempting an unsupported conversion will generate a descriptive MAXScript runtime error.

In most cases, MAXScript Value classes act as wrappers for C++ types and classes. Converters that yield pointer types all return pointers to the value being wrapped, so take care when modifying the data and be aware that the collector may invalidate the pointer underneath you (the topic [Protecting Newly Created Values from the Collector](#) shows you ways to prevent this). At the moment, the collector is synchronous and will only run if needed during an allocate.

```
class Value
virtual float to_float();
virtual char* to_string();
virtual int to_int();
virtual BOOL to_bool();
virtual Point3 to_point3();
virtual Point2 to_point2();
virtual AColor to_acolor();
virtual INode* to_node();
virtual Ray to_ray();
virtual Interval to_interval();
virtual Quat to_quat();
virtual AngAxis to_angaxis();
virtual Matrix3& to_matrix3();
virtual float* to_eulerangles();
virtual Mtl* to_mtl();
```

```
virtual Texmap* to_texmap();  
virtual Modifier* to_modifier();  
virtual TimeValue to_timevalue();  
virtual Control* to_controller();
```

### **Protecting Newly Created Values from the Collector (SDK)**

Value family instances created in running C++ code are immediately visible to the collector. The collector cannot see references in plain C++ local or global variables, so to prevent them from being collected while just in C++ variables, you can use the following macros to ensure the collector leaves them alone. You also need to do this if you take a value out of some structure such that nothing else is referencing it and still want to keep hold of it in a C++ variable.

An important convention is that values you get in arguments passed to you are guaranteed to be protected by someone up the call chain. You must ensure this convention holds yourself for any function you call.

**See Also:**

[Collector-Safe Value Local Macros](#)

## Collector-Safe Value Local Macros (SDK)

The general approach is to declare special C++ local variables that are known by the collector to contain protected values. You declare them with one of the following macros:

```
one_value_local(name);
two_value_locals(name1, name2);
three_value_locals(name1, name2, name3);
... etc., up to eight locals.
```

They are declared as type `Value*` to be able to hold any MAXScript value. There are also explicitly typed variants:

```
one_typed_value_local(type name);
two_typed_value_local(type1 name1, type2 name2);
...
```

For example,

```
two_typed_value_locals(Array* result, String* item);
```

These locals are implemented as members in a local struct named 'vl', so you work with them as members of that struct:

```
two_typed_value_locals(Array* result, String* item);
...
vl.result = new Array (5);
...
vl.item = new String ("foo");
...
vl.result->append(vl.item);
...
```

There can be only one `value_local` declaration macro per C++ block.

Value locals variables **must** be explicitly dismissed before the block is exited. One way to do this is with the macro:

```
pop_value_locals();
```

At this point, the collector protection is lost. In some cases, however, you want to return one of the new values as the result and this value must continue to be protected from the collector. The macro:

```
return_value(r);
```

will do this. It both dismisses the value locals and places the return value in a protected area. This return value protection will last as the value passes out through any number of levels of return until the next `return_value()` use.

In another case, you may have taken a value out of some rooted structure (say an array in a MAXScript global) and want to hand that back as a function result. To protect it against being collected if, for example, that rooted structure gets dropped, you can use the macro:

```
return_protected(r);
```

which protects the return value as does `return_value()`, but doesn't dismiss any value locals.

All functions in the MAXScript core that return potentially new values, including all the **new** operators, use one of these return value protection schemes. Since this means that a new value returned to you is safe until another function call or operation that might create another new value, you can forego the `value_local` protection in those cases in which you are immediately returning the new value.

One thing to note about these protected variables is that they are considered to be part of the root set and **are** scanned by the collector for internal references, so you only need to protect the top-level object if you are constructing some tree of values.

There are some examples of these macros in use in the [Scripter-](#)

[Callable Functions](#) topic.

## Value Local Arrays (SDK)

Sometimes, the number of new locals to be protected is unknown at compile time. To handle this, you can use the following macros that create protected arrays of values. One form creates the array on the stack using `_alloca()` and so is useful inside one function call, the other puts the array in the heap and so can be used across functions. In both cases, you also need to declare a `Value**` variable to hold the pointer to the array.

### Stack-resident

```
value_local_array(pointer_var, count); // allocate
pop_value_local_array(pointer_var); // dismiss
```

### Heap-resident

```
value_temp_array(pointer_var, count); // allocate
realloc_value_temp_array(pointer_var, count, old_count); // grow
pop_value_temp_array(pointer_var); //dismiss
```

For example,

```
Value** items;
...
value_local_array(items, n);
...
for (int i = 0; i < n; i++)
items[i] = Float::intern(data[i]);
...
pop_value_local_array(items);
...
```

## Marking Values as Permanent or Collectable (SDK)

You can also mark a value as permanent by calling the `make_permanent()` member function on it:

```
vl.foo->make_permanent();
```

From then on the value is in the root set and will be scanned for internal references but is not eligible for collection. This operation is fairly heavy-weight and so should not be used in place of `value_local` variables.

You can also make a permanent value collectable again by calling the member function `make_collectable()`:

```
foo->make_collectable();
```

## Scripter-Callable Functions (SDK)

Scripter-callable functions are represented by instances of the Function family of classes. These descend from Value and so are all first-class values in MAXScript. For the moment, this document only describes non-polymorphic primitive functions, of which there are three kinds.

Primitive functions are defined using one of the three declaration macros:

```
def_visible_primitive(fn, name);  
def_mapped_primitive(fn, name);  
def_lazy_primitive(fn, name);
```

corresponding to the 3 primitive function types. Mapped primitives are automatically mapped over collection first arguments and lazy primitives have their arguments delivered as unevaluated code fragments (see the [Access to the Compiler and Interpreter](#) topic for more details).

The first argument to these macros is the implementing C++ function, the second is a string literal giving the scripter-visible name for the function. This string winds up naming a MAXScript global variable that contains the Primitive instance representing the function.

All the def\_x macros have several replaceable definitions in different header files. One is used to statically instantiate the Primitive object that represents the function, another to create externs in a header file. The convention throughout the MAXScript codebase is to gather sets of related scripter function definitions in 'protocol' files (eg, mathpro.h) and include in various places with different def\_x macro definitions in force.

The def\_x macro definition headers relevant to Primitive function

defs are as follows:

"defextfn.h" - generate externs from def\_x macros

"definsfn.h" - generate static instances from def\_x macros

A side-effect of using the instantiating form of the def\_ macros is that a scripter global variable named by the second argument is automatically created and loaded with the representing instance; you don't have to do anything else to make them visible in the scripter.

**Example:**

```
#include "definsfn.h"
def_visible_primitive( open_file,"openFile");
def_visible_primitive( query_box,"queryBox");
def_lazy_primitive ( quote,"quote");
def_mapped_primitive ( dump,"dump");
```

The implementing C++ functions you provide to these macros must have the following signature to conform to scripter calling conventions:

```
Value* func_cf(Value** arg_list, int count);
```

Further, the C++ function name is a derivative of the name given in the def\_x macro (to avoid some naming conflicts). It is constructed by appending '\_cf' to the name give, so:

```
def_visible_primitive( open_file, "openFile");
```

would refer to:

```
Value* open_file_cf(Value** arg_list, int count) { ... }
```

The arguments from the script call are delivered in call order in the arg\_list array which is count Value\*'s long. The function must return a Value\*, which for otherwise void functions, should be the distinguished value **ok**.

Note that the caller-protects-new-values convention means that values in the arg\_list array are already protected from the collector.

There is no implicit type or argument checking in this calling mechanism - you have to do it yourself in the called function. There are several macros to help with this:

```
check_arg_count(fn_name, wanted_count, got_count);
type_check(val, class, fn_name_or_description);
```

**Example:**

```
Value* open_file_cf(Value** arg_list, int count)
{
// check we have 1 arg and that it's a string
check_arg_count(openFile, 1, count);
type_check(arg_list[0], String, "openFile filename");
...
char* fn = arg_list[0]->to_string();
...
}
```

In this example, you could have left out the `type_check()` and let the `to_string()` converter complain if it didn't have a string, but the error message might be less descriptive.

MAXScript functions also support keyword argument passing. The keyword arguments are placed in the `arg_list` after all the positional args, preceded by a marker value (accessible in the `Value*` global, `keyarg_marker`). The arguments are passed in pairs, a keyword Name and the argument's value, one after the other in the `arg_list` array.

There are several macros to help extract keyword arguments. All assume that the C++ function parameter names are 'arg\_list' and 'count' as shown in the above examples.

```
key_arg(key);
    get the named arg value or &unsupplied if not present
```

```
key_arg_or_default(key, def);
```

get the named argument or the default value 'def' if not supplied

```
int_key_arg(key, var, def);
```

get named key arg into Value\* 'var' variable and return it as a C++ int, return 'def' if not supplied.

```
float_key_arg(key, var, def);
```

get named key arg into Value\* 'var' variable and return it as a C++ float, return 'def' if not supplied.

```
bool_key_arg(key, var, def);
```

get named key arg into Value\* 'var' variable and return it as a C++ BOOL, return 'def' if not supplied.

There is also a macro for checking positional argument counts in the presence of keyword args:

```
check_arg_count_with_keys(fn, wanted, got);
```

### Examples:

The following 'delete\_file' example shows an arg count check, conversion of first argument (arg\_list[0]) to a string and the return of MAXScript 'true' or 'false' values.

```
def_visible_primitive(delete_file, "deleteFile");
```

```
...
```

```
Value*
```

```
delete_file_cf(Value** arg_list, int count)
```

```
{
```

```
// deleteFile "file_name"
```

```
check_arg_count("deleteFile", 1, count);
```

```
BOOL result = DeleteFile(arg_list[0]->to_string());
```

```
return result ? &true_value : &>false_value;
```

```
}
```

The 'file\_in' example shows working with an optional keyword argument 'quiet'. The required arg count is checked with `check_arg_count_with_keys()` in this case.

Two typed value locals are declared, one to hold the temporary `FileStream` instance and the other to hold the `fileIn` result.

```
def_visible_primitive(file_in, "fileIn");

...

Value*
file_in_cf(Value** arg_list, int count)
{
// fileIn "filename" [quiet:true]

check_arg_count_with_keys("fileIn", 1, count);
two_typed_value_locals(FileStream* file, Value* result);
Value* quiet = key_arg_or_default(quiet, &true_value);
type_check(quiet, Boolean, "fileIn quiet:");
Value* result;

// open a temp fileStream
vl.file = (new FileStream)->open(arg_list[0]->to_string(), "rt");
if (vl.file == (FileStream*)&undefined)
throw RuntimeError
(GetString(IDS_FILEIN_CANT_OPEN_FILE),
arg_list[0]);

// pass it to the stream-based file_in
try
{
vl.result = file_in(vl.file, (quiet == &true_value));
}
}
```

```

catch (...)
{
// catch any errors and close the temp filestream
vl.file->close();
throw;
}

// pop value locals & return fileIn result
return_value(vl.result);
}

```

The implementation of the **file\_in()** function is given as another example later in these notes.

### Useful Globals

Interface\* MAXScript\_interface; -- MAX interface object

### Exported Value\* Virtual Functions

```

void print(); -- print representation to listener
void sprin1(CharStream* s); -- print representation on stream with
no
-- terminating \n

```

### Listener Window I/O

```

mputs(char*); -- print string to listener
mprintf(char* fmt, ...); -- printf to listener

```

### MAXScript Exception Classes

Compile and runtime errors are reported in MAXScript using the C++ exception-handling mechanism. A MAXScript exception class hierarchy is defined for these errors, as follows:

```

class MAXScriptException
class UnknownSystemException
class SignalException
class CompileError

```

```
class SyntaxError
class TypeError
class NoMethodError
class AccessorError
class AssignToConstError
class ArgCountError
class RuntimeError
class IncompatibleTypes
class ConversionError
```

**Common exception constructors:**

```
AccessorError (Value* target, Value* prop)
ArgCountError (char* fn_name, int wanted, int got);
ConversionError (Value* val, char* typename);
IncompatibleTypes (Value* v1, Value* v2);
RuntimeError (char* description);
RuntimeError (char* desc_part1, char* desc_part1);
RuntimeError (char* description, Value* implicated_value);
RuntimeError (char* desc_part1, char* desc_part1,
Value* implicated_value);
RuntimeError (Value* implicated_value);
TypeError (char* descr, Value* wrong_val,
ValueMetaClass* should_be = NULL);
```

An error is signalled by throwing one of these exceptions, for example:

```
if (arg_list[0]->to_int() > max_index)
throw RuntimeError ("Index out of range: ", arg_list[0]);
```

## Working with MAX Objects in the SDK (SDK)

All of the MAX-side objects accessible in MAXScript are represented by instances of the MAXWrapper family of classes. As the name suggests, these are wrappers for MAX object references that live in the garbage-collected MAXScript world. The wrapped reference is a standard MAX reference created with `ReferenceMaker::MakeRefByID()`; the MAXWrapper class inherits from both `Value` and `ReferenceMaker`. This reference allows MAXScript to be notified of MAX-side changes to the object, particularly deletion.

The wrapper class family is as follows:

- Value, ReferenceMaker
- MAXWrapper
- MAXNode
- MAXModifier
- MAXControl
- MAXKey
- MAXMaterial
- MAXMultiMaterial
- MAXTexture
- MAXMaterialLibrary
- MAXObject
- MAXSubAnim

There is also a **Value** subclass, **MAXBitMap**, which can be used to wrap MAX Bitmaps, but it is not a **ReferenceMaker** because Bitmaps in MAX are not referenceable in this way.

It should be clear which wrapper classes wrap which MAX-side entities, except perhaps for **MAXSubAnim** which is used to wrap objects you only know as Animatables that are accessible inside other

Max objects via the **Animatable::SubAnim()** function. This is used in MAXScript to wrap things like modifier gizmo sub-objects, for example.

## Constructing Wrappers (SDK)

There are three ways to make wrapper values for a MAX object you have in hand, depending on how much you know about what kind of object it is.

1. If you know its an **Object\***, a **Control\***, or **Modifier\***, etc., you can use a wrapper class constructor directly. The available constructors are:

```
MAXNode (INode* node);
MAXModifier (Modifier* mod);
MAXControl (Control* cont, ParamDimension* dim);
MAXControl (Control* cont);
MAXObject (Object* obj);
MAXKey (Control* icont, int ikey, ParamDimension* dim);
MAXKey (Control* icont, int ikey);
MAXMaterial (Mtl* imat);
MAXMultiMaterial (MultiMtl* imat);
MAXMaterialLibrary (MtlBaseLib& ilib);
MAXMaterialLibrary (MtlBaseLib* ilib);
MAXMaterialLibrary ();
MAXTexture (Texmap* imap);
MAXBitMap ();
MAXBitMap (BitmapInfo bi, Bitmap* bm);
```

1. If you know you have a **ReferenceTarget\***, you can use the static member function:

```
static MAXClass::make_wrapper_for(ReferenceTarget* ref);
```

This looks at ClassIDs and SClassIDs inside the object and chooses the appropriate wrapper class.

2. If you want to wrap a subanim inside a known ReferenceTarget\*, you can use the MAXSubAnim constructor:

```
MAXSubAnim (ReferenceTarget* ref, int subanim_num);
```

Remember, these all construct new MAXScript values in the MAXScript heap, subject immediately to potential collection. You have to use one of the protection schemes mentioned in the [Protecting Newly Created Values from the Collector](#) topic to ensure the value is not collected while you are still working with it.

## Retrieving Wrapped Objects (SDK)

You can get MAX objects out of MAXScript wrapper values in several ways:

1. Using one of the coercion virtual functions that are implemented on the appropriate wrapper classes:

```
virtual INode* to_node();  
virtual Mtl* to_mtl();  
virtual Texmap* to_texmap();  
virtual Modifier* to_modifier();  
virtual Control* to_controller();
```

For example,

```
INode* node = arg_list[0]->to_node();
```

2. Using the general purpose MAXWrapper virtual function, `get_max_object()`:

```
virtual ReferenceTarget* get_max_object();
```

Note that this function will retrieve the **base object** in a node inside a MAXNode wrapper value, rather than the INode itself.

The reason for this is explained in the [MAX ClassIDs and SuperclassIDs](#) topic.

3. Using the ReferenceMaker member function, `GetReference()`. All MAXWrapper objects store their MAX-side object reference as reference 0, so:

```
ref = arg_list[1]->GetReference(0);
```

## Handling Deleted Objects (SDK)

There is a potential in MAXScript for a wrapper value to reference a deleted MAX object. This might happen, for example, if a MAX global variable contains a MAXNode value and the user interactively deletes the object that the MAXNode references. It is critical for any code that works with wrapper values to first check to make sure the wrapped MAX objects are not deleted using the macro:

```
deletion_check(val);
```

which takes a MAXWrapper derived class instance. This macro will throw an appropriate runtime error if the MAX object referenced by 'val' is deleted. If you want to handle the error reporting yourself or perform some conditional code, you can test the MAXWrapper public data member, **ref\_deleted**, which holds a C++ BOOL.

You typically only need to do this checking once on entry to your code because scripts always run synchronously - MAXScript locks out the MAX UI so users cannot do things to the scene that would potentially crash a running script.

## MAX Class IDs and Superclass IDs (SDK)

You'll notice that the MAXScript wrapper classes correspond roughly to MAX object superclasses, there are no C++ classes in MAXScript for individual MAX object classes (Box, Sphere, Bend, StdMaterial, etc.). This is primarily because the MAX object classes present in any running copy of MAX is dynamic depending on the plug-ins that are loaded. MAXScript deals with this in much the same way the MAX SDK does by having descriptor classes whose instances define individual MAX plug-in classes. There are two MAXScript classes for this MAXScript-specific metadata, both Value subclasses so their instances can be manipulated as values in the scripter:

- Value
- MAXSuperClass
- MAXClass

The MAXSuperClass instances correspond to object superclasses in MAX and there is one instance per MAX superclass ID. The MAXClass instances correspond roughly to ClassDesc instances in MAX and there is one MAXClass instance per MAX Class ID.

All of the superclass instances and most of the MAXClass instances corresponding to the core objects in MAX are statically instantiated in MAXScript. There are also many MAXClass instances created dynamically by the MAXScript plug-in scanner which attempts to construct metadata for 3rd-party and new MAX plug-ins that it finds as MAX starts up.

MAXScript extension DLLs can statically define new MAXClass instances to provide complete metadata for new or 3rd-party MAX

plug-ins. The advantage of doing this is that you can typically provide a much more complete description for new MAX object classes than the plug-in scanner can construct.

These are declared as static instances using the MAXClass constructor described below. As an example, here is the definition for the Box primitive object from the MAXScript core:

```
MAXClass box
("Box", Class_ID(BOXOBJ_CLASS_ID, 0),
GEOMOBJECT_CLASS_ID, &geom_class, 0,
accessors,
paramblock,
"length", BOXOBJ_LENGTH, TYPE_FLOAT, 25.0,
"width", BOXOBJ_WIDTH, TYPE_FLOAT, 25.0,
"height", BOXOBJ_HEIGHT, TYPE_FLOAT, 25.0,
"widthsegs", BOXOBJ_WSEGS, TYPE_INT, 1,
"lengthsegs", BOXOBJ_LSEGS, TYPE_INT, 1,
"heightsegs", BOXOBJ_HSEGS, TYPE_INT, 1,
"mapcoords", BOXOBJ_GENUVS, TYPE_BOOL, FALSE,
end,
end,
end
);
```

This declares a static instance named 'box' that defines a scripter-visible name, the MAX class ID and superclass ID, and specifies the MAXSuperClass instance for this class and a set of MAX SDK ParamBlock-based properties. The property definitions define a scripter-visible name, parameter ID, type info and a useful default value, used when creating instances of the class in the scripter. The property definitions can also include properties not carried in ParamBlocks, for which you supply getter and setter C++ functions, so you can define any number of real or virtual properties.

As a side-effect of this declaration, a scripter global variable of the given name is created and loaded with the MAXClass value. Scripts can use this value both as a class value in methods such as [classOf\(\)](#) and to construct new MAX objects. MAXClass values, like several other class values in MAXScript can be 'applied' like a constructor in C++ to a set of arguments to create instances of themselves. In the case of MAXClass instances, all the properties you describe in its constructor can be used as creation arguments in MAXScript.

## The MAXClass Constructor (SDK)

The MAXClass constructor you should use for static instances is as follows:

```
MAXClass ( char* cname, // scripter-visible class name
Class_ID cid, // MAX class ID
SClass_ID sid, // MAX superclass ID
MAXSuperClass* superclass, // MAXScript MAXSuperClass
short cflags, // option flags (see below)
... // accessor definitions
);
```

The **cflags** value can be one or more of:

`md_no_create`

This class is not instantiable. Scripts cannot use this class as a constructor. Typically specified on object classes that cannot exist alone (such as spacewarp bindings) but are still accessible in the scripter.

`md_use_getref0`

`md_use_getref1`

`md_direct_index`

These three flags relate to ParamBlock-hosted properties. It turns out that core MAX objects use ParamBlocks in a variety of non-standard ways and these flags basically cover the possibilities. According to the docs, object classes that use a ParamBlock can implement

**Object::GetParamBlock()** to return a pointer to the block. Some MAX classes do and some don't. All of them stick the block in `reference(0)` or `reference(1)`, so if the class does not implement **GetParamBlock()** you can specify **md\_use\_getref0** or **md\_use\_getref1** accordingly. If neither of these flags are specified and there are paramblock accessors defined, MAXScript will use **GetParamBlock()**.

Further, the docs suggest that all classes should implement

**Object::GetParamBlockIndex()** to map logical parameter ID's into ParamBlock indexes. Again, only some of the MAX core classes do this

and some of them do it inconsistently. The **md\_direct\_index** flag indicates that the parameter IDs in the accessor definitions are direct indexes into the ParamBlock. If not this flag is not specified, MAXScript uses **GetParamBlockIndex()** to map the given IDs to paramblock indexes.

### md\_auto\_parms

This flag indicates that the properties defined are **in addition to** any properties that can be discovered by the MAXScript plug-in scanner. This gives you a way to incrementally add definitions for just those properties that are not housed in ParamBlocks or are not properly exposed there.

The property accessor definitions are supplied in the ... **varargs** to the MAXClass constructor. These take the form of tagged lists of specifiers each terminated with an 'end' tag. The tags are enumerated constants defined in MAXObj.h.

The syntax for these is as follows:

```
property_defs ::= [ <accessors>, ] end
accessors ::= accessors, [ <pb_props>, ] [ <fn_props> ]
pb_props ::= paramblock, { <pb_prop>, } end
pb_prop ::= <name_str> <param_id> <type> [ <default_val> ]
fn_props ::= fns, { <fn_prop>, } end
fn_prop ::= <name_str> <getter_fn> <setter_fn> <type>
[<default_val>]
```

Where:

<**param\_id**> is a ParamBlock parameter ID or ParamBlock index depending on whether the **md\_direct\_index** flag was specified.

<**type**> is one of:

TYPE\_MSFLOAT // float

TYPE\_INT // integer

TYPE\_RGBA // point3 of 0-255 externally, 0-1 internally

TYPE\_POINT3

TYPE\_BOOL

TYPE\_ANGLE // degrees externally, radians internally  
TYPE\_PCNT\_FRAC // percent externall, 0-1 float internally  
TYPE\_STRING  
TYPE\_HSV // point3 of 0-255 externally, 0-1 internally  
TYPE\_COLOR\_CHANNEL // 0-255 float externally, 0-1  
internally  
TYPE\_TIMEVALUE // frames externally, tick TimeValue  
internally  
TYPE\_UNSPECIFIED

Some of the types imply automatic scaling between scripter visible values and internal parameter values. For example, TYPE\_ANGLE causes radians-to-degree conversion on the away out to the scripter for property gets and degrees-to-radians conversion on the way in from the scripter for property sets.

TYPE\_UNSPECIFIED can only be used in **fn**s property definitions and means that the accessor functions take care of the type conversions and that there is no default value.

The <default\_val> default values **must be** of the correct C++ type or definition parsing may fail.

Default value C++ type  
TYPE\_MSFLOAT float  
TYPE\_INT int  
TYPE\_RGBA three comma-separated floats  
TYPE\_POINT3 three comma-separated floats  
TYPE\_BOOL BOOL  
TYPE\_ANGLE float  
TYPE\_PCNT\_FRAC float  
TYPE\_STRING string  
TYPE\_HSV three comma-separated floats  
TYPE\_COLOR\_CHANNEL float  
TYPE\_TIMEVALUE float

TYPE\_UNSPECIFIED -- no default value --

If you do not wish to specify a default value, perhaps because the property is a synonym for another which already has a default value specified, make the type -ve (put a minus in front of the type code) and leave out the default.

The signature for the getter and setter functions supplied in the **fns** section is:

```
Value* getter_fn(ReferenceTarget* obj, Value* prop, TimeValue t, Interval& valid);
```

```
void setter_fn(ReferenceTarget* obj, Value* prop, TimeValue t, Value* val);
```

When these functions are invoked by the MAXScript property accessor system, you can be sure the ReferenceTarget parameter points to a MAX object of the Class ID specified in the MAXClass constructor referencing these functions. The TimeValue and Interval parameters are used in the same way as they are on the Control and IParamBlock GetValue() and SetValue() member functions in the MAX SDK.

Here are some more examples:

```
MAXClass quadpatch  
("Quadpatch", Class_ID(PATCHGRID_CLASS_ID, 0),  
GEOMOBJECT_CLASS_ID,  
&geom_class, md_use_getref0 + md_direct_index,  
accessors,  
paramblock,  
"length", PATCHGRID_LENGTH, TYPE_MSFLOAT, 25.0,  
"width", PATCHGRID_WIDTH, TYPE_MSFLOAT, 25.0,  
"widthsegs", PATCHGRID_WSEGS, TYPE_INT, 1,
```

```

"lengthsegs", PATCHGRID_LSEGS, TYPE_INT, 1,
end,
end,
end
);
MAXClass text
("Text", Class_ID(TEXT_CLASS_ID, 0), SHAPE_CLASS_ID,
&shape, 0,
accessors,
paramblock,
"size", TEXT_SIZE, TYPE_FLOAT, 100.0,
end,
fns,
"text", get_txt_strng, set_txt_strng, TYPE_UNSPECIFIED,
"font", get_txt_font, set_text_font, TYPE_UNSPECIFIED,
"italic", get_txt_italic, set_txt_italic, TYPE_BOOL, FALSE,
"underline", get_txt_under, set_txt_under, TYPE_BOOL, FALSE,
end,
end,
end
);

```

The Text class defines both ParamBlock and function-based properties. It is typical only to specify type information for function-based properties when you want to give default values.

These are the MAXSuperClass static instances you can reference in MAXClass constructors. They are externed in the header file "**Maxcls.h**".

```

geom_object
modifier
shape

```

helper\_object  
spacewarp\_object  
light\_object  
camera\_object  
material\_class  
texture\_map  
system\_object  
utility\_plugin  
spacewarp\_modifier  
float\_controller  
point3\_controller  
position\_controller  
quat\_controller  
rotation\_controller  
scale\_controller  
matrix3\_controller  
morph\_controller

## **classOf() and superClassOf() Methods for MAX Objects (SDK)**

Even though the individual MAX object classes are not represented by C++ classes in MAXScript, they are considered classes at the scripter level, represented by the MAXClass values. Accordingly, the MAXScript class inquiry methods, **classOf()**, **superClassOf()** and **isAKindOf()**, are implemented in the C++ MAXWrapper classes to use the appropriate MAXClass instance as the class of the wrapped object when called on MAX object wrapper values, such as scene nodes, modifiers, or controllers. On scene nodes, the **classOf()** function always returns the class of the world-state of the object, its state at the top of the modifier stack.

### **Examples:**

```
classOf $foo.bend => Bend
superClassOf $foo.bend => Modifier
classOf $baz.position.controller => Bezier_position
superClassOf $baz.position.controller => Position_controller
```

In order to determine the class of the base object, you can use the **.baseObject** property of the scene node, for example on a line with an extrude modifier:

```
classOf $line01 => Editable_mesh
classOf $line01.baseObject => Shape
```

## Collection Mapping (SDK)

For Object Sets, Pathnames, Modifier/Key/NodeChildren Arrays and Arrays.

The following virtual functions are defined on Value and implemented by those classes that support them to provide a collection mapping mechanism. Those that support mapping return true from the virtual function predicate `is_collection()`. The MAXScript 'for x in y ...' loop statement relies on these mapping functions, for example. Attempting to map a non-collection results in a descriptive MAXScript runtime error.

Some of the functions are driven by a mapping structure, 'node\_map', defined below.

Only PathNames, ObjectSets and Nodes implement the '\_path' variants. Mapping over a scene node Value (MAXNode instance), effectively maps over its descendants.

### class Value

```
Value* map(node_map& m);
```

```
    map node structure over collection
```

```
Value* map_path(PathName* path, node_map& m);
```

```
    map node collection over values matching pathname
```

```
Value* find_first(BOOL (*test_fn)(INode* node, int level, void*
```

```
arg),
```

```
void* test_arg);
```

```
    find first node matching test
```

```
Value* get_path(PathName* path);
```

```
    get the non-wild-card pathname-specified node
```

```
struct node_map
```

```
{
```

```
value_vfvpn_ptr; // virtual fn to map
```

```
value_cfcfn_ptr; // or, c fn to map (one or the other must be null)
Value**arg_list; // args to pass on...
intcount;
BOOL(*selector)(INode*, int, void*); // set selector fn
void*sel_arg; // arg for the selector
intget_index; // index if we are doing an indexed get
intget_count; // local running traverse count during get
Value**get_result_p; // ptr to result holder for indexed get
Array*collection; // append map results here if non-null (used
// in for ... collect)
shortflags; // control flags
};
```

```
#define NM_INVERT0x0001 // invert map order, map parents last
on
// the way out of the recursion
#define NM_SELECT0x0002 // applying a select, adjust clear flag
#define NM_GET 0x0004 // doing a get, return get_index'th item
```

## Array Access and Construction (SDK)

Arrays are useful for passing multiple values back and forth; there is no multiple-value return mechanism in MAXScript.

Arrays in MAXScript are dynamic and will grow as necessary (by a factor of 1.5 when needed). You can set the initial extent size on creation. Uninitialized elements are set to **&undefined**.

As with all Value creation inside C++ code, a freshly created array must be protected from garbage collection, either by placing it immediately inside some other protected value, or by using the protection macros described in the [Protecting Newly Created Values from the Collector](#) topic.

```
Array(int init_size); // constructor allocates initial array size
```

```
Value* append(Value*);
```

```
Value* get(int index);
```

```
Value* map(node_map& m);
```

## Stream I/O (SDK)

MAXScript contains a number of character stream classes that are used primarily by the compiler and by the text I/O subsystem. These are all Value subclasses and so are first-class MAXScript values. If you want to call the compiler directly or pass or take stream values from scripts, you should use these classes.

The text file I/O classes and functions in MAXScript create and work on FileStream instances.

```
FileStream
FileStream::FileStream ();
FileStream* open(char* file_name, char* mode);
char get_char();
void unget_char(char c);
char peek_char();
int at_eos();
void rewind();
void flush_to_eol();
char putch(char c);
char* puts(char* str);
int printf(const char *format, ...);
void flush();
void close();
```

You can construct and pass StringStreams to the compiler if you want to compile pieces of source text.

```
StringStream
StringStream(char* init_string);
StringStream(Value* init_string_value);
StringStream(int ilen);
```

```
void init(char* init_string);  
char* puts(char* str);  
char putch(char c);  
int printf(const char *format, ...);
```

## Access to the Compiler and Interpreter (SDK)

The compiler in MAXScript can be invoked to compile source in files or strings into executable code. You gain access to it by creating an instance of the Parser class with the following constructor:

```
Parser(CharStream* errout);
```

Debugging and error output is sent to the given 'errout' charstream. Typically, you would specify the current standard output for this, which defaults to the Listener window, for example:

```
Parser* reader = new Parser (thread_local(current_stdout));
```

Any number of Parsers can be instantiated and active; they each encapsulate a separate compile environment including lexical scoping tables and source stream tracking.

The following Parser member functions are used to compile source code in a CharStream instance, either a FileStream or a StringStream (see the [Stream I/O](#) topic for details on constructing these streams).

```
Value* compile(CharStream* stream);
```

Compile the next complete expression in the given stream. The stream pointer is left at the end of the expression compiled, so you can repeatedly call compile() to step through source an expression at a time. Remember that MAXScript is expression-based, so 'expression' here means any MAXScript construct including all the control structures, function definitions, blocks, utility definitions, etc.

```
Value* compile_factor(CharStream* stream);
```

Compile the next MAXScript <operand> in the source stream. See the MAXScript Grammar topic for the exact definition of <operand>. The

MAXScript function **readValue()** uses **compile\_factor()** to get the next value in a text file, so it is capable of reading global variable references, array indexes, property references, pathnames, etc. as well as simple literals.

```
Value* compile_all(CharStream* stream);
```

Compile the whole stream into one piece of executable code.

The compile functions all return a MAXScript value containing the executable code, usually a CodeTree instance, but possibly a Number or String if the expression you give it to compile is a simple literal. The code is executed by invoking the Value virtual function **eval()** on it. For example,

```
Value* code = reader->compile(source); // compile an expr  
result = code->eval(); // execute it
```

In the case of simple values, this is a no-op and just returns the value. The code returned from the compiler is always a first-class MAXScript value that you can store for later execution or pass back to running scripts.

Here is the low-level **file\_in()** function in MAXScript, showing how the compiler and interpreter are used.

```
Value* file_in(CharStream* source, int quiet)  
{  
  three_typed_value_locals(Parser* parser, Value* code, Value*  
  result);  
  CharStream* out = thread_local(current_stdout);  
  vl.parser = new Parser (out);  
  if (!quiet)  
    source->log_to(out);  
  // loop through file compiling and evaluating all expressions
```

```
try
{
source->flush_whitespace();
while (!source->at_eos())
{
vl.code = vl.parser->compile(source);
vl.result = vl.code->eval();
if (!quiet)
vl.result->print();
source->flush_whitespace();
}
source->close();
}
catch (...)
{
// catch any errors and tell what file we are in if any
out->puts("Error occurred during fileIn: ");
source->sprin1(out);
out->puts("\n");
throw;
}

// return last expression in stream as result
if (vl.result == NULL)
vl.result = &ok;
return_value(vl.result);
}
```

## Calling Scripted Functions (SDK)

If you have a scripted function value in hand (an instance of the class `MAXScriptFunction`), you can invoke it directly using the `Value` virtual function **`apply()`**:

```
virtual Value* apply(Value** arglist, int count);
```

## MAXScript Header Files (SDK)

There are many header files supplied with the MAXScript SDK in the directory 'includes'. The following list shows the main ones you might use:

**Maxscript.h** The main MAXScript header. It includes most of the core headers required for the MAXScript API and 'Max.h', needed for any MAX SDK code.

**defextfn.h** Scripter-callable function definition macros. See the section **definsfn.h** in the [Scripter-callable functions](#) topic for details.

**MAXObj.h** The prime MAXWrapper header, used if you are constructing or accessing any of the main MAXScript values that represent MAX objects within MAXScript. See the [Working with MAX Objects in the SDK](#) topic for more details on this and the rest of the header files in this group.

**MAXKeys.h** MAXScript classes for wrapping MAX controller keys.

**MAXMats.h** MAXScript classes for wrapping MAX materials and texture maps.

**BitMaps.h** MAXScript class for wrapping MAX bitmaps.

**Maxclses.h** Defines all the static MAXClass and MAXSuperclass instances which provide MAXScript-specific metadata for the most of the core MAX classes.

The following headers define most of the foundation classes in MAXScript. Many of them are basically wrappers for primitive C++ types or MAX SDK classes, such as Point2, TimeValue, etc. You would include these headers if you need to create or access any instances of these classes.

3DMath.h All the MAXScript linear algebra classes

Arrays.h The Array class.

ColorVal.h MAXScript color class.

Funcs.h Scriptor functions.

HashTab.h MAXScript's internal hashtable class.

MSTime.h Time classes.

Name.h MAXScript's Name class.

Numbers.h Float and Integer classes.

Parser.h The MAXScript compiler.

Streams.h Character stream I/O, notable FileStream.

Strings.h Character string class.

## Defining New System Globals (SDK)

You can define new system global variables with the **define\_system\_global()** function. It has the following signature:

```
void define_system_global(TCHAR* name,  
Value* (*getter)(),  
Value* (*setter)(Value*));
```

Where **name** points to the string naming the new global and **getter** and **setter** are pointers to the getter and setter functions for the variable. For example:

```
{  
...  
define_system_global("frameRate", get_frame_rate,  
set_frame_rate);  
...  
}
```

...

```
Value*  
get_frame_rate()  
{  
return Integer::intern(GetFrameRate());  
}
```

```
Value*  
set_frame_rate(Value* val)  
{  
SetFrameRate(val->to_int());  
return val;  
}
```

## Core Names (SDK)

Instances of the class Name are used extensively throughout MAXScript as variable and parameter names, keyword argument tags, symbolic arguments, etc. Names are immutable, interned values - the same name string always (caselessly) interns to the same Name value. To save interning overhead, many common names are interned once during MAXScript startup and made available in C++ global variables.

The following list shows the pre-interned name globals with the convention that the name for the global is the interned name with an 'n\_' prefix.

- n\_about
- n\_across
- n\_active
- n\_align
- n\_allKeys
- n\_ambientmap
- n\_ambientmapamount
- n\_ambientmapenable
- n\_angle
- n\_aspect
- n\_axis
- n\_beep
- n\_bezier
- n\_bezierCorner
- n\_bias
- n\_bitmap
- n\_blur
- n\_bumpmap

n\_bumpmapamount  
n\_bumpmapenable  
n\_button  
n\_camera  
n\_cameras  
n\_caption  
n\_center  
n\_change  
n\_changed  
n\_checkbox  
n\_checkbutton  
n\_checked  
n\_children  
n\_close  
n\_color  
n\_colorPicker  
n\_columns  
n\_combobox  
n\_constant  
n\_constantVelocity  
n\_continuity  
n\_contrast  
n\_controller  
n\_coordsys  
n\_corner  
n\_current  
n\_curve  
n\_custom  
n\_cycle  
n\_default  
n\_determinantsign  
n\_diffusemap  
n\_diffusemapamount

n\_diffusemapenable  
n\_dir  
n\_display  
n\_distance  
n\_dontSort  
n\_dropdownList  
n\_easeFrom  
n\_easeTo  
n\_edge  
n\_edittest  
n\_enabled  
n\_entered  
n\_even  
n\_faces  
n\_fast  
n\_featureBounds  
n\_fieldorder  
n\_fieldwidth  
n\_filename  
n\_filter  
n\_filtermap  
n\_filtermapamount  
n\_filtermapenable  
n\_float  
n\_for  
n\_force2sided  
n\_frame  
n\_framerange  
n\_from  
n\_fromframe  
n\_gamma  
n\_geometry  
n\_grayscale

n\_grid  
n\_group  
n\_height  
n\_helpers  
n\_hidden  
n\_incleft  
n\_incrigh  
n\_inTangent  
n\_inTangentType  
n\_integer  
n\_interpolate  
n\_into  
n\_isAnimated  
n\_items  
n\_keep  
n\_keys  
n\_label  
n\_labels  
n\_left  
n\_length  
n\_lengthsegs  
n\_level  
n\_lights  
n\_line  
n\_linear  
n\_linkToKeys  
n\_listbox  
n\_local  
n\_loop  
n\_matchSpace  
n\_material  
n\_materialIDs  
n\_max

n\_message  
n\_middle  
n\_mode  
n\_motionBounds  
n\_name  
n\_netrender  
n\_noCloseButton  
n\_nodeTM  
n\_noslide  
n\_nthframe  
n\_numfaces  
n\_numframes  
n\_numsubs  
n\_numtrackers  
n\_numverts  
n\_object  
n\_objects  
n\_off  
n\_offset  
n\_on  
n\_opacitymap  
n\_opacitymapamount  
n\_opacitymapenable  
n\_open  
n\_outputdevice  
n\_outputfile  
n\_outputheight  
n\_outputOnly  
n\_outputSize  
n\_outputwidth  
n\_outTangent  
n\_outTangentType  
n\_palette

n\_parent  
n\_pickbutton  
n\_picked  
n\_pickedobject  
n\_pingPong  
n\_pivot  
n\_pixelaspect  
n\_point3  
n\_pos  
n\_position  
n\_pressed  
n\_quiet  
n\_radiobuttons  
n\_range  
n\_rank  
n\_rankDistance  
n\_redraw  
n\_reflectionmap  
n\_reflectionmapamount  
n\_reflectionmapenable  
n\_refractionmap  
n\_refractionmapamount  
n\_refractionmapenable  
n\_relativeRepeat  
n\_renderatmosphericeffects  
n\_renderfields  
n\_renderhiddenobjects  
n\_rgb  
n\_right  
n\_rightToLeft  
n\_rolledUp  
n\_rollout  
n\_rotation

n\_rotationpart  
n\_scale  
n\_scalepart  
n\_scalerotationpart  
n\_screen  
n\_select  
n\_selected  
n\_selectedtracker  
n\_selection  
n\_selfillummap  
n\_selfillummapamount  
n\_selfillummapenable  
n\_selOnly  
n\_shapes  
n\_shinestrengthmap  
n\_shinestrengthmapamount  
n\_shinestrengthmapenable  
n\_shininessmap  
n\_shininessmapamount  
n\_shininessmapenable  
n\_simple  
n\_size  
n\_slide  
n\_slow  
n\_smooth  
n\_spacewarps  
n\_specularmap  
n\_specularmapamount  
n\_specularmapenable  
n\_spinner  
n\_state  
n\_step  
n\_steps

n\_superblack  
n\_systems  
n\_target  
n\_tension  
n\_text  
n\_textureVerts  
n\_time  
n\_title  
n\_to  
n\_toframe  
n\_toolTip  
n\_transform  
n\_translationpart  
n\_type  
n\_value  
n\_vertices  
n\_vfb  
n\_videocolorcheck  
n\_width  
n\_widthsegs  
n\_world  
n\_worldUnits  
n\_x  
n\_x\_locked  
n\_x\_rotation  
n\_y  
n\_y\_locked  
n\_y\_rotation  
n\_z  
n\_z\_locked  
n\_z\_rotation

## Class ShortcutTable

See Also: [Structure ShortcutDescription](#), [Class ClassDesc](#), [Class ShortcutOperation](#), [Class ShortcutCallback](#).

class ShortcutTable

### Description:

This class is available in release 3.0 and later only.

This class represents a table of accelerators used by plug-ins. A plug-in can have various keyboard shortcut tables it uses. These tables hold a name that appears in the Preference Settings dialog / Keyboard tab / Plug-Ins dropdown list, a unique ID for the table, the number of shortcuts in the table, and storage for each of the Shortcut operations.

The number and a pointer to the 'i-th' shortcut table are returned by the methods **ClassDesc::NumShortcutTables()** and **ClassDesc::GetShortcutTable(int i)**.

For an example of using this mechanism see the code in the directory **\MAXSDK\SAMPLES\MODIFIERS\FFD**.

All methods of this class are implemented by the system.

### Methods:

### Prototype:

```
ShortcutTable(ShortcutTableId id, TSTR& name, HACCEL  
hDefaults, int numIds, ShortcutDescription* pOps, HINSTANCE  
hInst);
```

### Remarks:

Constructor. The data members are initialized to the values passed. A typical call to this constructor looks like this:

```
const ShortcutTableId kNURBSShortcuts = 0x34f274e4;  
#define NumElements(array) (sizeof(array) / sizeof(array[0]))
```

```
ShortcutTable *GetShortcuts() {  
TSTR name = GetString(IDS_JUST_NURBS);  
HACCEL hAccel = LoadAccelerators(hInstance,
```

```

MAKEINTRESOURCE(IDR_NURBS_SHORTCUTS));
int numOps = NumElements(spShortcuts);
ShortcutTable* pTab;
pTab = new ShortcutTable(kNURBSShortcuts,
name, hAccel, numOps, spShortcuts, hInstance);
return pTab;
}

```

**Parameters:**

**ShortcutTableId id**

This is the **unique** ID for the table. Typically developers use 'half' a ClassID as output by the ClassID Generator program. See the example above. Note:

**typedef long ShortcutTableId;**

**TSTR& name**

This is the name for the table which appears in the 3ds max UI in the Files / Preference Settings / Keyboard tab / Plug-In drop down list.

**HACCEL hDefaults**

This is the handle to the default keyboard accelerators. When a plug-in registers a shortcut table it needs to give the system a table of default shortcut assignments. It doesn't need to give all the operations default assignments, but it does need to give it a table of defaults. This table could be empty. This table is also used to implement the "Reset Category" button on the keyboard preferences dialog.

**int numIds**

This is the size of the array of shortcut descriptors specified below.

**ShortcutDescription\* pOps**

Points to the array of shortcut descriptors.

**HINSTANCE hInst**

The Dll instance handle of the plug-in.

**Prototype:**

**~ShortcutTable();**

**Remarks:**

Destructor. Any memory allocated by the constructor is freed here.

**Prototype:**

**HACCEL GetHAccel();**

**Remarks:**

Returns the handle of the currently active keyboard accelerator table.

**Prototype:**

**void SetHAccel(HACCEL hAccel);**

**Remarks:**

Sets the currently active keyboard accelerator table.

**Parameters:**

**HACCEL hAccel**

The handle to use.

**Prototype:**

**HACCEL GetDefaultHAccel();**

**Remarks:**

Returns the handle of the table of default shortcut assignments that are registered in the constructor for **ShortcutTable**.

**Prototype:**

**TSTR& GetName();**

**Remarks:**

Returns the name of the table to use in the Preference Settings / Keyboard tab / Plug-Ins drop down list.

**Prototype:**

**ShortcutTableId GetId();**

**Remarks:**

Returns the unique identifier of the table. Note: **typedef long ShortcutTableId;**

**Prototype:**

**int Count();**

**Remarks:**

Returns the number of shortcut operations in the table.

**Prototype:**

**void DeleteThis();**

**Remarks:**

Deletes this instance of the class.

**Prototype:**

**ShortcutOperation& operator[](int i);**

**Remarks:**

Provides access to the table of shortcut descriptions.

**Parameters:**

**int i**

The zero based index of the entry to get.

## Class ShortcutOperation

See Also: [Class ShortcutTable](#), [Class ShortcutCallback](#), [Class Interface](#), [Class ClassDesc](#).

### Description:

This class is available in release 3.0 and later only.

This class describes an operation that can be attached to a shortcut. Each shortcut stores an ID that is passed to the window proc when the shortcut is executed, a name for the shortcut operation that appears in the list of accelerators in a table, and an enabled/disabled state. Methods of this class provide access to each of these.

All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**ShortcutOperation();**

#### Remarks:

Constructor. The command ID is set to zero, the name is set to NULL and the enabled state is set to FALSE.

#### Prototype:

**int GetId();**

#### Remarks:

Returns the command ID sent to the window proc when the shortcut is executed.

#### Prototype:

**void SetId(int id);**

#### Remarks:

Sets the command ID sent to the window proc.

#### Parameters:

**int id**

The ID to set.

**Prototype:**

**TCHAR\* GetName();**

**Remarks:**

Returns the name of the operation the user sees in the Preference Settings / Keyboard tab / command list for Plug-Ins.

**Prototype:**

**void SetName(TCHAR\* pName);**

**Remarks:**

Sets the name of the operation the user sees.

**Parameters:**

**TCHAR\* pName**

The name to set.

## Class ShortcutCallback

See Also: [Class ShortcutOperation](#), [Class ShortcutTable](#), [Class Interface](#).

class ShortcutCallback

### Description:

This class is available in release 3.0 and later only.

An instance of a class derived from this class is passed into the method

### Interface:: ActivateShortcutTable().

When the user presses a shortcut key that is assigned in an active table, the system calls the **KeyboardShortcut(int id)** method of this class with the id of the operation.

**KeyboardShortcut()** should normally return TRUE. If the **KeyboardShortcut()** call returns FALSE, the system will ignore that shortcut and process any system shortcut for the key. This lets the plug-in selectively override system defined shortcuts. For example, NURBS use "H" to bring up a sub-object hit by name dialog when in a sub-object level, but at the top-level, "H" is ignored and the system hit-by-name dialog comes up instead.

The **KeyboardShortcut()** method is always where the keyboard shortcut will be handled. Note:There is no need for a window proc to handle the message as the old **RegisterAccelTable()** call required. The old **RegisterAccelTable()** functionality has not been removed (for backward source compatibility) , but new plug-ins should use the new ShortcutTable mechanism.

### Methods:

public:

### Prototype:

**virtual BOOL KeyboardShortcut(int id) = 0;**

### Remarks:

Implemented by the Plug-In.

This method is called when the user press a shortcut key assigned to an active table.

### Parameters:

**int id**

The id of the operation associated with the key.

**Return Value:**

TRUE to process the shortcut; FALSE to ignore the shortcut and process any system shortcut for the key.

## Structure ShortcutDescription

See Also: [Class ShortcutTable](#).

### Description:

This structure is available in release 3.0 and later only.

It provides a description of a command for building shortcut tables from static data. A pointer to an array of these is passed to the **ShortcutTable** constructor.

Typically one is declared like this:

```
static ShortcutDescription spShortcuts[] = {  
    ID_CURVE_TOGGLE, IDS_DISPLAY_CURVES,  
    ID_DISP_DEP_TOGGLE, IDS_DISPLAY_DEPENDENTS,  
    ...  
};
```

```
struct ShortcutDescription {
```

```
    int mCmdID;
```

This is the command ID.

```
    int mResourceID;
```

This is the resource ID of the description string.

```
};
```

## List of CUI Frame Position Types

See Also: [Class ICUIFrame](#).

One or more of the following flag values (which may be ORed together, as in **CUI\_HORIZ\_DOCK | CUI\_VERT\_DOCK | CUI\_FLOATABLE | CUI\_SM\_HANDLES**):

### **CUI\_TOP\_DOCK**

May be docked at the top.

### **CUI\_BOTTOM\_DOCK**

May be docked at the bottom.

### **CUI\_LEFT\_DOCK**

May be docked on the left.

### **CUI\_RIGHT\_DOCK**

May be docked on the right.

The flags below are combinations of those above:

### **CUI\_ALL\_DOCK**

May be docked in any of the positions above (same as

**(CUI\_TOP\_DOCK|CUI\_BOTTOM\_DOCK|CUI\_LEFT\_DOCK|CUI\_RIGHT\_DOCK)**)

### **CUI\_HORIZ\_DOCK**

May be docked at the top or the bottom.

**(CUI\_TOP\_DOCK|CUI\_BOTTOM\_DOCK)**

### **CUI\_VERT\_DOCK**

May be docked at the left or the right.

**(CUI\_LEFT\_DOCK|CUI\_RIGHT\_DOCK)**

The flags below control if the frame may be floated, connected, or should display handles:

### **CUI\_FLOATABLE**

The frame may be floated.

### **CUI\_FLOATING**

This is a synonym for **CUI\_FLOATING** above.

### **CUI\_CONNECTABLE**

This is not currently implemented.

### **CUI\_SM\_HANDLES**

Set this flag if frame should display size/move handles and it is resized/moved.

## List of CUI Frame Size Types

See Also: [Class ICUIFrame](#).

The size type. One of the following values:

**CUI\_MIN\_SIZE** - The minimum size.

**CUI\_MAX\_SIZE** - The maximum size.

**CUI\_PREF\_SIZE** - The preferred size. 3ds max does not currently take advantage of this size, only MIN and 3ds max are used.

## List of CUI Frame Orientations

See Also: [Class ICUIFrame](#).

The orientation. One or more of the following values:

**CUI\_HORIZ** - Horizontal orientation.

**CUI\_VERT** - Vertical orientation.

**CUI\_FLOAT** - As a floating toolbar.

## List of CUI Docking Panel Locations

See Also: [Class CUIFrameMgr](#).

One or more of the following values:

**CUI\_TOP\_PANEL**

**CUI\_BOTTOM\_PANEL**

**CUI\_LEFT\_PANEL**

**CUI\_RIGHT\_PANEL**

**CUI\_FIXED\_PANELS** -- This is the same as:

**(CUI\_TOP\_PANEL|CUI\_BOTTOM\_PANEL|CUI\_LEFT\_PANE**

**CUI\_FLOATING\_PANELS**

**CUI\_ALL\_PANELS** --This is the same as:

**(CUI\_FIXED\_PANELS|CUI\_FLOATING\_PANELS)**

## Class ICurve

See Also: [Class ReferenceTarget](#), [Class ICurveCtl](#), [Class CurvePoint](#), [COLORREF](#), [Class BitArray](#).

```
class ICurve : public ReferenceTarget
```

### Description:

This class is available in release 3.0 and later only.

This class is an interface to a single curve used by a **ICurveCtl**. A pointer to one of these is returned from the method **ICurveCtl::GetControlCurve()**. All methods of this class are implemented by the system.

### Methods:

public:

### Prototype:

```
virtual void SetPenProperty(COLORREF color, int width = 0, int style = PS_SOLID)=0;
```

### Remarks:

Sets the pen properties of a curve

### Parameters:

**COLORREF color**

The color for the curve lines.

**int width = 0**

The width of the lines in pixels.

**int style = PS\_SOLID**

The pen style to use. One of the following types may be used. See the Win32 API Reference for more information on pen styles.

**PS\_SOLID**

**PS\_DASH**

**PS\_DOT**

**PS\_DASHDOT**

**PS\_DASHDOTDOT**

**PS\_NULL**

## **PS\_INSIDEFRAME**

### **Prototype:**

**virtual void GetPenProperty(COLORREF &color, int &width, int &style)=0;**

### **Remarks:**

Retrieves the color, width and style of a curve.

### **Parameters:**

**COLORREF &color**

The color in use.

**int &width**

The width in use.

**int &style**

The style in use. One of the following types:

**PS\_SOLID**

**PS\_DASH**

**PS\_DOT**

**PS\_DASHDOT**

**PS\_DASHDOTDOT**

**PS\_NULL**

**PS\_INSIDEFRAME**

### **Prototype:**

**virtual void SetDisabledPenProperty(COLORREF color, int width = 0, int style = PS\_SOLID)=0;**

### **Remarks:**

Sets the pen properties of a curve if it is disabled.

### **Parameters:**

**COLORREF color**

The color to set.

**int width = 0**

The width to set.

**int style = PS\_SOLID**

The style to set. One of the following types:

**PS\_SOLID**

**PS\_DASH**

**PS\_DOT**

**PS\_DASHDOT**

**PS\_DASHDOTDOT**

**PS\_NULL**

**PS\_INSIDEFRAME**

**Prototype:**

**virtual void GetDisabledPenProperty(COLORREF &color, int &width, int &style)=0;**

**Remarks:**

Retrieves the color of a curve if it is disabled.

**Parameters:**

**COLORREF &color**

The color in use.

**int &width**

The width in use.

**int &style**

The style in use. One of the following types:

**PS\_SOLID**

**PS\_DASH**

**PS\_DOT**

**PS\_DASHDOT**

**PS\_DASHDOTDOT**

**PS\_NULL**

**PS\_INSIDEFRAME**

**Prototype:**

**virtual float GetValue(TimeValue t, float fX, Interval &ivalid = FOREVER, BOOL UseLookupTable = FALSE)=0;**

**Remarks:**

Returns the Y-value for a given X-Value of the curve. Note that values outside the X-range are extrapolated from the curve using a straight line based on the tangents of the first or last point.

**Parameters:****TimeValue t**

The time to get the value.

**float fX**

The input X value.

**Interval &ivalid = FOREVER**

The validity interval which is updated by this method to reflect the validity of the curve.

**BOOL UseLookupTable = FALSE**

If TRUE a lookup table is used to get the value (for speed). If FALSE the value is computed.

This is used to speed up value access. The default value for the lookup table size is 1000. The lookup table will be calculated whenever this method is called with TRUE and the current lookup table is not initialized yet, or invalid (it will be invalidated, when a point or tangent is moved, or the time has changed in case it is animated).

**Prototype:**

**virtual void SetCanBeAnimated(BOOL Animated)=0;**

**Remarks:**

Sets if the curve can be animated or not.

**Parameters:****BOOL Animated**

TRUE if it can be animated; FALSE if it can't.

**Prototype:**

**virtual BOOL GetCanBeAnimated()=0;**

**Remarks:**

Returns TRUE if the curve can be animated; otherwise FALSE.

**Prototype:**

**virtual int IsAnimated(int index)=0;**

**Remarks:**

Returns nonzero if the specified point is animated; otherwise zero.

**Parameters:**

**int index**

The zero based index of the curve to check.

**Prototype:**

**virtual int GetNumPts()=0;**

**Remarks:**

Returns the number of points in the curve.

**Prototype:**

**virtual void SetNumPts(int count)=0;**

**Remarks:**

Sets the number of points in the curve.

**Parameters:**

**int count**

The number to set.

**Prototype:**

**virtual BitArray GetSelectedPts()=0;**

**Remarks:**

Returns a BitArray which contains the selected / de-selected state for each point. The BitArray is **GetNumPts()** in size where the 0-th bit corresponds to the 0-th point.

**Prototype:**

```
virtual void SetSelectedPts(BitArray &sel, int flags)=0;
```

**Remarks:**

Sets the selected state of the points in the curve using the BitArray passed. Bits which are affected as specified by the flags.

**Parameters:****BitArray &sel**

Specifies which points are affected. The 0-th bit corresponds to the 0-th point.

**int flags**

One or more of the following values:

**SELPTS\_SELECT**

Select the points.

**SELPTS\_DESELECT**

De-select the points.

**SELPTS\_CLEARPTS**

If this bit is set the method de-selects (clears) all the points before performing the operation as specified by the flags above.

**Prototype:**

```
virtual void SetPoint(TimeValue t, int index, CurvePoint *point,  
BOOL CheckConstraints = TRUE, BOOL notify = TRUE)=0;
```

**Remarks:**

Sets the specified point at the time passed.

**Parameters:****TimeValue t**

The time at which to set the point.

**int index**

The zero based index of the point in the curve.

**CurvePoint \*point**

Points to the curve point to set.

**BOOL CheckConstraints = TRUE**

When you're setting a point, it checks the in and outtan handles to prevent them from going beyond the previous or next point's location, since that would

create an invalid curve. Thus the Curve Control will adjust the tangents due to the constraints. However, if you set the first point, there is no next point, to check the tangent location against, thus you have to have a possibility to turn CheckConstraints off, so the constraints won't be checked (developers have to make sure that they are inserting valid points/tangents. )

**BOOL notify = TRUE**

This parameter is available in release 4.0 and later only.

This allows developers to control whether the command sends windows messages or not. When set to FALSE windows messages are not sent to the message handler. This lets developers constrain points when the user attempts to move them without getting into a message loop. When TRUE messages are sent.

**Prototype:**

**virtual CurvePoint GetPoint(TimeValue t, int index, Interval &valid = FOREVER)=0;**

**Remarks:**

Retrieves data about the specified point and updates the validity interval to reflect the validity of the point's controller.

**Parameters:**

**TimeValue t**

The time to get the data.

**int index**

The zero based index of the point.

**Interval &valid = FOREVER**

The validity interval which is updated.

**Prototype:**

**virtual void SetOutOfRangeType(int type)=0;**

**Remarks:**

This method is available in release 4.0 and later only.

This method allows you to set the out of range type.

**Parameters:**

## **int type**

Currently these types are supported;

**CURVE\_EXTRAPOLATE\_LINEAR**

**CURVE\_EXTRAPOLATE\_CONSTANT**

### **Prototype:**

**virtual int GetOutOfRangeType()=0;**

### **Remarks:**

This method is available in release 4.0 and later only.

This method returns the out of range type.

### **Return Value:**

One of the following; **CURVE\_EXTRAPOLATE\_LINEAR**,  
**CURVE\_EXTRAPOLATE\_CONSTANT**

### **Prototype:**

**virtual int Insert(int where, CurvePoint &p)=0;**

### **Remarks:**

Inserts the specified point at the location passed.

### **Parameters:**

**int where**

This value becomes the new index of the point.

**CurvePoint &p**

The point to insert.

### **Return Value:**

Nonzero if the point was inserted; otherwise zero.

### **Prototype:**

**virtual int Insert(int where, CurvePoint& p, BOOL  
do\_not\_hold)=0;**

### **Remarks:**

This method is available in release 4.0 and later only.

This is identical to the Insert above but allows you to turn off/on the hold that

occurs.this is useful when you are doing interactive inserts and moves from code, the original Insert hold would often get in the way

**Parameters:**

**int where**

This value becomes the new index of the point.

**CurvePoint &p**

The point to insert.

**BOOL do\_not\_hold**

TRUE in order not to hold; otherwise FALSE.

**Return Value:**

Nonzero if the point was inserted; otherwise zero.

**Prototype:**

**virtual void Delete(int index)=0;**

**Remarks:**

Deletes the point whose index is passed.

**Parameters:**

**int index**

The zero based index of the point to delete.

**Prototype:**

**virtual void SetLookupTableSize(int size)=0;**

**Remarks:**

This method sets the size of the Curve Control lookup table. The lookup table allows users of the Curve Control to easily speed up their value access. The default value for the lookup table size is 1000. The lookup table will be calculated whenever **GetValue()** is called with the parameter **UseLookupTable==TRUE** and the current LookupTable is not initialized yet, or invalid (it will be invalidated, when a point or tangent is moved, or the time has changed in case it is animated).

**Parameters:**

**int size**

The size to set.

**Prototype:**

**virtual int GetLookupTableSize()=0;**

**Remarks:**

Returns the current size of the lookup table. See **SetLookupTableSize()** above.

# Class CurvePoint

See Also: [Class ICurve](#).

class CurvePoint

## Description:

This class is available in release 3.0 and later only.

This class stores data about a single point on a curve used by the custom curve control.

## Data Members:

public:

**Point2 p;**

The curve point.

**Point2 in;**

The in tangent, relative to **p**.

**Point2 out;**

The out tangent, relative to **p**.

**int flags;**

One or more of the following values which describes the type of curve point:

**CURVEP\_BEZIER**

Indicates the point is a bezier smooth point.

**CURVEP\_CORNER**

Indicates the point is a corner point.

To get a Bezier Corner use:

**CURVEP\_CORNER & CURVEP\_BEZIER**

**CURVEP\_LOCKED\_Y**

Indicates the point is locked in Y.

**CURVEP\_LOCKED\_X**

Indicates the point is locked in X.

**CURVEP\_SELECTED**

Indicates the point is selected.

**CURVEP\_ENDPOINT**

Indicates a constrained endpoint on the curve.

## **CURVEP\_NO\_X\_CONSTRAINT**

Indicates a point should not be constrained to X.

# Class ResourceMakerCallback

See Also: [Class ICurveCtl](#).

class ResourceMakerCallback

public:

## Description:

This class is available in release 3.0 and later only.

This callback object may be used to handle custom bitmaps and tooltips for the display buttons of a curve control. There are also methods which get called to handle the situations when a curve is reset or a new curve is created.

All methods of this class are implemented by the plug-in.

## Methods:

public:

## Prototype:

```
virtual BOOL SetCustomImageList(HIMAGELIST  
&hCTools,ICurveCtl *pCCtl);
```

## Remarks:

This method is called to set the image list.

This callback is used to update the HIMAGELIST to handle custom bitmaps on the display buttons. The image list has to have 2\*NumCurves 16x15 bitmaps. The format for the first set of images is for Out&In Enabled. The second set is for Out&In Disabled (which are not yet used).

## Parameters:

### **HIMAGELIST &hCTools**

A reference to the image list to set. An HIMAGELIST is a Win32 data type that is a handle to an image list.

### **ICurveCtl \*pCCtl**

This pointer can be used to determine which ICurveCtl calls the callback, in case the plugin uses many CurveControls and want to set different bitmaps for different CurveControls.

## Return Value:

If the image list was assigned the callback should return TRUE. If it returns FALSE, the default bitmaps will be used.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual BOOL GetToolTip(int iButton, TSTR &ToolTip,ICurveCtl  
*pCctl);
```

**Remarks:**

This callback allows the developer to assign custom ToolTips to the display buttons. One simply has to assign a string to the ToolTip parameter for the specified button number.

**Parameters:****int iButton**

The zero based index of the button whose tool tip text to retrieve.

**TSTR &ToolTip**

The string for the tool tip text.

**ICurveCtl \*pCctl**

This pointer can be used to determine which ICurveCtl calls the callback, in case the plugin uses many CurveControls and want to set different Tooltips for different CurveControls.

**Return Value:**

TRUE if the method is implemented to use custom tooltips; otherwise FALSE.

**Default Implementation:**

```
{return FALSE;}
```

**Prototype:**

```
virtual void ResetCallback(int curvenum, ICurveCtl *pCctl);
```

**Remarks:**

This methods gets called when the user preses the Reset button (cross symbol) in the user interface (if the control is using the upper toolbar).

**Parameters:****int curvenum**

The zero based index of the curve.

**ICurveCtl \*pCctl**

Points to the interface for the custom curve control.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void NewCurveCreatedCallback(int curvenum, ICurveCtl \*pCctl);**

**Remarks:**

This method gets called after the curve control creates a new curve. The developer can set the default values here for the new curve. This call will be a result of a call to **SetNumCurves(i)** where the new size is bigger than the old size.

**Parameters:**

**int curvenum**

The zero based index of the curve.

**ICurveCtl \*pCctl**

Points to the interface for the custom curve control.

**Default Implementation:**

**{}**

## List of Custom Curve Control Flags

See Also: [Class ICurveCtl](#).

These flags control various display aspects of the custom curve control. See the methods **ICurveCtl::SetCCFlags()** and **GetCCFlags()**.

Developers can easily review these by running the curve control utility CCUtil in `\MAXSDK\SAMPLES\UTILITIES\CCUTIL`.

One or more of the following values:

### **CC\_DRAWBG**

Specifies to draw the white background in the graph window.

### **CC\_DRAWGRID**

Specifies to draw the grid lines and coordinates in the graph window.

### **CC\_DRAWUTOOLBAR**

Specifies to draw the upper toolbar above the control.

### **CC\_SHOWRESET**

Specifies to display the Reset button in the upper toolbar.

### **CC\_DRAWLTOOLBAR**

Specifies to draw the lower toolbar beneath the control.

### **CC\_DRAWSCROLLBARS**

Specifies to draw the horizontal and vertical scroll bars for the control.

### **CC\_AUTOSCROLL**

Specifies to do auto scrolling. Auto scroll happens when you drag a CurvePoint out of the currently visible range. In case you're zoomed in, the window will automatically scroll.

### **CC\_DRAWRULER**

Specifies to draw a small moveable ruler window (measures horizontal coordinates).

### **CC\_ASPOPUP**

Specifies to create the control as a pop-up with a title string. Have the Window as popup window. If this is not set, it's important to set the `SetCustomParentWnd(HWND hParent)=0;` to the parent of the CurveControl Window.

### **CC\_CONSTRAIN\_Y**

Specifies that no points (or handles) can be moved out of the value that is

set by **SetYRange()**.

### **CC\_HIDE\_DISABLED\_CURVES**

Specifies that disabled curves won't be displayed at all. If this is not set, the disabled curves will be drawn by the color set by

**SetDisabledPenProperty()**.

Note: The following flags may be specified to enable a right click menu with choices used to activate the corresponding command mode (for instance if the upper toolbar is not present where these choice are also available).

### **CC\_RCMENU\_MOVE\_XY**

### **CC\_RCMENU\_MOVE\_X**

### **CC\_RCMENU\_MOVE\_Y**

### **CC\_RCMENU\_SCALE**

### **CC\_RCMENU\_INSERT\_CORNER**

### **CC\_RCMENU\_INSERT\_BEZIER**

### **CC\_RCMENU\_DELETE**

**CC\_SHOW\_CURRENTXVAL** (a flag to allow a vertical bar to be drawn over the graph to show the current X value)

Also note: One loses the automatic switch from insert to move mode when right-clicking, if a RightClick menu is active. Also note the following 'user' features:

- Pressing Control key while inserting keys will insert the opposite type of key than what's currently selected (bezier or corner).
- Pressing the Shift key while moving a tangent convertes a bezier key into a bezier corner key.

### **CC\_SINGLESELECT**

This flag allows the user to single select a point normally if a bunch of points are stacked in area if you click on the area you get all of them with this flag you get the first one.

### **CC\_NOFILTERBUTTONS**

This flag turns off the curve visible/editable toggle in the top of the menu

bar useful for when you have lots of curves and want to do the display management yourself.

## Class RightClickMenu

See Also: [Class RightClickMenuManager](#), [Class Interface](#), [Class IPoint2](#).

class RightClickMenu

### Description:

This class provides methods to work with the right click menu that pops up when the user right clicks over an item in a viewport. Methods of this class are used to initialize the menu and process the users selections. A developer derives a class from this class. For sample code see

**\\MAXSDK\\SAMPLES\\MODIFIERS\\EDITPATCH\\EDITPAT.CPP.**

Also see the method **Interface::GetRightClickMenuManager()**.

### Methods:

### Prototype:

```
virtual void Init(RightClickMenuManager* manager, HWND  
hWnd, IPoint2 m)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called when the user right clicks on an object in the scene. At this point you can determine what you need in the menu, and add these items using **manager->AddMenu()**.

### Parameters:

**RightClickMenuManager\* manager**

The menu manager. You may use this pointer to call methods of this class (for example to add items to the menu).

**HWND hWnd**

The window handle the user right clicked in.

**IPoint2 m**

The screen point the user right clicked on.

### Prototype:

```
virtual void Selected(UINT id)=0;
```

### Remarks:

Implemented by the Plug-In.

This method is called when the user has selected an item from the menu.

**Parameters:**

**UINT id**

This is the **id** of the users selection. This **id** is established when the developer calls **RightClickMenuManager::AddMenu()**.

## List of IGraphObjectManager Filter Bits

See Also: [Class IGraphObjectManager](#).

The filter bits: One or more of the following values:

### **SV\_FILTER\_SEOBJECTS**

Show only Selected objects.

### **SV\_FILTER\_OBJECTMODS**

Show Modified Objects.

### **SV\_FILTER\_BASEPARAMS**

Show Base Objects.

### **SV\_FILTER\_MATPARAMS**

Show Materials.

### **SV\_FILTER\_GEOM**

Show geometry nodes.

### **SV\_FILTER\_SHAPES**

Show shape nodes.

### **SV\_FILTER\_LIGHTS**

Show light nodes.

### **SV\_FILTER\_CAMERAS**

Show camera nodes.

### **SV\_FILTER\_HELPERS**

Show helper nodes.

### **SV\_FILTER\_WARPS**

Show space warps.

### **SV\_FILTER\_VISIBLE\_OBJS**

Show only Visible objects.

### **SV\_FILTER\_CONTROLLERS**

Show controllers.

### **SV\_FILTER\_ANIMATEDONLY**

Show only animated nodes.

### **SV\_FILTER\_MAPS**

Show maps.

### **SV\_FILTER\_BONES**

Show bone base objects.

## Class SClassUIInfo

See Also: [Class SubClassList](#), [COLORREF](#).

class SClassUIInfo

### Description:

This class is available in release 3.0 and later only.

This class allows developers to provide some additional information on a superclass. Currently this includes a color, and a method which draws a representative image in a Windows DC.

DrawRepresentation(...) can return false to indicate that no image was drawn. DrawRepresentation(...) should cache its image (if applicable) as the method is called repeatedly while drawing certain UI components (like the schematic view).

### Methods:

public:

### Prototype:

**virtual COLORREF Color(SClass\_ID superClassID);**

### Remarks:

Returns a color associated with the specified super class. This is currently used to draw nodes in the schematic view at extreme zoom-outs where it is impossible to draw legible node names.

### Parameters:

**SClass\_ID superClassID**

The Super Class whose associated color to return.

### Default Implementation:

**{ return RGB(128, 128, 128); }**

### Prototype:

**virtual bool DrawRepresentation(SClass\_ID superClassID,  
COLORREF bkColor, HDC hDC, Rect &rect);**

### Remarks:

Draws an image which represents the superclass (usually an icon) in a

rectangle in a given Windows DC. The implementation should attempt to draw the image as fast as possible as this method is called repeatedly while drawing certain UI components.

**Parameters:**

**SClass\_ID superClassID**

The super class to draw.

**COLORREF bkColor**

This is the average background color of the surface on which the image is being drawn. It can be used, if desired, to antialias the image.

**HDC hDC**

The handle to the device context.

**Rect &rect**

The rectangle to draw in.

**Return Value:**

Return **false** if no image was drawn and a generic stand-in image will be used.

**Default Implementation:**

```
{ return false; }
```

## List of Macro Recorder Value Types

See Also: [Class MacroRecorder](#), [Class Point3](#), [Class Matrix3](#), [Class Color](#), [Class AngAxis](#), [Class Quat](#), [Class BitArray](#), [Class PBitmap](#), [Class Class\\_ID](#), [Class ParamDimension](#).

One of the following macro recorder type tags followed by the argument(s) indicated. Depending upon the case there may be one, two or three arguments as noted:

### Basic C Types:

#### **mr\_int**

An integer. Follow the tag with the **int**.

Example code: **91462**

#### **mr\_float**

A floating point value. Follow the tag with a **double**.

Example code: **123.45**

#### **mr\_string**

A string. Follow the tag with the string (**TCHAR\***).

Example code: **"Benson"**

#### **mr\_bool**

A boolean. Follow the tag with a **int**.

Example code: **true**

### MAX SDK Types:

#### **mr\_point3**

A Point3. Follow the tag with a **Point3\***.

Example code: **[10,0,0]**

#### **mr\_color**

A Color. Follow the tag with a **Color\***.

Example code: **color 12 128 0**

#### **mr\_angaxis**

An AngAxis. Follow the tag with an **AngAxis\***.

Example code: **angleAxis 45[1,0,0]**

#### **mr\_quat**

A Quaternion. Follow the tag with a **Quat\***.

Example code: **quat 1 0 0 0**

### **mr\_time**

A TimeValue, follow the tag with a **TimeValue**.

Example code: **22f**

### **mr\_reftarg**

A Reference Target pointer. Follow the tag with a **ReferenceTarget\***.

Example code: **\$foo** or **\$baz.modifiers[2]** or **\$foo.material**, etc.

### **mr\_bitarray**

A bit array. Follow the tag with a pointer to a **BitArray\***.

Example code: **#{1..20, 30,34..100}**

### **mr\_pbitmap**

A parameter block2 bitmap object. Follow the tag with a **PBBitmap\***.

Example code: **bitMap 640 480 fileName:"fuz.bmp"**

### **mr\_matrix3**

A Matrix3. Follow the tag with a **Matrix3\***.

Example code: **matrix3 [1,0,0] [0,1,0] [0,0,1] [10,0,0]**

### **mr\_dimfloat**

A floating point value with the specified **ParamDimension**. Follow the tag with: **ParamDimension\***, **double**.

Example code: **123.45** (scaled by the dimension)

### **mr\_dimpoint3**

A Point3 value with the specified **ParamDimension**. Follow the tag with: **ParamDimension\***, **Point3\***.

Example code: **[10,0,0]** (scaled by the dimension)

### **mr\_classid**

A Class ID, follow the tag with a **Class\_ID** and a **SClass\_ID**.

Example code: **box** (the class variable name)

### **mr\_create**

A constructor call for the class. Follow the tag with: **Class\_ID**, **SClass\_ID**, **int**, **<args>**

The **int** parameter is the number of keyword args following, these are specified in pairs as **TCHAR\***, **<arg>**

Example code: **sphere radius:20 pos:[10,0,0]**

### **mr\_angle**

A floating point value with a **ParamDimension** of a **stdAngleDim**. Follow the tag with a **double**.

Example code: **90**

### **mr\_percent**

A floating point value with a **ParamDimension** of **stdPercentDim**. Follow the tag with a **double**.

Example code: **10**

## **MAXScript Types:**

### **mr\_sel**

A selection set. This tag works alone with no additional data. This tag denotes the current scene node selection and will emit as either a '\$' or an array of explicit objects, depending on the state of the **EmitAbsoluteSceneNames()** flag.

Example code: **\$** or **\$sphere01**

### **mr\_funcall**

A function call. Follow the tag with an **int, int, <args>**

The first int parameter is the number of positional arguments, given first in the **<args>** items, as **<arg>**, the second int parameter is the number of keyword args following the positional args in **<args>**, given as **TCHAR\*, <arg>** pairs.

Example code: **foo x y z output:f**

### **mr\_varname**

A variable name. Follow the tag with a string (**TCHAR\***).

Example code: **baz**

### **mr\_index**

An index. Follow the tag with: **<op\_arg>**, **<index\_arg>**

**<op\_arg>** is an **<arg>** specifying the operand to be indexed and

**<index\_arg>** is an **<arg>** specifying the index value.

Example code: **meditMaterials[3]**

### **mr\_prop**

A property name. Follow the tag with **TCHAR\***, **<arg>**. The **TCHAR\*** is the property name, the **<arg>** specifies the item being accessed.

Example code: **\$foo.pos**

### **mr\_nurbssel**

A NURBS selection level. Follow the tag with: **int, ReferenceTarget\***

The **int** parameter is the selection level desired of the specified NURBS base object.

Example code: **#{1..20, 50..100}**

This is an example of a BitArray literal in MAXScript, so it says sub-objects 1 through 20 and 50 through 100 are currently selected at the given level (the **int**) in the NURBS ReferenceTarget\* base object.

### **mr\_name**

A name. Follow the tag with a string (**TCHAR \***).

Example code: **#relational**

Note: A name in MAXScript is a simple symbolic value, often used instead of numbers to denote options in some function call. For example, when setting the motion blur of a scene node, one can say:

**\$foo.motionBlur = #image (or #none, or #object)**

generated, perhaps by:

**macroRecorder->SetSelProperty(\_T("motionBlur"), mr\_name, blurname);**

or when choosing g-buffer channels in the render() function, one would say:

**bm = render camera:\$c1 channels:##(zdepth, coverage, objectID)**

## Class IKSlaveControl

See Also: [Class Control](#), [Class IKMasterControl](#), [Class Point3](#), [Class Quat](#).

```
class IKSlaveControl : public Control
```

### Description:

This class is available in release 2.0 and later only.

The IK Controller requires that you use the Bones system. When you create the bones, a slave IK controller is assigned to each bone. All of the slave IK controllers in a single hierarchy are, in turn, controlled by a master IK controller. This class provides access to the slave controller parameters.

If you have a pointer to a TM controller you may test the Class\_ID against **IKSLAVE\_CLASSID** to determine if it is an IK Slave Controller.

For an example the use of this class see

**\MAXSDK\SAMPLES\OBJECTS\BONES.CPP.**

### Methods:

public:

### Prototype:

```
virtual IKMasterControl *GetMaster()=0;
```

### Remarks:

Returns a pointer to the **IKMasterControl** that manages this IK slave controller.

### Prototype:

```
virtual void SetDOF(int which, BOOL onOff)=0;
```

### Remarks:

Sets the specified degree of freedom to on or off.

### Parameters:

**int which**

Specifies which degree of freedom to modify. Pass a value between 1 and 6. The first 3 DOFs are position and the next 3 are rotation.

**BOOL onOff**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetInitPos(Point3 pos)=0;**

**Remarks:**

Sets the initial position to the value passed.

**Parameters:**

**Point3 pos**

The initial position to set.

**Prototype:**

**virtual void SetInitRot(Point3 rot)=0;**

**Remarks:**

Sets the initial rotation to the value passed.

**Parameters:**

**Point3 rot**

The initial rotation to set.

**Prototype:**

**virtual void MakeEE(BOOL onOff,DWORD which,Point3 pos,Quat rot)=0;**

**Remarks:**

For any IK solution you explicitly move one object. 3ds max uses IK calculations to move and rotate all other objects in the kinematic chain to react to the object you moved. The object that you move is the end effector. There are two types of end effectors; Position and Rotation. This method create or deletes an end effector for position or rotation or both.

**Parameters:**

**BOOL onOff**

Controls if the end effector is on or off (creates or deletes the end effector controller). TRUE to create; FALSE to delete.

**DWORD which**

Specifies which end effector(s) to modify. Set the low order bit for position, the second bit for rotation, or set both for position and rotation.

**Point3 pos**

The initial position set at time 0 for the position controller

(CTRL\_ABSOLUTE).

### **Quat rot**

The initial rotation set at time 0 for the rotation controller  
(CTRL\_ABSOLUTE).

**The following function is not part of this class but is available for use:**

### **Function:**

```
IKSlaveControl *CreateIKSlaveControl(IKMasterControl  
*master,INode *slaveNode);
```

### **Remarks:**

This global function creates and returns a pointer to a new IK slave controller by specifying the master controller and a node to which the slave controller is assigned.

### **Parameters:**

**IKMasterControl \*master**

Points to the master controller to use.

**INode \*slaveNode**

Points to the slave node to use.

## Structure ISect

See Also: [Class RenderGlobalContext](#), [Class RenderInstance](#), [Class ISectList](#).

### Description:

This structure is available in release 2.0 and later only.

This structure is updated by the **RenderGlobalContext::IntersectWorld()** and **RenderGlobalContext::IntersectRay()** methods. It stores information about the intersection of a ray and a single triangle in a mesh.

**struct ISect {**

**float t;**

The ray that was intersected has a unit vector specifying a direction. The ray defines an infinite line in the specified direction. If you take this value **t** and multiply it by the direction itself it gives a point on the ray. This specifies a distance along the vector. For instance if this way 5.0, the point would 5.0 units along the ray vector. This is the point of intersection along the ray.

**BOOL exit;**

TRUE if the ray is exiting the object; otherwise FALSE.

**BOOL backFace;**

TRUE if the ray hits a back face of the triangle; otherwise FALSE.

**RenderInstance \*inst;**

Points to the render instance associated with this triangle hit.

**int fnum;**

The face number of the triangle.

**Point3 bc;**

The barycentric coordinates of the intersection.

**Point3 p;**

The intersection point in object coordinates.

**Point3 pc;**

The intersection point in camera coordinates.

**ULONG matreq;**

The material requirements of the intersected face. See [List of Material Requirement Flags](#).

**int mtlNum;**

The material number for a multi-material.

**ISect \*next;**

Points to the next **ISect** structure in the list.

**};**

# Class ISectList

See Also: [Structure ISect](#).

class ISectList

## Description:

This class is available in release 2.0 and later only.

It provides a list of ray / triangle intersection structures (**struct ISect**). Methods are available for initializing the list, adding to the list, and selectively removing items from the list.

All methods of this class are implemented by the system.

Note the following global functions may be used to create and free instances of this class.

## Function:

**ISect \*GetNewISect();**

## Remarks:

This global function is available in release 2.0 and later only.

Returns a pointer to a new **ISect** structure.

## Function:

**void DiscardISect(ISect \*is);**

## Remarks:

This global function is available in release 2.0 and later only.

Deletes the **ISect** structure whose pointer is passed.

## Parameters:

**ISect \*is**

Points to the **ISect** structure to free.

## Data Members:

public:

**ISect \*first;**

A pointer to the first intersection.

## Methods:

**Prototype:**

**ISectList();**

**Remarks:**

Constructor. The list is set to empty.

**Prototype:**

**~ISectList();**

**Remarks:**

Destructor. The list is freed. **first** is set to NULL.

**Prototype:**

**BOOL IsEmpty();**

**Remarks:**

Returns TRUE if the list is empty; otherwise FALSE.

**Prototype:**

**void Add(ISect \*is);**

**Remarks:**

Adds the specified **ISect** to the list with the most distant layers first.

**Parameters:**

**ISect \*is**

Points to the **ISect** structure to add to the list.

**Prototype:**

**void Prune(float a);**

**Remarks:**

Removes the **ISect** structures from the list whose **t** values are less than or equal to the specified value **a**.

**Parameters:**

**float a**

The ray parameter used for comparison.

**Prototype:**

**void Init();**

**Remarks:**

Deletes any items from the list and sets **first** to NULL.

## List of NURBSTess Types

See Also: [Class NURBSSurface](#), [Class TessApprox](#), [Class NURBSSet](#).

One of the following enum values determines the tessellation type.

### **kNTessSurface**

Surface tessellation.

### **kNTessDisplacement**

Displacement tessellation.

### **kNTessCurve**

Curve tessellation.

## List of Render Instance Flags

See Also: [Class RenderInstance](#).

One or more of the following values:

### **INST\_HIDE**

Indicates the instance is hidden.

### **INST\_CLIP**

Indicates this is a clip instance: ray tracers should skip it .

### **INST\_BLUR**

Indicates this is a secondary motion blur instance: ray tracers can skip it.

### **INST\_RCV\_SHADOWS**

Indicates the instance receives shadows.

### **INST\_TM\_NEGPARITY**

Indicates the mesh is inside-out: need to reverse normals on-the-fly.

### **INST\_MTL\_BYFACE**

Indicates the instance's object supports the material-by-face interface. See [Class IChkMtlAPI](#).

## List of Patch Display Flags

One or more of the following values:

**DISP\_VERTTICKS**

Display vertices as tick marks

**DISP\_SELVERTS**

Display selected vertices.

**DISP\_SELPATCHES**

Display selected patches.

**DISP\_SELEDGES**

Display selected edges.

**DISP\_SELPOLYS**

Display selected polygons.

**DISP\_LATTICE**

Display the patch lattice.

**DISP\_VERTS**

This is used by the Edit Patch modifier to indicate if the vertices are displayed or not. This may be toggled on or off.

## Class SubPatchHitList

See Also: [Class PatchMesh](#), [Class PatchSubHitRec](#).

class SubPatchHitList

### Description:

This class describes a list of sub-patch hit records. Methods are available to return the first PatchSubHitRec in the list, and to add hits to the list. All methods of this class are implemented by the system.

### Methods:

#### Prototype:

**SubPatchHitList();**

#### Remarks:

Constructor. The first sub hit is set to NULL.

#### Prototype:

**~SubPatchHitList();**

#### Remarks:

Destructor. The list of patch hits are deleted.

#### Prototype:

**PatchSubHitRec \*First();**

#### Remarks:

Returns the first sub hit record.

#### Prototype:

**void AddHit(DWORD dist, PatchMesh \*patch, int index, int type)**

#### Remarks:

Creates a new sub hit record and adds it to the list.

#### Parameters:

**DWORD dist**

The distance of the hit. If the user is in wireframe mode, this is the distance in

pixels to the item that was hit. If the user is in shaded mode, this is the Z depth distance. Smaller numbers indicate a closer hit.

**PatchMesh \*patch**

The PatchMesh associated with this sub-patch hit.

**int index**

The index of the sub-object component. For example, if vertices were being hit tested, this would be the index into the vertex table.

**int type**

The type of the hit. One of the following values:

- PATCH\_HIT\_PATCH**
- PATCH\_HIT\_EDGE**
- PATCH\_HIT\_VERTEX**
- PATCH\_HIT\_VECTOR**
- PATCH\_HIT\_INTERIOR**

## Class IManipulatorMgr

See Also: [Class FPStaticInterface](#)

class IManipulatorMgr : public FPStaticInterface

### Description:

This class is available in release 4.0 and later only.

This is the abstract interface class for Manipulator Manager Objects

The Interface ID of this class is **MANIP\_MGR\_INTERFACE**.

### Methods:

public:

### Prototype:

```
virtual Mesh* MakeSphere(Point3& pos, float radius, int  
segments)=0;
```

### Remarks:

This method allows you to create a spherical mesh gizmo.

### Parameters:

**Point3& pos**

The position of the sphere.

**float radius**

The radius of the sphere

**int segments**

The number of segments in the sphere.

### Return Value:

A pointer to the resulting mesh.

### Prototype:

```
virtual Mesh* MakeTorus(Point3& pos, float radius, float radius2,  
int segs, int sides)=0;
```

### Remarks:

This method allows you to create a torus mesh gizmo.

**Parameters:****Point3& pos**

The position of the torus.

**float radius**

The radius of the torus.

**float radius2**

The second radius of the torus.

**int segs**

The number of segments in the torus.

**int sides**

The number of sides of the torus.

**Return Value:**

A pointer to the resulting mesh.

**Prototype:**

```
virtual Mesh* MakeBox(Point3& pos, float l, float w, float h, int  
lsegs, int wsegs, int hsegs)=0;
```

**Remarks:**

This method allows you to create a box mesh gizmo.

**Parameters:****Point3& pos**

The position of the box.

**float l**

The length of the box.

**float w**

The width of the box.

**float h**

The height of the box.

**int lsegs**

The length segments of the box.

**int wsegs**

The width segments of the box.

**int hsegs**

The height segments of the box.

**Return Value:**

A pointer to the resulting mesh.

**Prototype:**

**virtual Plane\* MakePlane()**=0;****

**Remarks:**

This method creates a default plane gizmo.

**Prototype:**

**virtual Plane\* MakePlane(Point3& p1, Point3& p2, Point3& p3)**=0;****

**Remarks:**

This method creates a plane gizmo.

**Parameters:**

**Point3& p1**

The first point of the plane.

**Point3& p2**

The second point of the plane.

**Point3& p3**

The third point of the plane.

**Prototype:**

**virtual Plane\* MakePlane(Point3& normal, Point3& point)**=0;****

**Remarks:**

This method creates a plane gizmo.

**Parameters:**

**Point3& normal**

The normal of the plane

**Point3& point**

The center point in space of the plane.

**Prototype:**

**virtual Plane\* GetmsXYPlane()=0;**

**Remarks:**

This method returns the XY plane.

**Prototype:**

**virtual Plane\* GetmsXZPlane()=0;**

**Remarks:**

This method returns the XZ plane.

**Prototype:**

**virtual Plane\* GetmsYZPlane()=0;**

**Remarks:**

This method returns the YZ plane.

**Prototype:**

**virtual GizmoShape\* MakeGizmoShape()=0;**

**Remarks:**

This method will create a default gizmo shape.

**Prototype:**

**virtual GizmoShape\* MakeCircle(Point3& center, float radius, int segments)=0;**

**Remarks:**

This method will make a circular gizmo shape.

**Parameters:**

**Point3& center**

The center of the circle.

**float radius**

The radius of the circle.

**int segments**

The number of segments of the circle.

**Return Value:**

A pointer to the resulting gizmo shape.

## Class ISimpleManipulator

See Also: [Class FPMixinInterface](#), [Class SimpleManipulator Class Interface](#), [Class PolyShape](#), [Class Mesh](#), [Class Point3](#), [Class Point2](#), [Class ViewExp](#), [Class IColorManager](#), [Class Ray](#), [Class ManipHitData](#), [List of Marker Types](#)

```
class ISimpleManipulator : public FPMixinInterface
```

### Description:

This class is available in release 4.0 and later only.

The ISimpleManipulator class is an interface to SimpleManipulators with built-in ParamBlock2 support and a variety of additionally useful methods.

### Methods:

public:

### Prototype:

```
virtual void ClearPolyShapes()=0;
```

### Remarks:

Implemented by the system.

Removes all of the current gizmos in the manipulator. This is normally called at the top of **UpdateShapes()** to clear out any previous gizmos before creating new ones.

### Prototype:

```
virtual void AppendPolyShape(PolyShape* pPolyShape, DWORD  
flags = 0, Point3& unselColor =  
GetUIColor(COLOR_SEL_GIZMOS), Point3& selColor =  
GetSubSelColor())=0;
```

### Remarks:

Implemented by the system.

This method adds a new PolyShape gizmo to the manipulator. The shape is defined in the local coordinates of the node that owns the manipulator.

### Parameters:

**PolyShape\* pPolyShape**

A pointer to the poly shape to add.

## **DWORD flags = 0**

The flags can have one or more of the following values:

### **kGizmoDontDisplay**

Instruct the gizmo not to display. It will still hit-test.

### **kGizmoDontHitTest**

Instruct the gizmo not to do any hit testing. It will still display.

### **kGizmoScaleToViewport**

Instruct the gizmo to scale itself to have a constant size in the viewport. In this case the system will use the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case.

## **Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

## **Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

## **Prototype:**

```
virtual void AppendGizmo(GizmoShape* pGizmoShape, DWORD  
flags = 0, Point3& unselColor =  
GetUIColor(COLOR_SEL_GIZMOS), Point3& selColor =  
GetSubSelColor())=0;
```

## **Remarks:**

Implemented by the system.

This method adds a new GizmoShape to the manipulator. The shape is defined in the local coordinates of the node that owns the manipulator.

## **Parameters:**

### **GizmoShape\* pGizmoShape**

A pointer to the gizmo shape to add.

### **DWORD flags = 0**

The flags can have one or more of the following values:

### **kGizmoDontDisplay**

Instruct the gizmo not to display. It will still hit-test.

### **kGizmoDontHitTest**

Instruct the gizmo not to do any hit testing. It will still display.

### **kGizmoScaleToViewport**

Instruct the gizmo to scale itself to have a constant size in the viewport. In this case the system will use the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case.

### **Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

### **Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

### **Prototype:**

```
virtual void AppendMesh(Mesh* pMesh, DWORD flags = 0,  
Point3& unselColor = GetUIColor(COLOR_SEL_GIZMOS),  
Point3& selColor = GetSubSelColor())=0;
```

### **Remarks:**

Implemented by the system.

This method adds a new mesh to the manipulator. The mesh is defined in the local coordinates of the node that owns the manipulator.

### **Parameters:**

#### **Mesh\* pMesh**

A pointer to the mesh to add.

#### **DWORD flags = 0**

The flags can have one or more of the following values:

#### **kGizmoDontDisplay**

Instruct the gizmo not to display. It will still hit-test.

#### **kGizmoDontHitTest**

Instruct the gizmo not to do any hit testing. It will still display.

#### **kGizmoScaleToViewport**

Instruct the gizmo to scale itself to have a constant size in the viewport. In this case the system will use the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case.

**Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

**Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

**Prototype:**

**virtual void AppendMarker(MarkerType markerType, Point3& position, DWORD flags = 0, Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS), Point3& selColor = GetSubSelColor())=0;**

**Remarks:**

Implemented by the system.

This method adds a new marker to the manipulator.

**Parameters:**

**MarkerType markerType**

The marker type to add.

**Point3& position**

The position of the marker.

**DWORD flags = 0**

The flags can have one or more of the following values:

**kGizmoDontDisplay**

Instruct the gizmo not to display. It will still hit-test.

**kGizmoDontHitTest**

Instruct the gizmo not to do any hit testing. It will still display.

**kGizmoScaleToViewport**

Instruct the gizmo to scale itself to have a constant size in the viewport. In this case the system will use the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case.

**Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

**Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

**Prototype:**

```
virtual void AppendText(TCHAR* pText, Point3& position,  
DWORD flags = 0, Point3& unselColor =  
GetUIColor(COLOR_SEL_GIZMOS), Point3& selColor =  
GetSubSelColor())=0;
```

**Remarks:**

Implemented by the system.

This method adds a new text entry to the manipulator.

**Parameters:**

**TCHAR\* pText**

The string containing the text to add.

**Point3& position**

The position of the text.

**DWORD flags = 0**

The flags can have one or more of the following values:

**kGizmoDontDisplay**

Instruct the gizmo not to display. It will still hit-test.

**kGizmoDontHitTest**

Instruct the gizmo not to do any hit testing. It will still display.

**kGizmoScaleToViewport**

Instruct the gizmo to scale itself to have a constant size in the viewport. In this case the system will use the **ManipulatorGizmo::mGizmoSize** to determine how big the manipulator should be. It interprets **mGizmoSize** as pixels in this case.

**Point3& unselColor = GetUIColor(COLOR\_SEL\_GIZMOS)**

The color of the gizmo when unselected.

**Point3& selColor = GetSubSelColor()**

The color of the gizmo when selected.

**Prototype:**

```
virtual MouseState GetMouseState()=0;
```

**Remarks:**

This method returns the current status of the mouse.

**Return Value:**

One of the following values:

**kMouseIdle**

The mouse is idle, manipulator not active and the mouse is not over it.

**kMouseDragging**

The mouse is currently dragging the manipulator.

**kMouseOverManip**

The mouse is over the manipulator, but it is not being dragged.

**Prototype:**

```
virtual void GetLocalViewRay(ViewExp* pVpt, IPoint2& m,  
Ray& viewRay)=0;
```

**Remarks:**

This method gets the view ray going through the given screen coordinate. The result is in local coordinates of the owning INode.

**Parameters:****ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

**IPoint2& m**

The screen coordinate.

**Ray& viewRay**

The returned local view ray.

**Prototype:**

```
virtual void UpdateShapes(TimeValue t, TSTR& toolTip)=0;
```

**Remarks:**

Implemented by the Plug-In.

This method gets called whenever the manipulator needs to update its gizmos.

This is implemented by the manipulator to create the gizmos based on the current state of the node being manipulated.

**Parameters:****TimeValue t**

The time value at which the shapes should be updated.

## **TSTR& toolTip**

The tooltip string.

### **Prototype:**

```
virtual void OnMouseMove(TimeValue t, ViewExp* pVpt,  
IPoint2& m, DWORD flags, ManipHitData* pHitData)=0;
```

### **Remarks:**

This method gets called when the mouse moves within the manipulator context.

### **Parameters:**

#### **TimeValue t**

The time to display the object.

#### **ViewExp\* pVpt**

An interface that may be used to call methods associated with the viewports.

#### **IPoint2& m**

The screen coordinates of the mouse.

#### **DWORD flags**

Not used, should be set to 0.

#### **ManipHitData\* pHitData**

A pointer to the hitdata containing information on which manipulator was hit.

## Class GizmoShape

See Also: [Class SimpleManipulator](#), [Class ISimpleManipulator](#), [Class FPMixinInterface](#), [Class PolyLine](#), [Class PolyShape](#)

```
class GizmoShape : public FPMixinInterface
```

### Description:

This class is available in release 4.0 and later only.

This class represents the main gizmo shape.

The Function Publishing interface to SimpleManipulators is defined as:

```
#define MANIP_GIZMO_INTERFACE Interface_ID(0x124e3169,  
0xf067ad4)
```

### Data Members:

private:

```
PolyShape mPolyShape;
```

The gizmo polyshape.

```
PolyLine mLine;
```

A poly line.

### Methods:

public:

### Prototype:

```
GizmoShape();
```

### Remarks:

Constructor.

### Default Implementation:

```
{ mLine.Init(); }
```

### Prototype:

```
void StartNewLine();
```

### Remarks:

This method instructs the gizmo shape to start (append) a new line segment.

**Prototype:**

**AppendPoint(Point3& p);**

**Remarks:**

This method instructs the gizmo shape to append a new point to the line segment.

**Parameters:**

**Point3& p**

The vertex point to add.

**Prototype:**

**PolyShape\* GetPolyShape();**

**Remarks:**

This method returns a pointer to the gizmo's poly shape.

**Prototype:**

**Transform(Matrix3& tm);**

**Remarks:**

This method allows you to transform the gizmo shape.

**Parameters:**

**Matrix3& tm**

The transformation matrix.

## Class ManipHitData

See Also: [Class HitData](#), [Class Manipulator](#)

```
class ManipHitData : public HitData
```

### Description:

This class is available in release 4.0 and later only.

This is a special storage class for hit records that keep track of which manipulator was hit.

### Data Members:

public:

**Manipulator\* mpManip;**

A pointer to the manipulator associated with the hit data.

**int mShapeIndex;**

The index of the selected manipulator.

### Methods:

public:

### Prototype:

**ManipHitData(Manipulator\* pManip);**

### Remarks:

Constructor.

### Parameters:

**Manipulator\* pManip**

A pointer to the manipulator to use.

### Prototype:

**ManipHitData();**

### Remarks:

Constructor.

### Prototype:

**~ManipHitData();**

**Remarks:**

Destructor.

**Prototype:**

**virtual ManipHitData\* Copy();**

**Remarks:**

This method allows you to copy the hit data.

**Default Implementation:**

**{ return new ManipHitData(mpManip); }**

## Class ToneOperatorInterface

See Also: [Class FPStaticInterface](#), [Class ToneOperator](#).

```
class ToneOperatorInterface : public FPStaticInterface
```

### Description:

This class is available in release 4.0 and later only.

This class allows plug-ins and the scripter to get access to the tone operator assigned to a scene. You can get a pointer to the interface using the global interface pointer in this manner:

```
ToneOperatorInterface* toneOpInt =  
static_cast<ToneOperatorInterface*>  
( GetCOREInterface(TONE_OPERATOR_INTERFACE));
```

If the return value is NULL, the running version of 3ds max doesn't support tone operators. If the return value is not NULL, you can use these methods to perform some scene management.

### Methods:

public:

### Prototype:

```
virtual void OpenToneOperatorPanel()=0;
```

### Remarks:

This method opens the tone operator panel UI. If the panel has been minimized, it is restored.

### Prototype:

```
virtual void CloseToneOperatorPanel()=0;
```

### Remarks:

This method closes the tone operator panel UI.

### Prototype:

```
virtual void MinimizeToneOperatorPanel()=0;
```

### Remarks:

This method minimizes the tone operator panel UI.

**Prototype:**

```
virtual ToneOperator* GetToneOperator() const = 0;
```

**Remarks:**

This method returns the current tone operator assigned to a scene. If no tone operator is assigned NULL is returned.

**Prototype:**

```
virtual void SetToneOperator(ToneOperator* op) = 0;
```

**Remarks:**

This method assigns a tone operator to the scene. To remove a tone operator, assign NULL. When a new tone operator is assigned the current operator is removed automatically.

**Prototype:**

```
virtual void  
RegisterToneOperatorChangeNotification(ToneChangeCallback  
callback, void* param) = 0;
```

**Remarks:**

This method registers a callback that is called when the tone operator is changed. Note the definition of ToneChangeCallback:

```
typedef void (*ToneChangeCallback)(ToneOperator* newOp,  
ToneOperator* oldOp, void* param);
```

**Parameters:**

**ToneChangeCallback callback**

The callback to register.

**void\* param**

This parameter is passed to the callback function as the parameter argument when it is called.

**Prototype:**

```
virtual void  
UnRegisterToneOperatorChangeNotification(ToneChangeCallback  
callback, void* param) = 0;
```

**Remarks:**

This method un-registers a callback that was registered by RegisterToneOperatorChangeNotification. Note the definition of ToneChangeCallback:

```
typedef void (*ToneChangeCallback)(ToneOperator* newOp,  
ToneOperator* oldOp, void* param);
```

**Parameters:**

**ToneChangeCallback callback**

The callback to un-register.

**void\* param**

This parameter is passed to the callback function as the parameter argument when it is called.

## Class IReshadeFragment

See Also: [Class InterfaceServer](#), [Class IReshading](#), [Class Point2](#), [Class Point3](#)

```
class IReshadeFragment : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

The Reshade Fragment interface is the materials/shaders/textures interface to reshading fragments. This interface is only concerned with saving and retrieving values in the fragments variable sized cache. Values are always saved in multiples of 32 bit words, and they can be a variety of types, colors, float's, int's, etc. Unit vectors are compressed to a single 32 bit word. Point2 and Point3 data types save multiple floats.

Color channels will be compressed to a 32 bit rgba color and saved in the cache. Color32 channels are only used internally. This is the 'raw' form of what's stored, other forms may be accessed from this class.

Float and integer channels are stored uncompressed, they are stored directly in the 32 bits.

The Point2 & Point3 channels are shells that store multiple float channels so these values will be uncompressed. These should be used with great caution, as this memory is stored per fragment, in a scene with 100,000 fragments of a given material, a point3 channel will add 1.2 megabytes to the storage for the reshading buffer.

### Methods:

public:

### Prototype:

```
virtual int NChannels()=0;
```

### Remarks:

This method returns the number of textures for the fragment.

### Prototype:

```
virtual int NFirstChannel()=0;
```

### Remarks:

This method returns the index of the first channel.

**Prototype:**

**virtual int NTextures()=0;**

**Remarks:**

This method returns the number of textures.

**Prototype:**

**virtual Color32 Channel(int nChan)=0;**

**Remarks:**

This method returns the specified channel.

**Parameters:**

**int nChan**

The channel you wish to return.

**Prototype:**

**virtual void SetChannel(int nChan, Color32 c)=0;**

**Remarks:**

This method allows you to set a channel in raw mode to a new value.

**Parameters:**

**int nChan**

The channel you wish to set.

**Color32 c**

The new value to set.

**Prototype:**

**virtual void AddChannel(Color32 c)=0;**

**Remarks:**

This method allows you to add a new color channel in raw mode to the end of the cache.

**Parameters:**

### **Color32 c**

The channel value you wish to set.

#### **Prototype:**

**virtual void AddColorChannel(RGBA tex)=0;**

#### **Remarks:**

This method allows you to add a new color channel to the end of the cache.  
The color will be compressed to 32bits.

#### **Parameters:**

**RGBA tex**

The color channel value you wish to add.

#### **Prototype:**

**virtual void AddFloatChannel(float f)=0;**

#### **Remarks:**

This method allows you to add a float channel in the fragment cache.

#### **Parameters:**

**float f**

The channel value to add.

#### **Prototype:**

**virtual void AddIntChannel(int i)=0;**

#### **Remarks:**

This method allows you to add an integer channel in the fragment cache.

#### **Parameters:**

**int i**

The integer value to add.

#### **Prototype:**

**virtual void AddUnitVecChannel(Point3 v)=0;**

**Remarks:**

This method allows you to add a unit vector channel in the fragment cache, compressed to 32-bits.

**Parameters:****Point3 v**

The unit vector to add.

**Prototype:****virtual void AddPoint2Channel(Point2 p)=0;****Remarks:**

This method allows you to add a Point2 channel in the fragment cache, uncompressed.

**Parameters:****Point2 p**

The Point2 to add.

**Prototype:****virtual void AddPoint3Channel(Point3 p)=0;****Remarks:**

This method allows you to add a Point3 channel in the fragment cache, uncompressed.

**Parameters:****Point3 p**

The Point3 to add.

**Prototype:****virtual void SetColorChannel(int nChan, RGBA tex)=0;****Remarks:**

This method sets the existing color channel number **nChan** to the new value.

**Parameters:****int nChan**

The color channel number.

## **RGBA text**

The new color value.

### **Prototype:**

**virtual void SetFloatChannel(int nChan, float f)=0;**

### **Remarks:**

This method allows you to set a float channel in the fragment cache to a new value.

### **Parameters:**

**int nChan**

The float channel number.

**float f**

The new value to set.

### **Prototype:**

**virtual void SetIntChannel(int nChan, int i)=0;**

### **Remarks:**

This method allows you to set an int channel in the fragment cache to a new value.

### **Parameters:**

**int nChan**

The int channel number.

**int i**

The new value to set.

### **Prototype:**

**virtual void SetUnitVecChannel(int nChan, Point3 v)=0;**

### **Remarks:**

This method allows you to set a unit vector channel in the fragment cache to a new value.

### **Parameters:**

**int nChan**

The unit vector channel number.

**Point3 v**

The new value to set.

**Prototype:**

**virtual void SetPoint2Channel(int nChan, Point2 p)=0;**

**Remarks:**

This method allows you to set a Point2 channel in the fragment cache to a new value.

**Parameters:**

**int nChan**

The Point2 channel number.

**Point2 p**

The new value to set.

**Prototype:**

**virtual void SetPoint3Channel(int nChan, Point3 p)=0;**

**Remarks:**

This method allows you to set a Point3 channel in the fragment cache to a new value.

**Parameters:**

**int nChan**

The Point3 channel number.

**Point3 p**

The new value to set.

**Prototype:**

**virtual RGBA GetColorChannel(int nChan)=0;**

**Remarks:**

This method will return an existing color channel, expanded to float RGBA.

**Parameters:**

**int nChan**

The color channel you wish to return.

**Prototype:**

**virtual float GetFloatChannel(int nChan)=0;**

**Remarks:**

This method will return an existing float channel.

**Parameters:**

**int nChan**

The float channel you wish to return.

**Prototype:**

**virtual int GetIntChannel(int nChan)=0;**

**Remarks:**

This method will return an existing int channel.

**Parameters:**

**int nChan**

The int channel you wish to return.

**Prototype:**

**virtual Point3 GetUnitVecChannel(int nChan)=0;**

**Remarks:**

This method will return an existing unit vector channel.

**Parameters:**

**int nChan**

The unit vector channel you wish to return.

**Prototype:**

**virtual Point2 GetPoint2Channel(int nChan)=0;**

**Remarks:**

This method will return an existing Point2 channel.

**Parameters:**

**int nChan**

The Point2 channel you wish to return.

**Prototype:**

**virtual Point3 GetPoint3Channel(int nChan)=0;**

**Remarks:**

This method will return an existing Point3 channel.

**Parameters:**

**int nChan**

The Point3 channel you wish to return.

# Class IllumParams

See Also: [Class Shader](#), [Class Point3](#), [Class Color](#).

```
class IllumParams : public BaseInterfaceServer
```

## Description:

This class is available in release 3.0 and later only.

This class allows the Standard material to get the parameters from a Shader plug-in. It also allows the Shader to supply its computed color data back to the Standard material. The Standard material will handle the texturing but it needs to know the color before the texturing is done.

The shader object holds some raw parameters. But these parameters may be textured. So for each pixel that will be shaded, the 3ds max shade routine asks the shader to fill in the raw values. Then it applies the texture over the top of any of the values which are textured.

## Data Members:

public:

### **Shader\* pShader;**

This parameter is available in release 4.0 and later only.

A pointer to the shader.

### **Mtl\* pMtl;**

This parameter is available in release 4.0 and later only.

A pointer to the material being shaded or NULL if it's a background.

### **Color channels[32];**

A color channel for each of the possible parameters that may be textured. Note that these channels don't have specific meanings. They are defined by each Shader. The Shader tells the Standard material what data is in these channels using its **ChannelType()** and **StdIDToChannel()** methods.

### **float falloffOpac;**

Shaders using standard opacity can ignore this data member. This is available for certain Shaders which don't use the standard 3ds max transparency equation. This allows these Shaders to simulate the 3ds max version. This is the pre-computed textured opacity with standard falloff applied. The value here is pre-computed to consider all the standard opacity settings of falloff direction, etc. The standard transparency computation then uses this after

shading. So a shader could modify this value if it wanted to affect the transparency.

Note: The regular opacity can be obtained from the channel data.

### **float kR;**

This value is used when there is a reflection and an atmosphere present. Normally if there is no atmosphere (for instance no Fog in the scene) then the transparency of the reflection is 100% (it is unaffected). However, if there is an atmosphere present it will impart some level of opacity. This opacity (alpha value) is not available via the reflection color. Thus this data member is here to provide this opacity information to the Shader.

This value is the alpha which is returned by the reflection query that has the transparency of the atmosphere contained within it, which is then multiplied by the Amount spinner. This can then be used either by the Shader or by the standard handling for reflection maps.

### **ULONG hasComponents;**

These are the bits for the active components of bump, reflection, refraction and opacity mapping. If the bit is set that component is active. This provides a quick way for a Shader to check if they're used (as opposed to looking through the **channels** array searching for these channel types).

**HAS\_BUMPS** -- If bump mapping is present this is set.

**HAS\_REFLECT** -- If there is any kind of reflection (raytraced, etc) then this is set.

**HAS\_REFRACT** -- If there is any kind of refraction then this is set..

**HAS\_OPACITY** -- If opacity mapping is used this is set.

**HAS\_REFLECT\_MAP** -- If there is a reflection map only this is set. This is used by the Straus shader for example. If it sees a reflection map present it dims the diffuse channel.

**HAS\_REFRACT\_MAP** -- If there is a refraction map only then this is set.

### **ULONG stdParams;**

The standard parameter bits. See [List of Shader Standard Parameter Flags](#). This is filled in by the Standard material.

### **Color ambIllumOut;**

This is the ambient output from the **Illum()** method.

**Color diffIllumOut;**

This is the diffuse output from the **Illum()** method.

**Color transIllumOut;**

This is the transparency output from the **Illum()** method.

**Color selfIllumOut;**

This is the self illumination output from the **Illum()** method.

**Color specIllumOut;**

This is the specular illumination output from the **Illum()** method.

**Color reflIllumOut;**

This is the reflection output from the **Illum()** method. Certain shaders may wish to store the reflection output here (as opposed to providing it in the **channels** array). This is the "raw " color from the direction of reflection (unattenuated). Some combiner implementations (**Shader::CombineComponents()**) can get the reflection data here, others may get it from the channels.

**float diffIllumIntens;**

Used only by reflection dimming, intensity of diffIllum prior to color multiply.

**float finalAttenuation;**

The final attenuation for combining components.

**float finalOpac;**

This is the final opacity value used for combining components together in **Shader::CombineComponents()**.

**Color finalC;**

This is the final output color that the **Shader::CombineComponents()** composites together.

**Color finalT;**

This is the final Shader transparency color output.

**Methods:**

public:

**Prototype:**

```
void ClearOutputs();
```

**Remarks:**

This method is called by the Standard material prior to calling the **Illum()** method of the Shader. It sets to black all the output colors:

```
ambIllumOut=diffIllumOut=transIllumOut=selfIllumOut=  
specIllumOut=reflIllumOut=Color(0.0f, 0.0f, 0.0f);
```

## Class InterfaceServer

See Also: [Class BaseInterface](#), [Class IObject](#), [Function Publishing System](#).

class InterfaceServer

### Description:

This class is available in release 4.0 and later only.

Class InterfaceServer is the base class for interface servers in 3ds max and should be inherited by any class that wishes to implement the GetInterface() protocol. The InterfaceServer also adds a data member for storing interfaces, typically extension interfaces added to maintain API binary compatibility.

### Methods:

public:

### Prototype:

```
virtual BaseInterface* GetInterface(Interface_ID id) = 0;
```

### Remarks:

Returns a pointer to the interface whose ID is specified.

### Parameters:

**Interface\_ID id**

The ID of the interface to return.

### Default Implementation:

```
{ return NULL; }
```

## Class IIRenderMgr

See Also: [Class InterfaceServer](#), [Class IInteractiveRender](#) ,  
[IIRenderMgrSelector](#) , [Class ViewExp](#)

```
class IIRenderMgr : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This class represents the abstract (interface) for an interactive rendering manager.

### Methods:

public:

### Prototype:

```
virtual bool CanExecute() = 0;
```

### Remarks:

This method will indicate the viewport has valid data and can execute.

### Prototype:

```
virtual void SetActive(bool active) = 0;
```

### Remarks:

This method allows you to activate and deactivate the current interactive rendering manager.

### Parameters:

**bool active**

TRUE to enable; FALSE to disable.

### Prototype:

```
virtual TCHAR* GetName() = 0;
```

### Remarks:

This method will return the name of the render manager.

### Prototype:

**virtual bool IsActive() = 0;**

**Remarks:**

This method returns TRUE if the current interactive rendering manager is active, otherwise FALSE.

**Prototype:**

**virtual BOOL AreAnyNodesSelected() const = 0;**

**Remarks:**

This method returns TRUE if the rendering manager has any selected notes or FALSE if there are none.

**Prototype:**

**virtual IIRenderMgrSelector\* GetNodeSelector() = 0;**

**Remarks:**

This method allows you to get the interface that determines whether nodes are selected.

**Prototype:**

**virtual HWND GetHwnd() const = 0;**

**Remarks:**

This method returns a handle to the current window which is being rendered to.

**Prototype:**

**virtual ViewExp \*GetViewExp() = 0;**

**Remarks:**

This method returns a pointer to the ViewExp associated with the current interactive rendering manager.

**Prototype:**

**virtual void SetPos(int X, int Y, int W, int H) = 0;**

**Remarks:**

This method allows you to set the position and size of the window being rendered to.

**Parameters:**

**int X, int Y**

The x and y screen coordinates of the window.

**int W, int H**

The width and height of the window.

**Prototype:**

**virtual void Show() = 0;**

**Remarks:**

This method will show the window currently being rendered to.

**Prototype:**

**virtual void Hide() = 0;**

**Remarks:**

This method will hide the window currently being rendered to.

**Prototype:**

**virtual void UpdateDisplay() = 0;**

**Remarks:**

This method will issue an update of the current display.

**Prototype:**

**virtual void Render() = 0;**

**Remarks:**

This method starts the actual rendering process.

**Prototype:**

**virtual void SetDelayTime(int msecDelay) = 0;**

**Remarks:**

This method allows you to set the delay time in milliseconds.

**Prototype:**

**virtual int GetDelayTime() = 0;**

**Remarks:**

This method returns the delay time in milliseconds.

**Prototype:**

**virtual void Close() = 0;**

**Remarks:**

This method will close the window currently being rendered to.

**Prototype:**

**virtual void Delete() = 0;**

**Remarks:**

This method will delete this render manager.

**Prototype:**

**virtual void SetCommandMode(CommandMode commandMode)  
= 0;**

**Remarks:**

This method allows you to set the command mode.

**Parameters:**

**CommandMode commandMode**

One of the following; **CMD\_MODE\_DRAW\_REGION**, or  
**CMD\_MODE\_SELECT\_OBJECT**.

**Prototype:**

**virtual CommandMode GetCommandMode() const = 0;**

**Remarks:**

This method returns the command mode, which is one of the following;

**CMD\_MODE\_DRAW\_REGION**, or  
**CMD\_MODE\_SELECT\_OBJECT**.

**Prototype:**

**virtual void SetActOnlyOnMouseUp(bool actOnlyOnMouseUp) = 0;**

**Remarks:**

This method allows you to define whether an interactive rendering action and update should be issued when the mouse button is released upward.

**Parameters:**

**bool actOnlyOnMouseUp**

TRUE to act only on mouse-up, otherwise FALSE.

**Prototype:**

**virtual bool GetActOnlyOnMouseUp() const = 0;**

**Remarks:**

This method returns TRUE if the interactive rendering action and update should be issued when the mouse button is released upward, otherwise FALSE.

**Prototype:**

**virtual void ToggleToolbar() const = 0;**

**Remarks:**

This method toggles the toolbar display mode (for docked windows).

**Prototype:**

**virtual IImageViewer::DisplayStyle GetDisplayStyle() const = 0;**

**Remarks:**

This method returns the display style of the window being rendered in, which is one of the following; **IV\_FLOATING** or **IV\_DOCKED**.

**Prototype:**

**virtual BOOL IsRendering() = 0;**

**Remarks:**

This method returns TRUE if the renderer is currently rendering, otherwise FALSE.

**Prototype:**

**static IIRenderMgr\* GetActiveIIRenderMgr();**

**Remarks:**

This method returns a pointer to the active interactive rendering manager, or NULL if none exist.

**Prototype:**

**static unsigned int GetNumIIRenderMgrs();**

**Remarks:**

This method returns the number of interactive rendering managers.

**Prototype:**

**static IIRenderMgr\* GetIIRenderMgr(unsigned int i);**

**Remarks:**

This method returns a pointer to the I-th interactive rendering manager.

**Parameters:**

**unsigned int i**

The index of the IIRenderMgr to return.

## Class IRenderProgressCallback

See Also: [Class IInteractiveRender](#), [Class RendProgressCallback](#) , [Class Color](#)

class IRenderProgressCallback : public RendProgressCallback

### Description:

This class is available in release 4.0 and later only.

This class represent a callback object which can be used with an interactive renderer.

Information set via **RendProgressCallback::SetCurField()** or **RendProgressCallback::SetSceneStats()** will be ignored. if a title is set via the inherited method **SetTitle()**, it will appear in the main status bar, but will be replaced by the 'IRenderTitle' when necessary.

### Methods:

public:

### Prototype:

```
virtual void SetProgressLineOrientation(LineOrientation orientation) = 0;
```

### Remarks:

This method allows you to set the orientation of the progress line shown during rendering.

### Parameters:

**LineOrientation orientation**

The line orientation which is one of the following; **LO\_Horizontal** or **LO\_Vertical**.

### Prototype:

```
virtual LineOrientation GetProgressLineOrientation() const = 0;
```

### Remarks:

This method allows you to retrieve the orientation of the progress line shown during rendering.

### Return Value:

The line orientation which is one of the following; **LO\_Horizontal** or **LO\_Vertical**.

**Prototype:**

**virtual void SetProgressLineColor(const Color& color) = 0;**

**Remarks:**

This method allows you to set the color of the progress line shown during rendering.

**Parameters:**

**const Color& color**

The color to set the progress line to.

**Prototype:**

**virtual const Color& GetProgressLineColor() const = 0;**

**Remarks:**

This method returns the color of the progress line shown during rendering.

**Prototype:**

**virtual void SetIRenderTitle(const TCHAR \*pProgressTitle) = 0;**

**Remarks:**

This method allows you to set the current title. This will appear in the main status bar as "'Title': xx% complete". If no title is provided, 'ActiveShade' will be used instead.

**Parameters:**

**const TCHAR \*pProgressTitle**

The progress title string.

**Prototype:**

**virtual const TCHAR \*GetIRenderTitle() const = 0;**

**Remarks:**

This method returns the current title.

## Class FPStaticInterface

See Also: [Class FPInterfaceDesc](#).

```
class FPStaticInterface: public FPInterfaceDesc
```

### Description

This class is currently the same as **FPInterfaceDesc** as per the following typedef:

```
typedef FPInterfaceDesc FPStaticInterface
```

See [Class FPInterfaceDesc](#) for details.

## Class BaseInterface

See Also: [Class InterfaceServer](#), [Class InterfaceNotifyCallback](#), [Class FPInterface](#), [Function Publishing System](#).

```
class BaseInterface : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This class represents the base class for interfaces in 3ds max. The BaseInterface class should be used as the base class for any new interface class in 3ds max and provides basic identity, memory management, and cloning methods. **Class FPInterface**, which is part of the Function Publishing system, has the BaseInterface class as its base class.

### Methods:

public:

### Prototype:

```
virtual Interface_ID GetID() = 0;
```

### Remarks:

This method returns the unique interface ID.

### Prototype:

```
virtual LifetimeType LifetimeControl()
```

### Remarks:

This method allows enquiries into the actual lifetime policy of a client and provide a server-controlled delete notify callback.

### Return Value:

One of the following LifetimeTypes:

#### **noRelease**

Do not call release, use interface as long as you like.

#### **immediateRelease**

The interface is only good for one calls. The release is implied so a call to release is not required.

### **wantsRelease**

The clients are controlling the lifetime, so the interface needs a Release() when the client has finished. This is the default.

### **serverControlled**

The server controls the lifetime and will use the InterfaceNotifyCallback to inform the code when it is gone.

#### **Default Implementation:**

```
{ return noRelease; }
```

#### **Prototype:**

```
virtual void
```

```
RegisterNotifyCallback(InterfaceNotifyCallback<BaseInterface>*  
incb);
```

#### **Remarks:**

This method allows you to register an interface notify callback.

#### **Parameters:**

```
InterfaceNotifyCallback<BaseInterface>* incb
```

A pointer to the interface notify callback.

#### **Default Implementation:**

```
{ }
```

#### **Prototype:**

```
virtual void
```

```
UnRegisterNotifyCallback(InterfaceNotifyCallback<BaseInterface>  
incb);
```

#### **Remarks:**

This method allows you to un-register an interface notify callback.

#### **Parameters:**

```
InterfaceNotifyCallback<BaseInterface>* incb
```

A pointer to the interface notify callback.

#### **Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual bool Acquire(InterfaceNotifyCallback* incb=NULL);
```

**Remarks:**

This method is part of the interface reference management and can be implemented by dynamically allocated interfaces for ref-count based lifetime control). This method should return TRUE if it needs **Release()** to be called.

**Parameters:**

**InterfaceNotifyCallback\* incb**

A pointer to an InterfaceNotifyCallback class which will be used to signal deletion (usually a mixin).

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

```
virtual void Release(InterfaceNotifyCallback* incb=NULL);
```

**Remarks:**

This method is called when a reference to this object is deleted.

**Parameters:**

**InterfaceNotifyCallback\* incb**

A pointer to an InterfaceNotifyCallback class which will be used to signal deletion (usually a mixin).

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void DeleteInterface();
```

**Remarks:**

This method can be used as a direct interface delete request.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BaseInterface* CloneInterface(void* remapDir = NULL);
```

**Remarks:**

This method allows you to clone the base interface.

**Parameters:**

**void\* remapDir**

The RemapDir passed to the clone method.

**Default Implementation:**

```
{ return NULL; }
```

## Class IMaxBitmapViewer

See Also: [Class IImageViewer](#), [Class Bitmap](#), [Class CropCallback](#)

```
class IMaxBitmapViewer : public IImageViewer
```

### Description:

This class is available in release 4.0 and later only.

This class represents an abstract interface class for a default bitmap viewer.

The following functions are available for use but are not part of the class IMaxBitmapViewer.

Note that the minimum size of the floating window is 390 x 325.

### Function:

```
IMaxBitmapViewer* CreateIMaxBitmapViewer(Bitmap*  
pBitmap, IImageViewer::DisplayStyle displayStyle);
```

### Remarks:

This function allows you to create a new bitmap viewer as either a floating window or docked in a viewport.

### Parameters:

**Bitmap\* pBitmap**

The bitmap to use with the viewer. This should not be NULL.

**IImageViewer::DisplayStyle displayStyle**

The display style for the viewer, which is either **IV\_FLOATING** or **IV\_DOCKED**.

### Return Value:

A pointer to a new IMaxBitmapViewer or NULL if the viewer could not be created.

### Function:

```
void ReleaseIMaxBitmapViewer(IMaxBitmapViewer *);
```

### Remarks:

This method will delete and release the specified bitmap viewer. This method should not be used while the viewer is being displayed. Use **UnDisplay()** or

**Hide()** before calling this method.

**Parameters:**

**IMaxBitmapViewer \***

A pointer to the viewer.

**Methods:**

public:

**Prototype:**

**virtual void SetBitmap(Bitmap\* pBitmap) = 0;**

**Remarks:**

This method allows you to set the bitmap which should be displayed in the bitmap viewer.

Note: This is for internal use only.

**Parameters:**

**Bitmap\* pBitmap**

A pointer to the bitmap to display.

**Prototype:**

**virtual Bitmap\* GetBitmap() const = 0;**

**Remarks:**

This method returns a pointer to the bitmap that is being displayed by the bitmap viewer.

**Prototype:**

**virtual void SetCropCB(CropCallback\* pCropCallback) = 0;**

**Remarks:**

This method allows you to set the crop callback function which will assist in interactive adjustments of the bitmap cropping rectangle. This method should not be called after the window has been displayed. Preferably the callback should be set before the window is displayed or after the window has been destroyed.

**Parameters:**

## **CropCallback\* pCropCallback**

A pointer to the callback function to set.

### **Prototype:**

```
virtual CropCallback* GetCropCB() const = 0;
```

### **Remarks:**

This method returns a pointer to the crop callback function used by the bitmap viewer.

### **Prototype:**

```
virtual void SetAutonomous(bool isAutonomous) = 0;
```

### **Remarks:**

This method allows you to set whether the bitmap viewer is autonomous or not. When the bitmap viewer is set in autonomous mode, the viewer will not have a clone button, although the viewer will not display a clone button unless the save button is also displayed. Changes will not take effect until the next time the viewer is displayed using **Display()**.

### **Parameters:**

**bool isAutonomous**

TRUE to set the viewer to autonomous, otherwise FALSE.

### **Prototype:**

```
virtual bool GetAutonomous() const = 0;
```

### **Remarks:**

This method returns TRUE if the viewer is autonomous, otherwise FALSE. When the bitmap viewer is set in autonomous mode, the viewer will not have a clone button, although the viewer will not display a clone button unless the save button is also displayed.

### **Prototype:**

```
virtual void SetCurrentPosition(WindowPosition currentPosition)  
= 0;
```

### **Remarks:**

A call to this method will never impact the current opened window. This function should be used prior to calling **Display()** in order to specify the position of the next created window.

**Prototype:**

**virtual WindowPosition GetCurrentPosition() const = 0;**

**Remarks:**

This method returns the current position of the bitmap viewer.

**Prototype:**

**virtual void SetShowSaveButton(bool showSaveButton) = 0;**

**Remarks:**

This method allows you to set whether or not the save button should be shown as part of the bitmap viewer window. A change will not take effect until the next time the viewer is opened via **Display()**.

**Parameters:**

**bool showSaveButton**

TRUE to show; FALSE to hide.

**Prototype:**

**virtual bool GetShowSaveButton() const = 0;**

**Remarks:**

This method returns TRUE if the save button is shown as part of the bitmap viewer window, otherwise FALSE.

**Prototype:**

**virtual bool Display(TCHAR \*title, WindowPosition position = WPos\_Center) = 0;**

**Remarks:**

This method will display the bitmap image viewer. Note that the **Show()** and **Hide()** methods have no effect until the window is initialized by a call to **Display()**. This method should not be called when the window is already

open and this method can't be used to create a docked viewer.

You should also not call this twice on this interface. The second time the function is called, the viewer will lose all references to the previously opened window. To switch images using the same bitmap viewer, a call to "UnDisplay" should be made to close the previous window before calling this method again.

**Parameters:**

**TCHAR \*title**

The title of the window to set.

**WindowPosition position = WPos\_Center**

The window position you wish to set, which is one of the following enum values; **WPos\_NULL**, **WPos\_UpperLeft**, **WPos\_LowerLeft**, **WPos\_UpperRight**, **WPos\_LowerRight**, **WPos\_Center**.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

```
virtual bool Display(TCHAR *title, HWND hParent, int x, int y,  
int w, int h) = 0;
```

**Remarks:**

This method will display the bitmap image viewer. Note that the **Show()** and **Hide()** methods have no effect until the window is initialized by a call to **Display()**. For docked viewers, the **hParent** parameter is the window into which the viewer will be docked. For floating viewers, the parameter has no effect.

You should also not call this twice on this interface. The second time the function is called, the viewer will lose all references to the previously opened window. To switch images using the same bitmap viewer, a call to "UnDisplay" should be made to close the previous window before calling this method again.

**Parameters:**

**TCHAR \*title**

The title of the window to set.

**HWND hParent**

A handle to the parent window.

**int x, y, h, w**

The position and dimensions of the window.

Note that these parameters will be ignored unless 'Current Position' is

**WPos\_NULL**.

**Return Value:**

TRUE if successful, otherwise FALSE.

**Prototype:**

**virtual bool UnDisplay() = 0;**

**Remarks:**

This method will undisplay the current bitmap image. The **Show()** and **Hide()** methods will no longer function after the window is destroyed.

**Prototype:**

**virtual void ClearScreen() = 0;**

**Remarks:**

This method will clear the bitmap image viewer window contents. This will not delete the window as it would using the Delete button on the viewer toolbar which actually deletes the contents of the bitmap.

**Prototype:**

**virtual POINT XFormScreenToBitmap(const POINT &pt) const = 0;**

**Remarks:**

This method will transform a specified point between the window's client coordinates and the bitmap coordinates and return the result.

**Parameters:**

**const POINT &pt**

The point on screen.

**Prototype:**

**virtual POINT XFormBitmapToScreen(const POINT &pt) const = 0;**

**Remarks:**

This method will transform a specified point between the window's client coordinates and the bitmap coordinates and return the result.

**Parameters:**

**const POINT &pt**

The point on the bitmap.

**Prototype:**

**virtual Rect XFormScreenToBitmap(const Rect &rect) const = 0;**

**Remarks:**

This method will transform a specified rectangle between the window's client coordinates and the bitmap coordinates and return the result.

**Parameters:**

**const Rect &rect**

The screen rectangle.

**Prototype:**

**virtual Rect XFormBitmapToScreen(const Rect &rect) const = 0;**

**Remarks:**

This method will transform a specified rectangle between the window's client coordinates and the bitmap coordinates and return the result.

**Parameters:**

**const Rect &rect**

The bitmap rectangle.

**Prototype:**

**virtual void ShowToolbar(bool show) = 0;**

**Remarks:**

This method allows you to set whether the toolbar should be shown as part of

the bitmap image viewer window.

**Parameters:**

**bool show**

TRUE to show; FALSE to hide.

**Prototype:**

**virtual void ToggleToolbar() = 0;**

**Remarks:**

This method allows you to toggle the toolbar of the bitmap image viewer window on and off.

**Prototype:**

**virtual void GetDrawableRect(Rect& drawableRect) = 0;**

**Remarks:**

This method will retrieve the portion of the window's client area that is safe to draw in (in client coordinates and below the toolbar). This method can return a value larger than the displayed bitmap if the window size is large enough.

**Parameters:**

**Rect& drawableRect**

The client area.

**Prototype:**

**virtual void RefreshWindow(Rect\* pRefreshRegion = NULL) = 0;**

**Remarks:**

This method will refresh the region of the window, or the entire window if region is NULL

**Parameters:**

**Rect\* pRefreshRegion = NULL**

The region of the window you wish to refresh.

## Class IMenu

See Also: [Class IMenuElement](#), [Class IMenuItem](#), [Class IMenuGlobalContext](#), [Class IMenuLocalContext](#), [Structure MenuEvent](#), [Class IPoint2](#), [Class Interface](#).

```
class IMenu : public IMenuElement
```

### Description:

This class is available in release 4.0 and later only.

This abstract class represents an interface for a menu item. Methods that are marked as internal should not be used.

### Methods:

public:

### Prototype:

```
virtual void SetIMenuGlobalContext(IMenuGlobalContext*  
pIMenuGlobalContext) = 0;
```

### Remarks:

This method is used internally.

This method sets a new context for the menu, invalidating the menu's cache.

### Parameters:

**IMenuGlobalContext\* pIMenuGlobalContext**

Points to the context to set.

### Prototype:

```
virtual IMenuLocalContext* GetIMenuLocalContext() = 0;
```

### Remarks:

This method is used internally.

Returns a pointer to the menu's local context.

### Prototype:

```
virtual int NumItems() const = 0;
```

### Remarks:

Returns the number of items in the menu.

**Prototype:**

**virtual IMenuItem\* GetItem(int position) = 0;**

**Remarks:**

Returns a pointer to the specified menu item.

**Parameters:**

**int position**

The position/index in the menu.

**Prototype:**

**virtual void AddItem(IMenuItem\* item, int position = -1) = 0;**

**Remarks:**

This method adds the specified item into the menu at the position passed. The default position is at the end.

**Parameters:**

**IMenuItem\* item**

Points to the menu item to add.

**int position = -1**

Position 0 indicates the beginning of the list. A negative or otherwise invalid position defaults to end of list.

**Prototype:**

**virtual void RemoveItem(int position) = 0;**

**Remarks:**

Removes the item from the menu whose position is passed.

**Parameters:**

**int position**

The zero based index in the list of the item to remove. Position 0 is the first item.

**Prototype:**

**virtual void RemoveItem(IMenuItem\* item) = 0;**

**Remarks:**

Removes the specified item from the menu (if it indeed appears in the menu).

**Parameters:**

**IMenuItem\* item**

Points to the menu item to remove.

**Prototype:**

**virtual IPoint2 GetMaxItemSize() = 0;**

**Remarks:**

This method is used internally.

Returns the maximum size of all items in the menu in pixels.

**Prototype:**

**virtual void Initialize() = 0;**

**Remarks:**

This method is used internally.

This method is called before menu is first displayed during a user / menu interaction.

**Prototype:**

**virtual void PostMenuInteraction() = 0;**

**Remarks:**

This method is used internally.

This method is called after a user / menu interaction.

**Prototype:**

**virtual bool HandleEvent(MenuEvent event) = 0;**

**Remarks:**

This method is used internally.

This method is called to handle an event occurring within the menu.

**Parameters:**

**MenuEvent event**

A menu event structure containing the event data.

**Return Value:**

TRUE if the event was handled successfully, otherwise FALSE.

**Prototype:**

```
virtual void Show(DisplayMethod displayMethod =  
DM_NORMAL, Box2 *rect = NULL) = 0;
```

**Remarks:**

This method is used internally.

This method will display the menu using the provided display method.

**Prototype:**

```
virtual void Hide(DisplayMethod displayMethod =  
DM_NORMAL) = 0;
```

**Remarks:**

This method is used internally.

This method will hide the menu using the provided display method.

**Prototype:**

```
virtual IMenuItem* FindAccelItem(TCHAR accelerator) = 0;
```

**Remarks:**

This method is used internally.

Finds and returns a pointer to the menu item whose accelerator is passed.

**Parameters:**

**TCHAR accelerator**

The single character of the accelerator.

**Return Value:**

A pointer to the menu item or NULL if not found.

**Prototype:**

```
virtual IMenuItem* FindNewSelectedItem() = 0;
```

**Remarks:**

This method is used internally.

Returns a pointer to the currently selected menu item.

**Prototype:**

**virtual void Display(IMenu\* pParentMenu = NULL) = 0;**

**Remarks:**

This method is used internally.

This method displays the menu.

**Parameters:**

**IMenu\* pParentMenu = NULL**

Points to the parent menu.

**Prototype:**

**virtual void DisplayItems(IPoint2& origin, bool descending) = 0;**

**Remarks:**

This method is used internally.

**Prototype:**

**virtual void Undisplay() = 0;**

**Remarks:**

This method is used internally.

This method removes the menu from the display.

**Prototype:**

**virtual bool IsDisplayingSubMenu() = 0;**

**Remarks:**

This method is used internally.

Returns TRUE if the menu is displaying a sub-menu; otherwise FALSE.

**Prototype:**

**virtual void TimerElapsed(EventParam timingType) = 0;**

**Remarks:**

This method is used internally.

This method notifies the menu that the timer has elapsed.

**Prototype:**

```
virtual void SetShowTitle(bool showTitle) = 0;
```

**Remarks:**

This method is used internally.

Sets whether to show the title or not.

**Parameters:**

**bool showTitle**

Pass true to show the title; false to not show it.

**Prototype:**

```
virtual bool GetShowTitle() const = 0;
```

**Remarks:**

This method is used internally.

Returns true if a title is shown; otherwise false.

**Prototype:**

```
virtual void SetCustomTitle(const TCHAR *customTitle) = 0;
```

**Remarks:**

Sets the custom title to the string passed.

**Parameters:**

**const TCHAR \*customTitle**

Points to the string to use.

**Prototype:**

```
virtual const TSTR& GetCustomTitle() const = 0;
```

**Remarks:**

Returns the custom title string.

**Prototype:**

**virtual void SetUseCustomTitle(bool useCustomTitle) = 0;**

**Remarks:**

Sets if the item should use a custom title.

**Parameters:**

**bool useCustomTitle**

Pass true to use a custom title; false to not use one.

**Prototype:**

**virtual bool GetUseCustomTitle() const = 0;**

**Remarks:**

This method returns TRUE if the menu uses a custom title, otherwise FALSE.

**Prototype:**

**virtual void SetUseGlobalWidths(bool useGlobalWidths) = 0;**

**Remarks:**

This method is used internally.

This method allows you to instruct the menu to use global widths in order to show the title.

**Parameters:**

**bool useGlobalWidths**

TRUE to use global widths, otherwise FALSE.

**Prototype:**

**virtual bool GetUseGlobalWidths() const = 0;**

**Remarks:**

This method is used internally.

This method returns TRUE if the menu uses global widths in order to show the title, otherwise FALSE.

**Prototype:**

**virtual bool NoVisibleItems() = 0;**

**Remarks:**

This method is used internally.

Returns true if the menu has no visible items in it; otherwise false.

**The following functions are not part of the class but are available for use.**

**Function:**

**IMenu \* GetIMenu();**

**Remarks:**

This method will return a pointer to the IMenu.

**Prototype:**

**void ReleaseIMenu(IMenu \*);**

**Remarks:**

This method will release the specified IMenu.

**Parameters:**

**IMenuItem \***

A pointer to the IMenu you wish to release.

## Class IMenuBarContext

See Also: [Class IMenuContext](#), [Class Interface](#).

class IMenuBarContext: public IMenuContext

### Description:

This class is available in release 4.0 and later only.

### Description:

This class is available in release 4.0 and later only.

This abstract class represents an interface for a menu bar context. Methods that are marked as internal should not be used.

### Methods:

public:

### Prototype:

**virtual void SetMenu(IMenu\* pMenu) = 0;**

### Remarks:

This method allows you to set the menu associated with this context.

### Parameters:

**IMenu\* pMenu**

A pointer to the menu.

### Prototype:

**virtual IMenu\* GetMenu() = 0;**

### Remarks:

This method returns a pointer to the menu associated with this context.

### Prototype:

**virtual HMENU CreateWindowsMenu() = 0;**

### Remarks:

Used Internally.

This method will create a new windows menu and return it's handle.

**Prototype:**

**virtual void UpdateWindowsMenu() = 0;**

**Remarks:**

Used Internally.

This method will update the current windows menu.

**Prototype:**

**virtual HMENU GetCurWindowsMenu() = 0;**

**Remarks:**

Used Internally.

This method returns the handle to the current windows menu.

**Prototype:**

**virtual void ExecuteAction(int cmdId) = 0;**

**Remarks:**

This method executes an action based on the provided command ID.

**Parameters:**

**int cmdId**

The command ID of the action to execute.

**Prototype:**

**virtual bool CommandIDInRange(int cmdId) = 0;**

**Remarks:**

Used internally.

**Parameters:**

**int cmdId**

The command ID.

## Class IQuadMenuContext

See Also: [Class IMenuContext](#), [Class Interface](#), [List of Right-Click Contexts](#).

```
class IQuadMenuContext: public IMenuContext
```

### Description:

This class is available in release 4.0 and later only.

This abstract class represents the interface for a quad menu bar context and provides the functionality to manage the quad menu context by adding and removing menu sections.

Note that methods marked for internal use only should not be used.

### Methods:

public:

### Prototype:

```
virtual bool AddQuadMenu(IQuadMenu* pMenu, TCHAR* pName) = 0;
```

### Remarks:

This method allows you to add a new quad menu to the context.

### Parameters:

**IQuadMenu\* pMenu**

A pointer to the quad menu you wish to add.

**TCHAR\* pName**

The name of the quad menu.

### Return Value:

TRUE if the quad menu is successfully added, otherwise FALSE.

### Prototype:

```
virtual void SetMenu(int index, IQuadMenu* pMenu, TCHAR* pName) = 0;
```

### Remarks:

This method allows you to set a quad menu for a slot in the context.

**Parameters:**

**int index**

The index of the slot in the context.

**IQuadMenu\* pMenu**

A pointer to the quad menu you wish to set.

**TCHAR\* pName**

The name of the quad menu.

**Prototype:**

**virtual void RemoveMenu(int index) = 0;**

**Remarks:**

This method allows you to remove a quad menu from the context.

**Parameters:**

**int index**

The index of the quad menu to remove.

**Prototype:**

**virtual int MenuCount() = 0;**

**Remarks:**

This method returns the number of quad menu's in this context.

**Prototype:**

**virtual IQuadMenu\* GetMenu(int index) = 0;**

**Remarks:**

This method returns a pointer to a quad menu based on its index in the context.

**Parameters:**

**int index**

The index of the quad menu you wish to retrieve.

**Prototype:**

**virtual int GetCurrentMenuIndex() = 0;**

**Remarks:**

This method returns the index of the currently set default right-click menu.

**Prototype:**

```
virtual void SetCurrentMenuIndex(int index) = 0;
```

**Remarks:**

This method allows you to set the current default right-click menu.

**Parameters:**

**int index**

The index of the menu you wish to set as the default menu.

**Prototype:**

```
virtual bool GetShowAllQuads(int index) = 0;
```

**Remarks:**

This method returns the state of the "Show All Menus" flag in each quad registered in the menu manager. TRUE if the flag is set or FALSE if the flag is not set.

**Parameters:**

**int index**

The index of the quad menu.

**Prototype:**

```
virtual void SetShowAllQuads(int index, bool showAll) = 0;
```

**Remarks:**

This method allows you to set the "Show All Menus" flag in each quad registered in the menu manager.

**Parameters:**

**int index**

The index of the quad menu.

**bool showAll**

Set this parameter to TRUE if you wish to enable the "Show All Menus" flag. Otherwise FALSE.

**Prototype:**

**virtual RightClickContext GetRightClickContext() = 0;**

**Remarks:**

This method queries the state of the modifier keys and returns the appropriate context.

**Return Value:**

See the [List of Right-Click Contexts..](#)

**Prototype:**

**virtual IQuadMenu\* GetRightClickMenu(RightClickContext context) = 0;**

**Remarks:**

This method returns a pointer to the quad menu which has been assigned to the specified right-click context, or NULL if no menu is assigned.

**Parameters:**

**RightClickContext context**

See the [List of Right-Click Contexts..](#)

**Prototype:**

**virtual void SetRightClickMenu(RightClickContext context, IQuadMenu \*pMenu) = 0;**

**Remarks:**

This method allows you to set the quad menu associated with a specific right-click context.

**Parameters:**

**RightClickContext context**

See the [List of Right-Click Contexts..](#)

**IQuadMenu \*pMenu**

A pointer to the quad menu you wish to set.

**Prototype:**

**virtual int FindMenu(IQuadMenu\* pMenu) = 0;**

**Remarks:**

This method returns the index for a specified quad menu, or -1 if the menu is not in the context.

**Parameters:****IQuadMenu\* pMenu**

A pointer to the quad menu you wish to obtain the index of.

**Prototype:****virtual IQuadMenu\* FindMenuByTitle(TCHAR\* pTitle) = 0;****Remarks:**

This method returns a pointer to a quad menu by specifying the title of the menu you wish to find. NULL will be returned if the menu was not found.

**Parameters:****TCHAR\* pTitle**

The title string of the menu.

## List of Right-Click Contexts

See Also: [Class IQuadMenuContext](#), [Class IMenuManager](#)

This is the list of the right-click contexts:

### **kNonePressed**

No key is pressed.

### **kShiftPressed**

The shift key is pressed.

### **kAltPressed**

The alt key is pressed.

### **kControlPressed**

The control key is pressed.

### **kShiftAndAltPressed**

The shift and alt keys are pressed.

### **kShiftAndControlPressed**

The shift and control keys are pressed.

### **kControlAndAltPressed**

The control and alt keys are pressed.

### **kShiftAndAltAndControlPressed**

The shift, alt, and control keys are pressed.

## List of Standard Color IDs

See Also: [Class IColorManager](#).

This is the list of the standard color IDs that 3ds max registers:

**kBackground** -- Used for all windows backgrounds.  
**kText** --Used for static and button text  
**kActiveCommand** --Used for active command mode buttons  
**kHilight** -- COLOR\_BTNHILIGHT  
**kShadow** -- COLOR\_BTNSHADOW  
**kWindow** -- COLOR\_WINDOW  
**kActiveCaption** -- COLOR\_ACTIVECAPTION  
**kToolTipBackground** -- COLOR\_INFOBK  
**kToolTipText** -- COLOR\_INFOTEXT  
**kHilightText** -- COLOR\_HILIGHTTEXT  
**kWindowText** -- COLOR\_WINDOWTEXT  
**kItemHilight** -- COLOR\_HILIGHT  
**kSubObjectColor** -- This is the blue sub-object color  
**k3dDarkShadow** -- COLOR\_3DDKSHADOW  
**k3dLight** -- COLOR\_3DLIGHT  
**kAppWorkspace** -- COLOR\_APPWORKSPACE  
**kTrackbarBg**  
**kTrackbarBgSel**  
**kTrackbarText**  
**kTrackbarTicks**  
**kTrackbarKeys**  
**kTrackbarSelKeys**  
**kTrackbarCursor**  
**kPressedButton**  
**kTimeSliderBg**  
**kViewportBorder**

**kActiveViewportBorder**

**kRollupTitleFace**

**kRollupTitleText**

**kRollupTitleHiligh**

**kRollupTitleShadow**

**kSelectionRubberBand**

**kStackViewSelection** -- This is the yellow sub-object color.

## Class DragAndDropHandler

See Also: [Class IDragAndDropMgr](#), [Class DropType](#)

```
class DragAndDropHandler : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

The **DragAndDropHandler** class is the base class from which specialized drag-and-drop handlers should be derived. Instances of these classes can be registered with the **DragAndDropMgr** when enabling a window for DnD activity and virtual methods on them are called to handle the various DnD events. Use **IDragAndDropMgr::EnableDnD(hwnd, flag, handler)** to enable DnD in a window and specify the **DragAndDropHandler** instance that will handle DnD events for that window. Note that the event method calls all deliver the current target window as an explicit argument, so one instance can potentially be shared among many windows. Further, a parent window can be enabled for DnD and this effectively causes all child windows to be enabled and handled by the given handler (unless overridden by an explicit **EnableDnD** with a different handler on a child).

```
// enable DnD on my window
```

```
GetDragAndDropMgr()->EnabledDnD(hWnd, TRUE,  
&myHandler);
```

There are actually two sets of DnD event handler virtual methods in the base class, a low-level set that equates to the methods on the OLE **IDropTarget** interface that take raw **IDataObject** drop data and a high-level set that take fully-parsed **DropType** drop data. The low-level methods have default implementations provided that do this parsing and call the corresponding high-level method, so in most cases you only need to provide implementations for the high-level methods. You would provide implementations of the low-level methods if custom parsing of the **IDataObject** was required.

### Data Members:

```
protected:
```

```
DropType* current_droptype;
```

Cache for the currently parsed DropType. This is usually filled in during **DragEnter()** processing in the **DragAndDropHandler** for the current window.

**static IDragAndDropMgr\* dndMgr;**

Cached pointer to the DnD manager. For use by subclasses.

#### Methods:

public:

#### Prototype:

```
virtual HRESULT DragEnter(HWND window, IDataObject*  
pDataObject, DWORD grfKeyState, POINTL pt, DWORD*  
pdwEffect);
```

#### Remarks:

Override this method in your **DragAndDropHandler** subclass to get low-level control over DnD operations. This is just a redirect of the identical method called on the OLE **IDropTarget** interface, see MSDN docs for details.

The default implementation for this methods use the **DropClipFormat** and **DropType** classes to recognize and parse the incoming **IDataObject** into a **DropType** instance and hand this to the associated high-level DnD handler methods described next. As an example, here is the default **DragEnter()** implementation which does the initial parsing on entry to a window:

```
// DragAndDropHandler default methods, parse IDataObject into  
// DropTypes & hand-off main DnD methods to specific handlers  
HRESULT DragAndDropHandler::DragEnter(HWND hWnd,  
IDataObject* pDataObject, DWORD grfKeyState, POINTL pt,  
DWORD* pdwEffect)  
{  
    current_droptype = NULL;  
    // look for one of our accepted clip formats  
    DropClipFormat* cf =
```

```

DropClipFormat::FindClipFormat(pDataObject);
if (cf != NULL)
{
    // have one, get it to parse it into a DropType subclass
    current_droptype = cf->ParseDataObject(pDataObject);
    if (current_droptype != NULL)
    {
        // got recognizable drop data,
            // pass on to high-level method
        if(pdwEffect)
            *pdwEffect = DROPEFFECT_LINK|DROPEFFECT_COPY;
        POINT p = { pt.x, pt.y };
        DragEnter(hWnd, current_droptype,
                grfKeyState, p, pdwEffect);
        return S_OK;
    }
}
// nothing for us
if(pdwEffect)
    *pdwEffect = DROPEFFECT_NONE;
return S_OK;
}

```

**Parameters:**

**HWND window**

The specified handle to the window in which the DnD event is occurring. This is one of the windows that was enabled via a

**IDragAndDropMgr::EnabledDnD()** call, so it may be the parent of the lowest-level window that the mouse is actually over.

**IDataObject\* pDataObject**

The incoming IDataObject.

**DWORD grfKeyState**

The specified current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags **MK\_CONTROL**, **MK\_SHIFT**, **MK\_ALT**, **MK\_BUTTON**, **MK\_LBUTTON**, **MK\_MBUTTON**, and **MK\_RBUTTON**.

**POINTL pt**

The specified current cursor coordinates in the coordinate space of the drop-target window.

**DWORD\* pdwEffect**

On entry, pointer to the value of the **pdwEffect** parameter of the **DoDragDrop** function. On return, must contain one of the effect flags from the Win32 **DROPEFFECT** enumeration, which indicates what the result of the drop operation would be.

**Return Value:**

Standard return values of **E\_OUTOFMEMORY**, **E\_INVALIDARG**, **F\_UNEXPECTED**, and **E\_FAIL**, **S\_OK**.

**Prototype:**

```
virtual HRESULT Drop(HWND window, IDataObject*  
pDataObject, DWORD grfKeyState, POINTL pt, DWORD*  
pdwEffect);
```

**Remarks:**

This method will parse the dropped dataObject.

**Parameters:**

**HWND window**

The specified handle to the window in which the DnD event is occurring. This is one of the windows that was enabled via a **IDragAndDropMgr::EnabledDnD()** call, so it may be the parent of the lowest-level window that the mouse is actually over.

**IDataObject\* pDataObject**

The incoming IDataObject.

**DWORD grfKeyState**

The specified current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags **MK\_CONTROL**,

**MK\_SHIFT, MK\_ALT, MK\_BUTTON, MK\_LBUTTON, MK\_MBUTTON, and MK\_RBUTTON.**

**POINTL pt**

The specified current cursor coordinates in the coordinate space of the drop-target window.

**DWORD\* pdwEffect**

On entry, pointer to the value of the **pdwEffect** parameter of the **DoDragDrop** function. On return, must contain one of the effect flags from the Win32 **DROPEFFECT** enumeration, which indicates what the result of the drop operation would be.

**Return Value:**

Standard return values of **E\_OUTOFMEMORY, E\_INVALIDARG, F\_UNEXPECTED, and E\_FAIL, S\_OK.**

**Prototype:**

**virtual HRESULT DragOver(HWND window, DWORD grfKeyState, POINTL pt, DWORD \* pdwEffect);**

**Remarks:**

This method handles the process of dragging over a drop target.

**Parameters:**

**HWND window**

The specified handle to the window in which the DnD event is occurring. This is one of the windows that was enabled via a

**IDragAndDropMgr::EnabledDnD()** call, so it may be the parent of the lowest-level window that the mouse is actually over.

**DWORD grfKeyState**

The specified current state of the keyboard modifier keys on the keyboard.

Valid values can be a combination of any of the flags **MK\_CONTROL, MK\_SHIFT, MK\_ALT, MK\_BUTTON, MK\_LBUTTON, MK\_MBUTTON, and MK\_RBUTTON.**

**POINTL pt**

The specified current cursor coordinates in the coordinate space of the drop-target window.

## **DWORD\* pdwEffect**

On entry, pointer to the value of the **pdwEffect** parameter of the **DoDragDrop** function. On return, must contain one of the effect flags from the Win32 **DROPEFFECT** enumeration, which indicates what the result of the drop operation would be.

### **Return Value:**

Standard return values of **E\_OUTOFMEMORY**, **E\_INVALIDARG**, **F\_UNEXPECTED**, and **E\_FAIL**, **S\_OK**.

### **Prototype:**

```
virtual HRESULT DragEnter(HWND window, DropType* type,  
DWORD grfKeyState, POINT& pt, DWORD* pdwEffect);
```

### **Remarks:**

This is the high-level method called to handle DnD events with already recognized and parsed data object. Override the above methods as needed in your **DragAndDropHandler** subclass to handle DnD events.

### **Parameters:**

#### **HWND window**

The specified handle to the window in which the DnD event is occurring. This is one of the windows that was enabled via a **IDragAndDropMgr::EnabledDnD()** call, so it may be the parent of the lowest-level window that the mouse is actually over.

#### **DropType\* type**

The specified Pointer to the **DropType** instance that corresponds to the data in the dropped **IDataObject**. You can use **the DropType::TypeCode()** method to determine the droptype (see the built-in codes in the **DropType** section). Each **DropType** subclass instance has utility methods and public data members containing the parsed drop data. See each subclass definition for details.

#### **DWORD grfKeyState**

The specified current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags **MK\_CONTROL**, **MK\_SHIFT**, **MK\_ALT**, **MK\_BUTTON**, **MK\_LBUTTON**,

**MK\_MBUTTON**, and **MK\_RBUTTON**.

**POINT& pt**

The specified current cursor coordinates in the coordinate space of the drop-target window.

**DWORD\* pdwEffect**

On entry, pointer to the value of the **pdwEffect** parameter of the **DoDragDrop** function. On return, must contain one of the effect flags from the Win32 **DROPEFFECT** enumeration, which indicates what the result of the drop operation would be.

**Return Value:**

Standard return values of **E\_OUTOFMEMORY**, **E\_INVALIDARG**, **F\_UNEXPECTED**, and **E\_FAIL**, **S\_OK**.

**Default Implementation:**

```
{ return E_FAIL; }
```

**Prototype:**

```
virtual HRESULT Drop(HWND window, DropType* type,  
    DWORD grfKeyState, POINT& pt, DWORD* pdwEffect);
```

**Remarks:**

This method will parse the dropped dataObject.

Here's an example implementation of Drop() in the default handler:

```
HRESULT DefaultDragAndDropHandler::Drop(HWND hwnd,  
    DropType* type,  
    DWORD grfKeyState, POINT& pt, DWORD* pdwEffect)  
{  
    // This could take a while, set wait cursor  
    HCURSOR hOldCursor = SetCursor(LoadCursor(NULL,  
        IDC_WAIT));  
    HRESULT result = S_OK;  
  
    // load the dropped data if needed
```

```
if (type->Load())
{
    // see if dropped on a viewport, if so adjust point
    // to be vp-relative
    HWND vpwin = FindDropViewport(hwnd, pt);

    // Handle the drop depending on drop type
    BOOL bRet;
    switch (type->TypeCode())
    {
        case SCENEFILE_DROPTYPE:
            bRet = HandleDroppedGeom(
                hwnd, vpwin, pt,
                sceneFileDropType.current_package[0]);

            break;
        case IMAGEFILE_DROPTYPE:
            bRet = HandleDroppedBitmap(
                hwnd, vpwin, pt,
                imageFileDropType.current_package[0]);

            break;
        case DROPSCRIPTFILE_DROPTYPE:
            bRet = HandleDroppedDropScript(
                hwnd, vpwin, pt,
                dropScriptFileDropType.current_package[0]);

            break;
    }
    result = bRet ? S_OK : E_FAIL;
}
// restore cursor
SetCursor(hOldCursor);
```

```
    return result;
}
```

#### Parameters:

##### **HWND window**

The specified handle to the window in which the DnD event is occurring. This is one of the windows that was enabled via a **IDragAndDropMgr::EnabledDnD()** call, so it may be the parent of the lowest-level window that the mouse is actually over.

##### **DropType\* type**

The specified Pointer to the **DropType** instance that corresponds to the data in the dropped **IDataObject**. You can use **the DropType::TypeCode()** method to determine the droptype (see the built-in codes in the **DropType** section). Each **DropType** subclass instance has utility methods and public data members containing the parsed drop data. See each subclass definition for details.

##### **DWORD grfKeyState**

The specified current state of the keyboard modifier keys on the keyboard. Valid values can be a combination of any of the flags **MK\_CONTROL**, **MK\_SHIFT**, **MK\_ALT**, **MK\_BUTTON**, **MK\_LBUTTON**, **MK\_MBUTTON**, and **MK\_RBUTTON**.

##### **POINT& pt**

The specified current cursor coordinates in the coordinate space of the drop-target window.

##### **DWORD\* pdwEffect**

On entry, pointer to the value of the **pdwEffect** parameter of the **DoDragDrop** function. On return, must contain one of the effect flags from the Win32 **DROPEFFECT** enumeration, which indicates what the result of the drop operation would be.

#### Return Value:

Standard return values of **E\_OUTOFMEMORY**, **E\_INVALIDARG**, **F\_UNEXPECTED**, and **E\_FAIL**, **S\_OK**.

#### Default Implementation:

```
{ return E_FAIL; }
```

**Prototype:**

```
virtual HRESULT DragOver(HWND window, DWORD  
grfKeyState, POINT& pt, DWORD * pdwEffect);
```

**Remarks:**

This method handles the process of dragging over a drop target.

**Parameters:****HWND window**

The specified handle to the window in which the DnD event is occurring. This is one of the windows that was enabled via a

**IDragAndDropMgr::EnabledDnD()** call, so it may be the parent of the lowest-level window that the mouse is actually over.

**DWORD grfKeyState**

The specified current state of the keyboard modifier keys on the keyboard.

Valid values can be a combination of any of the flags **MK\_CONTROL**, **MK\_SHIFT**, **MK\_ALT**, **MK\_BUTTON**, **MK\_LBUTTON**, **MK\_MBUTTON**, and **MK\_RBUTTON**.

**POINT& pt**

The specified current cursor coordinates in the coordinate space of the drop-target window.

**DWORD\* pdwEffect**

On entry, pointer to the value of the **pdwEffect** parameter of the

**DoDragDrop** function. On return, must contain one of the effect flags from the Win32 **DROPEFFECT** enumeration, which indicates what the result of the drop operation would be.

**Return Value:**

Standard return values of **E\_OUTOFMEMORY**, **E\_INVALIDARG**, **F\_UNEXPECTED**, and **E\_FAIL**, **S\_OK**.

**Default Implementation:**

```
{ return E_FAIL; }
```

**Prototype:**

```
virtual HRESULT DragLeave(HWND window);
```

**Remarks:**

This method handles the process of the drag operation leaving the drop target window.

**Parameters:****HWND window**

The specified handle to the window in which the DnD event is occurring. This is one of the windows that was enabled via a

**IDragAndDropMgr::EnabledDnD()** call, so it may be the parent of the lowest-level window that the mouse is actually over.

**Return Value:**

Standard return values of **E\_OUTOFMEMORY**, **E\_INVALIDARG**, **F\_UNEXPECTED**, and **E\_FAIL**, **S\_OK**.

**Default Implementation:**

```
{ return E_FAIL; }
```

**Prototype:**

```
virtual void Acquire();
```

**Remarks:**

This method is called called when the DnD manager starts managing DnD events for a particular window for this handler. You can provide an implementation if you need to keep track of extant uses of the handler (say, by ref-counting) or to do handler-specific cleanup.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void Release();
```

**Remarks:**

This method is called called when the DnD manager stops managing DnD events for a particular window for this handler. You can provide an implementation if you need to keep track of extant uses of the handler (say, by ref-counting) or to do handler-specific cleanup.

**Default Implementation:**

**{ }**

## Class URLTab

See Also: [Class DragAndDropHandler](#)

```
class URLTab : public Tab<TCHAR*>
```

### Description:

This class is available in release 4.0 and later only.

The URLTab class is a **Tab<TCHAR\*>** utility class that is used by certain components in the Drag and Drop manager to hold and pass around packages of file URLs. The class manages its own local copies of URL strings. This class represents the additional API support by URLTab, over-and-above that provided by any **Tab<>** template instantiation.

### Data Members:

protected:

#### **BOOL downloaded;**

This flag is set to indicate the URL package has been downloaded and names will reflect local copies.

### Methods:

public:

### Prototype:

```
URLTab();
```

### Remarks:

Constructor.

### Default Implementation:

```
{ downloaded = FALSE; }
```

### Prototype:

```
~URLTab();
```

### Remarks:

Destructor.

URLTabs manage their own local string element copies, the destructor frees all these strings.

**Default Implementation:**

```
{ Clear(); }
```

**Prototype:**

```
URLTab& operator=(const Tab<TCHAR*>& tb);
```

**Remarks:**

Assignment operator.

**Prototype:**

```
URLTab& operator=(const URLTab& tb);
```

**Remarks:**

Assignment operator.

**Prototype:**

```
void Add(TCHAR* url);
```

**Remarks:**

This method adds a URL string to the package. A local copy of the string will be made.

**Parameters:**

**TCHAR\* url**

The URL string to add.

**Prototype:**

```
void Change(int i, TCHAR* url);
```

**Remarks:**

This method replaces the i'th element by deletes the old string, taking a local copy of the new one. This is used by the various loaders to replace a URL with its local copy path name upon download.

**Parameters:**

**int i**

The index of the URL to replace.

**TCHAR\* url**

The new URL string.

**Prototype:**

**void Clear();**

**Remarks:**

This method clears the package (deletes all the strings), zeros the Tab<> and resets 'downloaded' to FALSE.

## Class INodeDisplayControl

See Also: [Class InterfaceServer](#), [Class NodeDisplayCallback](#), [Class INode](#).

```
class INodeDisplayControl : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This class is an interface that is used to register the node display callback. To get a pointer to this interface the developer should use the following macro:

```
#define GetNodeDisplayControl(i) ((INodeDisplayControl*)i->GetInterface(I_NODEDISPLAYCONTROL))
```

### Methods:

public:

### Prototype:

```
virtual void RegisterNodeDisplayCallback(NodeDisplayCallback *cb)=0;
```

### Remarks:

Register a node display callback which can be used to control the display of nodes in the scene.

### Parameters:

**NodeDisplayCallback \*cb**

Points to the callback object.

### Prototype:

```
virtual void
```

```
UnRegisterNodeDisplayCallback(NodeDisplayCallback *cb)=0;
```

### Remarks:

Un-registers the node display callback.

### Parameters:

**NodeDisplayCallback \*cb**

Points to the callback object.

**Prototype:**

**virtual bool SetNodeCallback(NodeDisplayCallback\* hook)=0;**

**Remarks:**

Sets the current current callback. The callback must be previously registered.

**Parameters:**

**NodeDisplayCallback\* hook**

Points to the node display callback to set.

**Return Value:**

Returns true if the callback was set; otherwise false.

**Prototype:**

**virtual NodeDisplayCallback\* GetNodeCallback()=0;**

**Remarks:**

Returns a pointer to the current current callback. The callback must be previously registered.

**Prototype:**

**virtual void InvalidateNodeDisplay()=0;**

**Remarks:**

Viewport refresh routine. Tthis function only invalidates the display, it's up to the callback to select the correct redraw technique.

# Class XTCContainer

See Also: [Class XTCTObject](#)

class XTCContainer

## Description:

This class is available in release 4.0 and later only.

This class represents a container class for XTCTObjects.

## Data Members:

public:

**XTCTObject \*obj;**

A pointer to the XTCTObject.

**int prio;**

The priority.

**int branchID;**

The branch identifier.

## Methods:

public:

## Prototype:

**XTCContainer();**

## Remarks:

Constructor.

## Default Implementation:

**{obj = NULL; prio = 0; branchID = -1;}**

## Class IXTCAccess

See Also: [Class Object](#), [Class XTCTObject](#)

class IXTCAccess

### Description:

This class is available in release 4.0 and later only.

This class provides an interface to access Extension Channels.

### Methods:

public:

### Prototype:

```
virtual Interface_ID GetID();
```

### Remarks:

This method returns the IXTCAccess interface ID.

### Default Implementation:

```
{ return IXTCACCESS_INTERFACE_ID; }
```

### Prototype:

```
virtual LifetimeType LifetimeControl();
```

### Remarks:

This method allows enquiries into the actual lifetime policy of a client and provide a server-controlled delete notify callback.

### Return Value:

One of the following LifetimeTypes:

#### **noRelease**

Do not call release, use interface as long as you like.

#### **immediateRelease**

The interface is only good for one calls. The release is implied so a call to release is not required.

#### **wantsRelease**

The clients are controlling the lifetime, so the interface needs a Release() when the client has finished. This is the default.

## **serverControlled**

The server controls the lifetime and will use the InterfaceNotifyCallback to inform the code when it is gone.

### **Default Implementation:**

```
{ return noRelease; }
```

### **Prototype:**

```
virtual void AddXTCObject(XTCObject *pObj, int priority = 0,  
int branchID = -1)=0;
```

### **Remarks:**

This method adds an extension object into the pipeline.

### **Parameters:**

**XTCObject \*pObj**

The extension object you wish to add.

**int priority = 0**

The priority to set.

**int branchID = -1**

The branch identifier to set.

### **Prototype:**

```
virtual int NumXTCObjects()=0;
```

### **Remarks:**

This method returns the number of extension objects.

### **Prototype:**

```
virtual XTCObject *GetXTCObject(int index)=0;
```

### **Remarks:**

This method returns the I-th extension object.

### **Parameters:**

**int index**

The index of the extension object to return.

**Prototype:**

**virtual void RemoveXTCObject(int index)=0;**

**Remarks:**

This method allows you to remove the I-th extension object.

**Parameters:**

**int index**

The index of the extension object you wish to remove.

**Prototype:**

**virtual void SetXTCObjectPriority(int index,int priority)=0;**

**Remarks:**

This method allows you to set the priority for the I-th extension object.

**Parameters:**

**int index**

The index of the extension object for which to set the priority.

**int priority**

The priority to set.

**Prototype:**

**virtual int GetXTCObjectPriority(int index)=0;**

**Remarks:**

This method returns the priority for the I-th extension object.

**Parameters:**

**int index**

The index of the extension object.

**Prototype:**

**virtual void SetXTCObjectBranchID(int index,int branchID)=0;**

**Remarks:**

This method allows you to set the branch identifier for the I-th extension object.

**Parameters:**

**int index**

The index of the extension object.

**int branchID**

The branch identifier to set.

**Prototype:**

**virtual int GetXTCObjectBranchID(int index)=0;**

**Remarks:**

This method returns the branch identifier for the I-th extension object.

**Parameters:**

**int index**

The index of the extension object.

**Prototype:**

**virtual void MergeAdditionalChannels(Object \*from, int branchID)=0;**

**Remarks:**

This method has to be called whenever the CompoundObject updates a branch (calling Eval on it). Object \*from is the object returned **from Eval (os.obj)**; branchID is an int, that specifies that branch. The extension channel will get a callback to **RemoveXTCObjectOnMergeBranches()** and **MergeXTCObject()**. By default it returns true to **RemoveXTCObjectOnMergeBranches()**, which means, that the existing XTCObjects with that branchID will be deleted. The method **MergeXTCObject()** simply copies the XTCObjects from the incoming branch into the compound object.

**Parameters:**

**Object \*from**

The object from which to merge additional channels

**int branchID**

The branch identifier.

**Prototype:**

```
virtual void BranchDeleted(int branchID, bool  
reorderChannels)=0;
```

**Remarks:**

This method has to be called on the CompoundObject, so it can delete the XTCTObjects for the specific branch. The XTCTObject will have again the final decision if the XTCTObject gets really deleted or not in a callback to **RemoveXTCTObjectOnBranchDeleted()**, which will return true, if the XTCTObject should be removed.

**Parameters:**

**int branchID**

The branch identifier.

**bool reorderChannels**

TRUE to reorder the channels, otherwise FALSE.

**Prototype:**

```
virtual void CopyAdditionalChannels(Object *from, bool  
deleteOld = true, bool bShallowCopy = false)=0;
```

**Remarks:**

This method copies all extension objects from the "from" objects into the current object. In case deleteOld is false, the objects will be appended. In case it is true, the old XTCTObjects will be deleted.

**Parameters:**

**Object \*from**

The object to copy from.

**bool deleteOld = true**

TRUE to delete the old channel, FALSE to append the channels.

**bool bShallowCopy = false**

TRUE to create a shallow copy, FALSE to create a deep copy.

**Prototype:**

```
virtual void DeleteAllAdditionalChannels()=0;
```

**Remarks:**

This method allows you to delete all additional channels.

## Class IDataChannel

See Also: [Class InterfaceServer](#), [Class Class\\_ID](#), [Class IFaceDataChannel](#), [Class IFaceDataChannelsEnumCallBack](#)

```
class IDataChannel : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

A data channel is a homogeneous collection of objects of a user defined type (data objects). Data channels are uniquely identified by a Class\_ID. Data channels can be associated with any element type of a 3ds max object: faces or vertexes of Meshes, etc. You can use the macro

**GetDataChannelInterface(obj)** to obtain a pointer to this interface.

### Methods:

public:

### Prototype:

```
virtual Class_ID DataChannelID() const =0;
```

### Remarks:

This method returns the unique class ID of the channel.

### Prototype:

```
virtual ULONG Count() const;
```

### Remarks:

This method returns the number of data objects in this channel.

### Default Implementation:

```
{ return 0; }
```

### Prototype:

```
virtual void DeleteThis() = 0;
```

### Remarks:

Destructor. Deletes self.

## Class IFaceDataChannelsEnumCallBack

See Also: [Class IDataChannel](#), [Class IFaceDataChannel](#) , [Class IFaceDataMgr](#)

class IFaceDataChannelsEnumCallBack

### Description:

This class is available in release 4.0 and later only.

This is an interface class that will allow a callback procedure to execute for all face data channels of an object. You should derive your own classes from this interface and overwrite the **Proc()** method to call the desired **IFaceDataChannel** method. It is up to the derived class to interpret the context parameter passed to **Proc()**.

Classes that hold face data channels can implement the method:

**EnumFaceDataChannels(IFaceDataEnumCallBack& cb, void\* pContext)**

This method would be called with a reference to an instance of a class derived from **IFaceDataEnumCallBack** in which **Proc()** is overwritten. The implementation of **EnumFaceDataChannels** would call **cb.Proc** for each of the face-data channels of the object

Note: Do not delete data channels from within the Proc(). This could lead to unexpected behaviour.

### Methods:

public:

### Prototype:

```
virtual BOOL Proc(IFaceDataChannel* pChan, void* pContext) = 0;
```

### Remarks:

The callback method that should be overridden.

### Parameters:

**IFaceDataChannel\* pChan**

A pointer to the face-data channel interface.

**void\* pContext**

A pointer to the context data.

**Return Value:**

TRUE if successful, otherwise FALSE.

## Class CollisionOps

See Also: [Class ICollision](#), [Class CollisionPlane](#), [Class CollisionSphere](#), [Class CollisionVNormal](#), [Class CollisionMesh](#), [Class Box3](#), [Class Point3](#), [Class FPInterface](#)

```
class CollisionOps : public FPInterface
```

### Description:

This class is available in release 4.0 and later only.

This class represents the operation interface to the collision detection system.

The interface ID is defined as **COLLISION\_FO\_INTERFACE**. To obtain a pointer to this interface you can use the macro

**GetCollisionOpsInterface(cd)**, which will return **(CollisionOps \*)(cd)->GetFPInterface(COLLISION\_FO\_INTERFACE)**.

### Methods:

public:

### Prototype:

```
virtual int SupportedCollisions(ReferenceTarget *r) = 0;
```

### Remarks:

This method returns the collision type supported by the engine.

### Parameters:

**ReferenceTarget \*r**

A pointer to the reference target to check the collision type for.

### Return Value:

One of the following;

**POINT\_COLLISION** for point collision, currently supported.

**SPHERE\_COLLISION** for spherical collision, currently not supported.

**BOX\_COLLISION** for box collision, currently not supported.

**EDGE\_COLLISION** for edge collision, currently not supported.

### Prototype:

**virtual void PreFrame(ReferenceTarget \*r, TimeValue &t,  
TimeValue &dt) = 0;**

**Remarks:**

This method will be called once before the checkcollision is called for each frame which allows you to do any required initialization.

**Parameters:**

**ReferenceTarget \*r**

A pointer to the reference target.

**TimeValue t**

The time at which to initialize.

**TimeValue dt**

The delta of time the particle wil travel.

**Prototype:**

**virtual void PostFrame(ReferenceTarget \*r, TimeValue &t,  
TimeValue &dt) = 0;**

**Remarks:**

This method will be called at the end of each frame solve to allow you to destroy and deallocate any data you no longer need.

**Parameters:**

**ReferenceTarget \*r**

A pointer to the reference target.

**TimeValue t**

The time at which to initialize.

**TimeValue dt**

The delta of time the particle wil travel.

**Prototype:**

**virtual BOOL CheckCollision (ReferenceTarget \*r,TimeValue t,  
Point3 pos, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3  
&norm, Point3 &friction, Point3 &inheritedVel) = 0;**

**Remarks:**

This method will be called to execute a point to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**ReferenceTarget \*r**

A pointer to the reference target.

**TimeValue t**

The end time of the particle.

**Point3 pos**

The position of the particle in world space.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

```
virtual BOOL CheckCollision (ReferenceTarget *r,TimeValue t,Point3 pos, float radius, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;;
```

**Remarks:**

This method will be called to execute a sphere to surface collision and compute the time at which the particle hit the surface.

**Parameters:****ReferenceTarget \*r**

A pointer to the reference target.

**TimeValue t**

The end time of the particle.

**Point3 pos**

The position of the particle in world space.

**float radius**

The radius of the sphere.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

**virtual BOOL CheckCollision (ReferenceTarget \*r,TimeValue t,**

**Box3 box, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;**

**Remarks:**

This method will be called to execute a box to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**ReferenceTarget \*r**

A pointer to the reference target.

**TimeValue t**

The end time of the particle.

**Box3 box**

The box itself.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

```
virtual BOOL CheckCollision (ReferenceTarget *r,TimeValue  
t,Point3 edgeA,Point3 edgeB ,Point3 vel, float dt, float &at, Point3  
&hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel)  
= 0;
```

**Remarks:**

This method will be called to execute an edge to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**ReferenceTarget \*r**

A pointer to the reference target.

**TimeValue t**

The end time of the particle.

**Point3 edgeA**

The first edge.

**Point3 edgeB**

The second edge.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

## Class CollisionPlane

See Also: [Class ICollision](#), [Class CollisionOps](#), [Class CollisionSphere](#), [Class CollisionVNormal](#), [Class CollisionMesh](#), [Class Box3](#), [Class Point3](#), [Class IParamBlock2](#), [Class INode](#), [Class Control](#)

```
class CollisionPlane : public ICollision
```

### Description:

This class is available in release 4.0 and later only.

This class represents the planar collision object with the ClassID defined as **PLANAR\_COLLISION\_ID**. This class allows you to define a plane in space and determine if a particle hit it.

### Data Members:

private:

**INode \*node;**

The associated node.

public:

**IParamBlock2 \*pblock;**

The parameter block data. You can use the following enum parameter ID's:

**collisionplane\_width**

**collisionplane\_height**

**collisionplane\_quality**

**collisionplane\_node**

**Interval validity;**

The validity interval.

**Matrix3 tm;**

The plane's TM.

**Matrix3 invtm;**

The inverse TM.

**Matrix3 prevInvTm;**

The cached previous inverse TM.

**int initialTime;**

The initial time.

**Tab<Matrix3> invTmList;**

The table of inverse TM's.

**float width, height;**

The width and height of the plane.

**int quality;**

The collision quality value.

**Methods:**

public:

**Prototype:**

**CollisionPlane();**

**Remarks:**

Constructor.

**Prototype:**

**~CollisionPlane();**

**Remarks:**

Destructor.

**Prototype:**

**int SupportedCollisions();**

**Remarks:**

This method determines the type of collisions that are supported.

**Return Value:**

One of the following;

**POINT\_COLLISION** for point collision, currently supported.

**SPHERE\_COLLISION** for spherical collision, currently not supported.

**BOX\_COLLISION** for box collision, currently not supported.

**EDGE\_COLLISION** for edge collision, currently not supported.

**Default Implementation:**

```
{ return POINT_COLLISION; }
```

**Prototype:**

```
void PreFrame(TimeValue t, TimeValue dt);
```

**Remarks:**

This method will be called once before the checkcollision is called for each frame which allows you to do any required initialization.

**Parameters:**

**TimeValue t**

The time at which to initialize.

**TimeValue dt**

The delta of time the particle will travel.

**Prototype:**

```
void PostFrame(TimeValue t, TimeValue dt);
```

**Remarks:**

This method will be called at the end of each frame solve to allow you to destroy and deallocate any data you no longer need.

**Parameters:**

**TimeValue t**

The time at which to initialize.

**TimeValue dt**

The delta of time the particle will travel.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual BOOL CheckCollision (TimeValue t, Point3 pos, Point3 vel,  
float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3  
&friction, Point3 &inheritedVel) = 0;
```

**Remarks:**

This method will be called to execute a point to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**TimeValue t**

The end time of the particle.

**Point3 pos**

The position of the particle in world space.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

```
virtual BOOL CheckCollision (TimeValue t,Point3 pos, float  
radius, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3  
&norm, Point3 &friction, Point3 &inheritedVel) = 0;;
```

**Remarks:**

This method will be called to execute a sphere to surface collision and compute the time at which the particle hit the surface.

**Parameters:****TimeValue t**

The end time of the particle.

**Point3 pos**

The position of the particle in world space.

**float radius**

The radius of the sphere.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

```
virtual BOOL CheckCollision (TimeValue t, Box3 box, Point3 vel,  
float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3  
&friction, Point3 &inheritedVel) = 0;
```

**Remarks:**

This method will be called to execute a box to surface collision and compute the time at which the particle hit the surface.

**Parameters:****TimeValue t**

The end time of the particle.

**Box3 box**

The box itself.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

```
virtual BOOL CheckCollision (TimeValue t,Point3 edgeA,Point3 edgeB ,Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;
```

**Remarks:**

This method will be called to execute an edge to surface collision and compute the time at which the particle hit the surface.

**Parameters:****TimeValue t**

The end time of the particle.

**Point3 edgeA**

The first edge.

**Point3 edgeB**

The second edge.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

```
void SetWidth(TimeValue t, float w);
```

**Remarks:**

Sets the width of the plane.

**Parameters:****TimeValue t**

The time at which to set the width.

**float w**

The width.

**Default Implementation:**

```
{ pblock->SetValue(collisionplane_width,t,w); }
```

**Prototype:**

```
void SetHeight(TimeValue t, float h);
```

**Remarks:**

Sets the height of the plane.

**Parameters:**

**TimeValue t**

The time at which to set the height.

**float h**

The height.

**Default Implementation:**

```
{ pblock->SetValue(collisionplane_height,t,h); }
```

**Prototype:**

```
void SetQuality(TimeValue t, int q);
```

**Remarks:**

Sets the quality of the solve. This is the maximum number of iterations the solver will take to find the hit point. The lower quality the more likely a particle will leak through the surface but the faster the solver will be.

**Parameters:**

**TimeValue t**

The time at which to set the quality.

**int q**

The quality value.

**Default Implementation:**

```
{ pblock->SetValue(collisionplane_quality,t,q); }
```

**Prototype:**

```
void SetNode(TimeValue t, INode *n);
```

**Remarks:**

Sets the node which drives the TM to put the plane in world space.

**Parameters:**

**TimeValue t**

The time at which to set the node.

**Inode \*n**

The node to set.

**Default Implementation:**

```
{ pblock->SetValue(collisionplane_node,t,n); node = n; }
```

**Prototype:**

```
void SetWidth(Control *c);
```

**Remarks:**

Sets the controller for the plane width.

**Parameters:****Control \*c**

A pointer to the controller to set.

**Default Implementation:**

```
{ pblock->SetController(collisionplane_width,0,c); }
```

**Prototype:**

```
void SetHeight(Control *c);
```

**Remarks:**

Sets the controller for the plane height.

**Parameters:****Control \*c**

A pointer to the controller to set.

**Default Implementation:**

```
{ pblock->SetController(collisionplane_height,0,c); }
```

**Prototype:**

```
void SetQuality(Control *c);
```

**Remarks:**

Sets the quality of the solve. This is the maximum number of iterations the

solver will take to find the hit point. The lower quality the mire likely a particle will leak through the surface but the faster the solver will be.

**Parameters:**

**Control \*c**

A pointer to the controller to set.

**Default Implementation:**

```
{ pblock->SetController(collisionplane_quality,0,c); }
```

**Prototype:**

```
void DeleteThis();
```

**Remarks:**

Self deletion.

**Default Implementation:**

```
{ delete this; }
```

**Prototype:**

```
Class_ID ClassID();
```

**Remarks:**

This method returns the class ID.

**Default Implementation:**

```
{return PLANAR_COLLISION_ID;}
```

**Prototype:**

```
SClass_ID SuperClassID();
```

**Remarks:**

This method returns the super class ID.

**Default Implementation:**

```
{return REF_MAKER_CLASS_ID;}
```

**Prototype:**

```
int NumRefs();
```

**Remarks:**

This method returns the number of references.

**Default Implementation:**

```
{ return 1; }
```

**Prototype:**

```
RefTargetHandle GetReference(int i);
```

**Remarks:**

This method returns the I-th parameter block.

**Default Implementation:**

```
{ return pblock; }
```

**Prototype:**

```
void SetReference(int i, RefTargetHandle rtarg);
```

**Remarks:**

This method allows you to set the I-th parameter block.

**Parameters:**

**int i**

The I-th parameter block to set.

**RefTargetHandle rtarg**

The reference target handle to the parameter block.

**Default Implementation:**

```
{pblock = (IParamBlock2*)rtarg;}
```

**Prototype:**

```
RefTargetHandle Clone(RemapDir &remap = NoRemap());
```

**Remarks:**

This method is called to have the plug-in clone itself. This method should copy both the data structure and all the data residing in the data structure of this reference target. The plug-in should clone all its references as well.

**Parameters:**

## **RemapDir &remap = NoRemap()**

This class is used for remapping references during a Clone. See [Class RemapDir](#).

### **Return Value:**

A pointer to the cloned item.

### **Prototype:**

**RefResult NotifyRefChanged(Interval changeInt,  
RefTargetHandle hTarget, PartID& partID,RefMessage message);**

### **Remarks:**

A plug-in which makes references must implement this method to receive and respond to messages broadcast by its dependents.

### **Parameters:**

#### **Interval changeInt**

This is the interval of time over which the message is active.

#### **RefTargetHandle hTarget**

This is the handle of the reference target the message was sent by. The reference maker uses this handle to know specifically which reference target sent the message.

#### **PartID& partID**

This contains information specific to the message passed in. Some messages don't use the partID at all. See the section [List of Reference Messages](#) for more information about the meaning of the partID for some common messages.

#### **RefMessage message**

The msg parameters passed into this method is the specific message which needs to be handled. See [List of Reference Messages](#).

### **Return Value:**

The return value from this method is of type RefResult. This is usually **REF\_SUCCEED** indicating the message was processed. Sometimes, the return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

## Class CollisionVNormal

See Also: [Class ICollision](#), [Class CollisionOps](#), [Class CollisionPlane](#), [Class CollisionSphere](#), [Class CollisionMesh](#), [Class Box3](#), [Class Point3](#)

class CollisionVNormal

### Description:

This class is available in release 4.0 and later only.

This class represents a general list of collision vertex normals.

### Data Members:

public:

**Point3 norm;**

The normal vector

**DWORD smooth;**

The smoothing flag.

**CollisionVNormal \*next;**

A pointer to the next normal in the linked list.

**BOOL init;**

The initialization flag.

### Methods:

public:

### Prototype:

**CollisionVNormal();**

### Remarks:

Constructor.

### Default Implementation:

```
{smooth=0;next=NULL;init=FALSE;norm=Point3(0,0,0);}
```

### Prototype:

**CollisionVNormal(Point3 &n,DWORD s);**

### Remarks:

Constructor.

**Parameters:**

**Point3 &n**

The vector to initialize with.

**DWORD s**

The smoothing flag to initialize with.

**Default Implementation:**

```
{next=NULL;init=TRUE;norm=n;smooth=s;}
```

**Prototype:**

```
~CollisionVNormal();
```

**Remarks:**

Destructor.

**Default Implementation:**

```
{delete next;}
```

**Prototype:**

```
void AddNormal(Point3 &n,DWORD s);
```

**Remarks:**

Add a vector to the list.

**Parameters:**

**Point3 &n**

The vector to add.

**DWORD s**

The smoothing flag to add.

**Prototype:**

```
Point3 &GetNormal(DWORD s);;
```

**Remarks:**

Returns the specified normal from the list.

**Parameters:**

**DWORD s**

The index of the normal in the list.

**Prototype:**

```
void Normalize();
```

**Remarks:**

This method normalizes the vector.

## Class CollisionMesh

See Also: [Class ICollision](#), [Class CollisionOps](#), [Class CollisionPlane](#), [Class CollisionSphere](#), [Class CollisionVNormal](#), [Class Box3](#), [Class Point3](#), [Class IParamBlock2](#), [Class INode](#), [Class Control](#)

```
class CollisionMesh : public ICollision
```

### Description:

This class is available in release 4.0 and later only.

This class represents the planar collision object with the ClassID defined as **MESH\_COLLISION\_ID**. This class allows you to define a plane in space and determine if a particle hit it.

### Data Members:

private:

**INode \*node;**

The associated node.

public:

**IParamBlock2 \*pblock;**

The parameter block data. You can use the following enum parameter ID's:

**collisionmesh\_hit\_face\_index**

**collisionmesh\_hit\_bary**

**collisionmesh\_node**

**Interval validity;**

The validity interval.

**Matrix3 tm;**

The plane's TM.

**Matrix3 invtm;**

The inverse TM.

**Matrix3 tmPrev;**

The previous TM.

**Matrix3 prevInvTm;**

The cached previous inverse TM.

**float radius;**

The radius of the sphere.

**Mesh \*dmesh;**

The mesh pointer.

**int nv, nf;**

The mesh number of vertices and number of faces.

**CollisionVNormal \*vnorms;**

The collision vertex normals.

**Point3 \*fnorms;**

The face normals.

**Methods:**

public:

**Prototype:**

**CollisionMesh();**

**Remarks:**

Constructor.

**Prototype:**

**~CollisionMesh();**

**Remarks:**

Destructor.

**Prototype:**

**int SupportedCollisions();**

**Remarks:**

This method determines the type of collisions that are supported.

**Return Value:**

One of the following;

**POINT\_COLLISION** for point collision, currently supported.

**SPHERE\_COLLISION** for spherical collision, currently not supported.

**BOX\_COLLISION** for box collision, currently not supported.

**EDGE\_COLLISION** for edge collision, currently not supported.

**Default Implementation:**

```
{ return POINT_COLLISION; }
```

**Prototype:**

```
void PreFrame(TimeValue t, TimeValue dt);
```

**Remarks:**

This method will be called once before the checkcollision is called for each frame which allows you to do any required initialization.

**Parameters:**

**TimeValue t**

The time at which to initialize.

**TimeValue dt**

The delta of time the particle wil travel.

**Prototype:**

```
void PostFrame(TimeValue t, TimeValue dt);
```

**Remarks:**

This method will be called at the end of each frame solve to allow you to destroy and deallocate any data you no longer need.

**Parameters:**

**TimeValue t**

The time at which to initialize.

**TimeValue dt**

The delta of time the particle wil travel.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual BOOL CheckCollision (TimeValue t, Point3 pos, Point3 vel,  
float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3  
&friction, Point3 &inheritedVel) = 0;
```

**Remarks:**

This method will be called to execute a point to surface collision and compute the time at which the particle hit the surface.

**Parameters:****TimeValue t**

The end time of the particle.

**Point3 pos**

The position of the particle in world space.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

**virtual BOOL CheckCollision (TimeValue t,Point3 pos, float radius, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;;**

**Remarks:**

This method will be called to execute a sphere to surface collision and

compute the time at which the particle hit the surface.

**Parameters:**

**TimeValue t**

The end time of the particle.

**Point3 pos**

The position of the particle in world space.

**float radius**

The radius of the sphere.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

**virtual BOOL CheckCollision (TimeValue t, Box3 box, Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;**

**Remarks:**

This method will be called to execute a box to surface collision and compute

the time at which the particle hit the surface.

**Parameters:**

**TimeValue t**

The end time of the particle.

**Box3 box**

The box itself.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

**virtual BOOL CheckCollision (TimeValue t,Point3 edgeA,Point3 edgeB ,Point3 vel, float dt, float &at, Point3 &hitPoint, Point3 &norm, Point3 &friction, Point3 &inheritedVel) = 0;**

**Remarks:**

This method will be called to execute an edge to surface collision and compute the time at which the particle hit the surface.

**Parameters:**

**TimeValue t**

The end time of the particle.

**Point3 edgeA**

The first edge.

**Point3 edgeB**

The second edge.

**Point3 vel**

The velocity of the particle in world space.

**float dt**

The delta of time that the particle travels (**t-dt** being the start of time of the particle)

**float &at**

The point in time that the collision occurs with respect to the **dt**.

**Point3 &hitPoint**

The point of collision.

**Point3 &norm**

The bounce vector component of the final velocity.

**Point3 &friction**

The friction vector component of the final velocity.

**Point3 inheritedVel**

The approximated amount of velocity inherited from the motion of the deflector.

**Return Value:**

TRUE if there's a collision, otherwise FALSE.

**Prototype:**

```
void SetNode(TimeValue t, INode *n);
```

**Remarks:**

Sets the node which drives the TM to put the plane in world space.

**Parameters:****TimeValue t**

The time at which to set the node.

**INode \*n**

The node to set.

**Default Implementation:**

```
{ pblock->SetValue(collisionmesh_node,t,n); node = n; }
```

**Prototype:**

```
void DeleteThis();
```

**Remarks:**

Self deletion.

**Default Implementation:**

```
{ delete this; }
```

**Prototype:**

```
Class_ID ClassID();
```

**Remarks:**

This method returns the class ID.

**Default Implementation:**

```
{return SPHERICAL_COLLISION_ID;}
```

**Prototype:**

```
SClass_ID SuperClassID();
```

**Remarks:**

This method returns the super class ID.

**Default Implementation:**

```
{return REF_MAKER_CLASS_ID;}
```

**Prototype:**

```
int NumRefs();
```

**Remarks:**

This method returns the number of references.

**Default Implementation:**

```
{ return 1; }
```

**Prototype:**

```
RefTargetHandle GetReference(int i);
```

**Remarks:**

This method returns the I-th parameter block.

**Default Implementation:**

```
{ return pblock; }
```

**Prototype:**

```
void SetReference(int i, RefTargetHandle rtarg);
```

**Remarks:**

This method allows you to set the I-th parameter block.

**Parameters:**

**int i**

The I-th parameter block to set.

**RefTargetHandle rtarg**

The reference target handle to the parameter block.

**Default Implementation:**

```
{pblock = (IParamBlock2*)rtarg;}
```

**Prototype:**

```
RefTargetHandle Clone(RemapDir &remap = NoRemap());
```

**Remarks:**

This method is called to have the plug-in clone itself. This method should copy both the data structure and all the data residing in the data structure of this reference target. The plug-in should clone all its references as well.

**Parameters:**

**RemapDir &remap = NoRemap()**

This class is used for remapping references during a Clone. See [Class RemapDir](#).

**Return Value:**

A pointer to the cloned item.

**Prototype:**

**RefResult NotifyRefChanged(Interval changeInt,  
RefTargetHandle hTarget, PartID& partID,RefMessage message);**

**Remarks:**

A plug-in which makes references must implement this method to receive and respond to messages broadcast by its dependents.

**Parameters:****Interval changeInt**

This is the interval of time over which the message is active.

**RefTargetHandle hTarget**

This is the handle of the reference target the message was sent by. The reference maker uses this handle to know specifically which reference target sent the message.

**PartID& partID**

This contains information specific to the message passed in. Some messages don't use the partID at all. See the section [List of Reference Messages](#) for more information about the meaning of the partID for some common messages.

**RefMessage message**

The msg parameters passed into this method is the specific message which needs to be handled. See [List of Reference Messages](#).

**Return Value:**

The return value from this method is of type RefResult. This is usually **REF\_SUCCEED** indicating the message was processed. Sometimes, the return value may be **REF\_STOP**. This return value is used to stop the message from being propagated to the dependents of the item.

## Class IVertexShader

See Also: [Class ID3DGraphicsWindow](#) , [Class Mesh](#), [Class INode](#), [Class MNMesh](#)

class IVertexShader

### Description:

This class is available in release 4.0 and later only.

The abstract interface for a vertex shader.

### Methods:

public:

### Prototype:

```
virtual HRESULT Initialize(Mesh *mesh, INode *node) = 0;
```

### Remarks:

This method loads VertexShader instructions, create any additional per vertex data on a per node basis. VertexShader instructions should be loaded once and shared among all the nodes using this VertexShader. Additional per vertex data should be (re)created only when the associated node data changes. In addition, if there is sufficient memory, VertexBuffers can be created and updated only when node data changes in order to improve rendering performance.

### Parameters:

**Mesh \*mesh**

A pointer to the mesh to initialize.

**INode \*node**

A pointer to the node to initialize.

### Prototype:

```
virtual HRESULT Initialize(MNMesh *mnmesh, INode *node) = 0;
```

### Remarks:

This method loads VertexShader instructions, create any additional per vertex data on a per node basis. VertexShader instructions should be loaded once and

shared among all the nodes using this VertexShader. Additional per vertex data should be (re)created only when the associated node data changes. In addition, if there is sufficient memory, VertexBuffers can be created and updated only when node data changes in order to improve rendering performance.

**Parameters:**

**MNMesh \*mnmesh**

A pointer to the MNMesh to initialize.

**INode \*node**

A pointer to the node to initialize.

## Class BaseInterfaceServer

See Also: [Class InterfaceServer](#), [Class BaseInterface](#)

```
class BaseInterfaceServer : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

The BaseInterface server class specializes the InterfaceServer class with an implementation based on a Tab<> of interface pointers for storing interfaces, typically extension interfaces, and providing an interface iteration protocol.

**class IObject** in the Function Publishing System specializes class BaseInterfaceServer. The class contains a protected table of BaseInterface pointers. **Class IObject** is an example of a class which is based on the BaseInterfaceServer class.

### Methods:

public:

### Prototype:

```
virtual BaseInterface* GetInterface(Interface_ID id);
```

### Remarks:

This method returns a pointer to the BaseInterface of the specified interface.

### Parameters:

**Interface\_ID id**

The interface ID for which to return the BaseInterface.

### Prototype:

```
virtual int NumInterfaces() const;
```

### Remarks:

This method returns the number of interfaces.

### Default Implementation:

```
{ return interfaces.Count(); }
```

### Prototype:

**virtual BaseInterface\* GetInterfaceAt(int i) const;**

**Remarks:**

This method returns a pointer to the BaseInterface of the I-th interface.

**Parameters:**

**int i**

The index of the interface in the table.

**Default Implementation:**

```
{ return interfaces[i]; }
```

**Prototype:**

```
~BaseInterfaceServer();
```

**Remarks:**

Destructor.

## Structure GBufData

See Also: [List of Image Channels](#), [Class GBufReader](#), [Class GBufWriter](#), [Class GBuffer](#), [Structure RealPixel](#), [Structure Color24](#), [Class Point2](#), [List of G-Buffer Channel Types](#).

This structure is used by the **GBufReader** and **GBurWriter** code to hold the G-Buffer data.

**struct GBufData {**

**float z;**

The floating point depth value at the center of the fragment that is foremost in the sorted list of a-buffer fragments.

**UBYTE mtl\_id;**

The ID assigned to the material via the Material Editor.

**UWORD node\_id;**

This is the ID assigned to node via the Object Properties / G-buffer ID spinner.

**Point2 uv;**

UV coordinates, stored as a Point2.

**DWORD normal;**

Normal vector in view space, compressed.

**RealPixel realpix;**

Non clamped colors in "RealPixel" format.

**UBYTE coverage;**

Pixel coverage of the front surface.

**UWORD rend\_id;**

The renderer will set this ID for all rendered nodes, and will set all non-rendered nodes to 0xffff.

**Color24 color;**

This is color returned by the material shader for the fragment.

**Color24 transp;**

This is transparency returned by the material shader for the fragment.

**Color24 weight;**

This is the sub-pixel weight of a fragment.

**Point2 veloc;**

This gives the velocity vector of the fragment relative to the screen, in screen

coordinates.

};

# Class RadiosityInterface

See Also: : [Class RadiosityEffect](#)

class RadiosityInterface : public FPStaticInterface

## Description:

This class is only available in release 5 or later.

This class provides access to the Advanced Lighting dialog. It allows you to open and close the dialog, and get or set the currently active Advanced Lighting plug-in. This class is a function-published static interface; you can use GetCOREInterface() to obtain an instance of the class, as follows:

```
RadiosityInterface r = static_cast<IRadiosityInterface*>  
(GetCOREInterface(RADIOSITY_INTERFACE))
```

This interface is also accessible via MAXScript as “SceneRadiosity”.

All methods of this class are implemented by the system.

## Methods:

### Prototype:

**virtual void OpenRadiosityPanel() = 0**

### Remarks:

Displays the Advanced Lighting dialog, unless it is already displayed.

### Prototype:

**virtual void CloseRadiosityPanel() = 0**

### Remarks:

Hides the Advanced Lighting dialog, if it is currently displayed.

### Prototype:

**virtual void MinimizeRadiosityPanel() = 0**

**Remarks:**

Minimizes the Advanced Lighting dialog if it is open.

**Prototype:**

**virtual RadiosityEffect\* GetRadiosity () = 0**

**Return value:**

Returns a pointer to the currently active Advanced Lighting plug-in (RadiosityEffect) if any.

**Prototype:**

**virtual void SetRadiosity (RadiosityEffect\* op) = 0**

**Remarks:**

Sets the given Advanced Lighting plug-in (RadiosityEffect) as the active one in the UI. When switching Advanced Lighting types in the UI, a dialog sometimes appears, warning that the current lighting solution will be discarded; but this dialog is not displayed when using SetRadiosity(). It is also valid to pass NULL as a parameter, in which case no active lighting plug-in will be active.

**Parameters:**

**RadiosityEffect\* op**

The RadiosityEffect instance to be made active, or NULL if no lighting plug-in should be active.

# Class IRadiosityEffectExtension

See Also: : [Class RadiosityEffect](#)

```
class IRadiosityEffectExtension : public BaseInterface
```

## Description:

This class is only available in release 5 or later.

This class provides additional functionality for class RadiosityEffect. Given an instance of RadiosityEffect, you may retrieve the extension interface as follows:

```
IRadiosityEffectExtension* r = static_cast<IRadiosityEffectExtension*>  
(radiosityInstance->GetInterface(IRADIOSITYEFFECT_EXTENSION_INTERFACE));
```

If the result is NULL, the RadiosityEffect does not support this interface.

All methods of this class are implemented by the plug-in.

## Methods:

### Prototype:

```
virtual bool useDefaultLight(const DefaultLight& defLight) const =  
0
```

## Remarks:

Returns whether the specified default light should be used by the scanline renderer. The scanline renderer normally creates default lights when there are no lights in the scene. A radiosity plug-in could override this if it uses objects other than lights as light sources (e.g. self-emitting surfaces)

## Parameters:

**Const DefaultLight& defLight**

A default light created by the scanline renderer when it begins rendering.

## Return value:

Returns whether or not the scanline renderer should use the light for the current rendering.

**Prototype:**

**virtual bool IsInterestedInChannels(PartID part) const**

**Remarks:**

This is used to control reference messages sent to the RadiosityEffect plug-in. It allows the RadiosityEffect to tell the system which messages will not invalidate the lighting solution.

If the RadiosityEffect decides that all messages of a given PartID are irrelevant to the lighting solution, it can return false when that PartID flag is present in the input parameter (and no other, more relevant, PartID flags are present). Otherwise it should return true, indicating it needs those messages.

If the return value is false, the system will add the PartID flag `PART_EXCLUDE_RADIOSITY` to all appropriate reference messages; this flag generically indicates an event which should not invalidate a radiosity solution. The RadiosityEffect will still receive the message but can ignore it. Other modules which need to discriminate messages pertinent radiosity, can also check this flag.

As an example of when this is important, an edit mesh modifier may send change messages which are meant to flush internal caches but not relevant to radiosity. As the message propagates, some PartID flags are added in order to force modifiers to re-evaluate downstream in the stack. This may confuse the radiosity engine into invalidating its lighting solution. But the edit mesh ensures the original message bears the `PART_EXCLUDE_RADIOSITY` flag, and as subsequent messages inherit the flag, the RadiosityEffect can correctly ignore the message.

**Parameters:****PartID part**

One or more PartID flags defining the category of messages to be filtered.

**Return value:**

Whether or not to filter the given messages, by setting their

PART\_EXCLUDE\_RADIOSITY PartID flag.

**Prototype:**

**virtual Interface\_ID GetID()**

**Remarks:**

This returns the ID of the interface, IRADIOSITYEFFECT\_EXTENSION\_INTERFACE, and should not be overridden by an implementation class.

**Return value:**

The IRadiosityEffectExtension interface ID, IRADIOSITYEFFECT\_EXTENSION\_INTERFACE.

**Default Implementation:**

```
{ return IRADIOSITYEFFECT_EXTENSION_INTERFACE; }
```

# Class IRadiosityPreferences

See Also: : [Class RadiosityEffect](#)

class IRadiosityPreferences : public FPStaticInterface

## Description:

This class is only available in release 5 or later.

This class defines the interface for accessing the Advanced Lighting Preferences from the 'Advanced Lighting' tab in the preferences dialog. It also provides access to the use/compute advanced lighting controls found in the Rendering dialog. This is a function-published static interface; you can use GetCOREInterface() to obtain an instance of the class, as follows:

```
IRadiosityPreferences* r = static_cast<IRadiosityPreferences*>  
(GetCOREInterface(IRADIOSITYPREFERENCES_INTERFACE));
```

This interface is also accessible via MAXScript as “RadiosityPreferences”.

All methods of this class are implemented by the system.

## Methods:

### Prototype:

**virtual BOOL GetAutoProcessObjectRefine() const = 0**

### Remarks:

Returns the state of the checkbox “Automatically Process Refine Iterations Stored in Geometric Objects”, in the preferences dialog Advanced Lighting tab

### Prototype:

**virtual void SetAutoProcessObjectRefine(BOOL val) = 0**

### Remarks:

Sets the state of the checkbox “Automatically Process Refine Iterations Stored in Geometric Objects”, in the preferences dialog Advanced Lighting tab

**Parameters:**

**BOOL val**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetDisplayReflectanceInMEditor() const = 0**

**Remarks:**

Returns the state of the checkbox “Display Reflectance & Transmittance Information”, in the preferences dialog Advanced Lighting tab

**Prototype:**

**virtual void SetDisplayReflectanceInMEditor(BOOL val) = 0**

**Remarks:**

Sets the state of the checkbox “Display Reflectance & Transmittance Information”, in the preferences dialog Advanced Lighting tab

**Parameters:**

**BOOL val**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetDisplayInViewport() const = 0**

**Remarks:**

Returns the state of the checkbox “Display Radiosity in Viewports”, in the preferences dialog Advanced Lighting tab

**Prototype:**

**virtual void SetDisplayInViewport(BOOL val) = 0**

**Remarks:**

Sets the state of the checkbox “Display Radiosity in Viewports”, in the preferences dialog Advanced Lighting tab

**Parameters:**

**BOOL val**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetDisplayResetWarning() const = 0**

**Remarks:**

Returns the state of the checkbox “Display Reset Warning”, in the preferences dialog Advanced Lighting tab

**Prototype:**

**virtual void SetDisplayResetWarning(BOOL val) = 0**

**Remarks:**

Sets the state of the checkbox “Display Reset Warning”, in the preferences dialog Advanced Lighting tab

**Parameters:**

**BOOL val**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetDisplayWarningOnGIPropsChange() const = 0**

**Remarks:**

Returns the state of the checkbox “Display Undo Warning on Properties Change”, in the preferences dialog Advanced Lighting tab

**Prototype:**

**virtual void SetDisplayWarningOnGIPropsChange(BOOL val) = 0**

**Remarks:**

Sets the state of the checkbox “Display Undo Warning on Properties Change”, in the preferences dialog Advanced Lighting tab

**Parameters:**

**BOOL val**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetSaveScene() const = 0**

**Remarks:**

Returns the state of the checkbox “Save Scene Information in MAX File”, in the preferences dialog Advanced Lighting tab

**Prototype:**

**virtual void SetSaveScene(BOOL val) = 0**

**Remarks:**

Sets the state of the checkbox “Save Scene Information in MAX File”, in the

preferences dialog Advanced Lighting tab

**Parameters:**

**BOOL val**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetUseRadiosity() const = 0**

**Remarks:**

Returns the state of the “Use Advanced Lighting” checkbox in the render dialog

**Prototype:**

**virtual void SetUseRadiosity(BOOL val) = 0**

**Remarks:**

Sets the state of the “Use Advanced Lighting” checkbox in the render dialog

**Parameters:**

**BOOL val**

TRUE for on; FALSE for off.

**Prototype:**

**virtual BOOL GetComputeRadiosity() const = 0**

**Remarks:**

Returns the state of the “Compute Advanced Lighting when Required” checkbox in the render dialog

**Prototype:**

**virtual void SetComputeRadiosity(BOOL val) = 0**

**Remarks:**

Sets the state of the “Compute Advanced Lighting when Required” checkbox in the render dialog

**Parameters:**

**BOOL val**

TRUE for on; FALSE for off.

## Class IDXShaderManagerInterface

See Also: [Class IViewportShaderManager](#)

```
class IDXShaderManagerInterface : public FPStaticInterface
```

### Description:

**This class is only available in release 5 or later.**

The class provides access to the Viewport Manager feature in release 5. The viewport manager controls the loading of Viewport Shaders in 3ds max and is displayed on each material. The class provides methods to query the manager to find out whether it is visible or active, and also to retrieve the active shader.

The viewport manager ONLY works in DirectX mode. It can however be visible in Heidi and OpenGL, so as to provide the artist feedback when loading someone else's file.

There is a global method you can call to get access to the DX manager.

**IDXShaderManagerInterface\* GetDXShaderManager()**

### Methods:

#### Prototype:

**virtual CustAttrib\* FindViewportShaderManager (MtlBase\* mtl)=0**

### Remarks:

This method will check the Material passed into find out whether it contains the ViewportManager Custom Attribute. If it does it will return its pointer, otherwise it will be NULL. This pointer can safely be cast to

**IViewportShaderManager.**

### Parameters:

**MtlBase\* mtl**

The material to search for the Viewport Manager.

**Prototype:**

**virtual CustAttrib\* AddViewportShaderManager(MtlBase\* mtl)=0**

**Remarks:**

This method will add the ViewportManager custom attribute to the material supplied. If successful it will return the newly created custom attribute. This pointer can safely be cast to **IViewportShaderManager**.

**Parameters:**

**MtlBase\* mtl**

The material to add the Viewport Manager to

**Prototype:**

**virtual void SetVisible(BOOL show=TRUE)=0**

**Remarks:**

This method allows the system to either show or hide the ViewportManager. This will physically remove it from the Material Editor, whoever the manager will still exist on the material

**Parameters:**

**BOOL show**

The value to set the visible state of the manager.

**Prototype:**

**virtual BOOL IsVisible ()=0**

**Remarks:**

This method will return the actual visible state of the manager

## Class IHardwareRenderer

### Description:

**This class is only available in release 5 or later.**

This interface provides access to the DirectX interfaces on the drawing thread of max. This allows creation and loading of various DirectX objects such as Textures. This class has many interfaces not currently implemented, although they will compile and link correctly. This is due to support for DirectX 9, meaning that when a compatible driver is released these methods will be implemented. The documentation for this class only deals with methods that developers can and would normally use.

The basic idea of this class is to provide the developer with access to the DirectX device without actually giving the developer total control, and thus possibility to de-stable the 3ds max Viewports.

A pointer to this class can be obtained with the following code snippet: -

```
ViewExp *pview = GetCOREInterface()->GetActiveViewport();  
GraphicsWindow *gw = pview->getGW();  
IHardwareRenderer * phr = (IHardwareRenderer *)gw->  
>GetInterface(HARDWARE_RENDERER_INTERFACE_ID);
```

For an example usage of this class see

**MAXSDK\SAMPLES\HardwareShaders\LightMap\Lightmap.cpp**

### Methods:

### Prototype:

```
Virtual DWORD_PTR BuildTexture(BITMAPINFO *bmi, UINT  
miplevels, DWORD usage, DWORD format)
```

### Remarks:

This will create a DX texture stored in local storage. The pointer returned can be used in IHardwareMaterial::SetTexture method. An example of using this

method can be seen in the Lightmap sample.

**Parameters:**

**BITMAPINFO \*bmi**

A pointer to the bitmap from which the texture will be created

**UINT miplevels**

The number of miplevels to create

**DWORD usage**

The usage flag is the same as for D3DXCreateTexture – see the DirectX documentation for more information

**DWORD format**

The pixel format for the texture

**Prototype:**

**Virtual DWORD\_PTR LoadTexture(LPCSTR filename)**

**Remarks:**

This simple loads a texture from the supplied filename using the default options for D3DXCreateTextureFromFile. Please refer to the DirectX documentation for further information.

## Class MNNormalSpec

See Also: [Class MNMESH](#) , [Class MNNormalFace](#)

```
class MNNormalSpec : public IPipelineClient, public FlagUser
```

### Description:

This class is available in release 5.0 and later only.

This class is an interface used to store user-specified normals (as created in the Edit Normals modifier). These normals have very limited pipeline support. They are used for viewport display, but not for rendering.

The MNNormalSpec contains three types of normals:

- **Unspecified** - these are the usual normals that are computed from smoothing groups. All normals are unspecified by default.
- **Specified** - these are normals that are intended for use by particular corners of particular faces, without regard to smoothing groups. For instance, you can create a box, apply Edit Normals, select a group of normals at a particular vertex, and click "Unify". Now those three faces are told to specifically use that one unified normal, and they ignore their smoothing groups at that vertex (which would normally tell them they should each have their own normal).
- **Explicit** - these are normals that are set to particular values. For instance, if the user wants to use the Edit Normals Move or Rotate commands to set a normal to something other than its default value, it has to be made explicit, so it won't be recomputed based on the face normals. All explicit normals are also considered to be

specified..

**Flags:**

**MNNORMAL\_NORMALS\_BUILT**

Indicates that non-specified normals have been constructed using smoothing groups. If not set, non-specified normals may be invalid.

**MNNORMAL\_NORMALS\_COMPUTED**

Indicates that non-explicit normals have been computed using geometrically computed face normals. (If not set, only explicit normals may be assumed to be pointing the right direction.)

**Data Members**

All data members are private.

**int mNumNormalAlloc, mNumFaceAlloc;**

The current allocation length of the mpNormal and mpFace arrays.

**int mNumNormals, mNumFaces;**

The number of normals and faces in the mpNormal and mpFace arrays.

(May be less than the actual allocation above.)

**MNNormalFace \*mpFace;**

The array of normal faces.

**Point3 \*mpNormal;**

The array of normals, all of which should be either length 1 or (occasionally) 0.

**BitArray mNormalExplicit;**

Indicates whether mpNormal[i] is explicit or computed from face normals.

**BitArray mNormalSel;**

Current normal selection.

**float mDisplayLength;**

The length to use when displaying, hit testing, or moving normals.

**MNMesh \*mpParent;**

A pointer to the "parent" MNMesh that owns this MNNormalSpec. This parent information is required for some operations, such as display. (Such operations should indicate below where parent information is required.)

:

**Methods:****Prototype:****MNNormalSpec ();****Remarks:**

Constructor. Initializes all data members.

**Prototype:****~ MNNormalSpec ();****Remarks:**

Destructor. Calls ClearAndFree().

**Prototype:**  
**void Initialize();**

**Remarks:**  
Initializes all data members. Do not call if memory has already been allocated, or that memory will be leaked.

**Prototype:**  
**bool NAlloc (int num, bool keep=TRUE);**

**Remarks:**  
Sets the size of the normal array

**Return Value:**  
True if successful; false indicates a failed memory allocation.

**Prototype:**  
**void NShrink ();**

**Remarks:**  
Reduces the allocation size down to the actual number of normals.

**Prototype:**  
**bool FAlloc (int num, bool keep=TRUE);**

**Remarks:**

Sets the size of the face array.

**Return Value:**

True if successful; false indicates a failed memory allocation.

**Prototype:**

```
void FShrink ();
```

**Remarks:**

Reduces the allocation size down to the actual number of faces.

**Prototype:**

```
void Clear ();
```

**Remarks:**

Clears out all data, but doesn't necessarily free array memory.

**Prototype:**

```
void ClearAndFree ();
```

**Remarks:**

Clears out all data and frees all memory.

**Prototype:**

```
int GetNumFaces () const
```

**Remarks:**

Returns the current number of faces in the MNNormalSpec

**Prototype:**

**bool SetNumFaces (int numFaces);**

**Remarks:**

Sets the current number of faces in the MNNormalSpec, increasing the allocation size as needed.

**Return Value:**

True if successful; false indicates a failed memory allocation.

**Prototype:**

**int GetNumNormals () const**

**Remarks:**

Returns the current number of normals in the MNNormalSpec

**Prototype:**

**bool SetNumNormals (int numNormals);**

**Remarks:**

Sets the current number of normals in the MNNormalSpec, increasing the allocation size as needed.

**Return Value:**

True if successful; false indicates a failed memory allocation.

**Prototype:**  
**Point3 & Normal (int normID) const**

**Remarks:**  
Returns the normal indicated. Since it returns a reference, you can use it as a set method as well:  
Normal(i) = Normalize (Point3(1,1,0));  
(Note that all normals should be normalized to a length of 1.)

**Prototype:**  
**Point3 \* GetNormalArray () const**

**Remarks:**  
Returns a pointer to the whole normal array.

**Prototype:**  
**bool GetNormalExplicit (int normID) const**

**Remarks:**  
Indicates whether a given normal is explicit or not.

**Prototype:**  
**void SetNormalExplicit (int normID, bool value);**

**Remarks:**  
Sets a particular normal to be explicit or not. Note that if you make a normal non-explicit, it may need to be recomputed, so you may want to call ComputeNormals or at least clear the MNNORMAL\_NORMALS\_COMPUTED flag.

**Parameters:**

**int normID**

The index of the normal

**bool value**

True to make the normal explicit; false to make it non-explicit.

**Prototype:**

**MNNormalFace & Face(int faceID) const**

**Remarks:**

Returns the indicated face.

**Prototype:**

**MNNormalFace \* GetFaceArray () const**

**Remarks:**

Returns a pointer to the whole face array.

**Prototype:**

**void SetParent (MNMesh \*pMesh);**

**Remarks:**

Tells the MNNormalSpec what MNMesh "owns" it. This "Parent" MNMesh is used in methods such as Display, Hit-Testing, and certain operations like Unify to get information about the vertices that

normals are based on. (There's no vertex info in the MNNormalSpec itself.)

If you have an isolated MNNormalSpec which doesn't really have an associated "parent", you can temporarily set this to a mesh with the right sort of faces and vertices, but you should clear it afterwards by calling SetParent (NULL). See the Edit Normals modifier source in maxsdk\samples\mesh\EditablePoly\EditNormals.cpp for an example of this sort of usage.

**Prototype:**

**Point3 & GetNormal (int face, int corner);**

**Remarks:**

Returns the normal used by the indicated face, in the indicated corner.

**Prototype:**

**void SetNormal (int face, int corner, Point3 & normal);**

**Remarks:**

Creates a new (explicit) normal and uses it in the indicated corner of the indicated face. If "normal" is not already normalized, this method will take care of it.

**Prototype:**

**int GetNormalIndex (int face, int corner);**

**Remarks:**

Returns the index of the normal used in the indicated corner of the indicated face.

**Prototype:**

**void SetNormalIndex (int face, int corner, int normalIndex);**

**Remarks:**

Sets the index of the normal used in the indicated corner of the indicated face, and marks it as specified.

**Prototype:**

**int NewNormal (Point3 & normal, bool explic=true);**

**Remarks:**

Creates a new normal at the end of the normal array.

**Parameters:****Point3 & normal**

The desired normal direction. Will be normalized to a length of 1 by the method if needed.

**bool explic=true**

Indicates whether the new normal should be considered explicit or not.

**Prototype:**

**void SetSelection (BitArray & newSelection);**

**Remarks:**

Sets the current normal selection.

**Prototype:**

**BitArray & GetSelection();**

**Remarks:**

Returns the current normal selection.

**Prototype:**

**void SetDisplayLength (float displayLength);**

**Remarks:**

Sets the current length used for normal display, hit-testing, and Translations.

**Prototype:**

**float GetDisplayLength ();**

**Remarks:**

Returns the current length used for normal display, hit-testing, and Translations.

**Prototype:**

**void Display (GraphicsWindow \*gw, bool showSel);**

**Remarks:**

Requires an accurate "parent" pointer (see SetParent).

Displays the current normals in the graphics window indicated.  
If "showSel" is true, selected normals are displayed in the usual subobject selection color.

**Prototype:**

**bool HitTest (GraphicsWindow \*gw, HitRegion \*hr, DWORD flags, SubObjHitList& hitList);**

**Remarks:**

Requires an accurate "parent" pointer (see SetParent). Hit-tests on the current normals.

**Parameters:**

**GraphicsWindow \*gw**

The window to hit-test in.

**HitRegion \*hr**

A hit region, typically generated by a call like  
MakeHitRegion(hr,type, crossing,4,p);

**DWORD flags**

Hit testing flags. Please see BaseObject::HitTest for a description of these flags and of the "type" and "crossing" variables used to generate the HitRegion.

**SubObjHitList & hitList**

Where the hits get stored.

**Return Value:**

True if a hit was found; false if not.

**Prototype:**

**Box3 GetBoundingBox (Matrix3 \*tm=NULL, bool selectedOnly=false);**

**Remarks:**

Requires an accurate "parent" pointer (see SetParent). Computes the bounding box of the normals.

**Parameters:**

**Matrix3 tm=NULL**

An optional transform for computing the bounding box in a different space (such as world space).

**bool selectedOnly=false**

Indicates whether all normals should be included in the bounding box,  
or only selected ones.

**Prototype:**

**void ClearNormals ();**

**Remarks:**

This method dumps all unspecified normals. Best to use only from within BuildNormals, since it leaves all unspecified normals in faces initialized to -1.

**Prototype:****void CollapseDeadFaces ();****Remarks:**

Requires an accurate "parent" pointer (see SetParent).

This method is used in conjunction with the parent MNMesh's CollapseDeadFaces method to keep the normal faces in synch with the parent MNMesh faces. It removes any normal face whose equivalent face in the parent mesh is considered "Dead".

Called by MNMesh::CollapseDeadFaces, so you generally don't need or want to call it separately.

**Prototype:****void BuildNormals ();****Remarks:**

Requires an accurate "parent" pointer (see SetParent).

Fills in the mpSpecNormal data by building all the unspecified normals, and computing non-explicit ones. Does nothing if face array is not allocated yet!

**Prototype:****void ComputeNormals ();****Remarks:**

Requires an accurate "parent" pointer (see SetParent).

This method just recomputes the directions of non-explicit normals, without rebuilding the normal list.

**Prototype:**

**void CheckNormals ();**

**Remarks:**

Requires an accurate "parent" pointer (see SetParent).

This checks our flags and calls BuildNormals or ComputeNormals as needed.

**Prototype:**

**MNNormalSpec & operator= (const MNNormalSpec & from);**

**Remarks:**

Typical = operator. Allocates arrays in this, and makes copies of all the data in "from". Does NOT copy "Parent" pointer.

**Prototype:**

**void CopySpecified (const MNNormalSpec & from);**

**Remarks:**

This is similar to operator=, but copies only the specified and explicit information from "from". Result will need to have BuildNormals and ComputeNormals called.

**Prototype:**

**MNNormalSpec & operator+= (const MNNormalSpec & from);**

**Remarks:**

Adds the faces and normals from "from" to our normal spec, renumbering the normals so they don't conflict with existing ones. Called by the "AppendAllChannels" method below (which itself is called by MNMesh::operator+=).

**Prototype:**

**void MNDebugPrint (bool printAll=false);**

**Remarks:**

Uses "DebugPrint" to output information about this MNNormalSpece to the Debug buffer in DevStudio.

**Parameters:**

**bool printAll=false**

If false, only explicit normals and faces using specified normals will be printed out. If true, all normals and faces will be completely printed out.

Here is what the output looks like on a box with mostly default (non-specified) normals, but with one corner "Unified" into a single specified normal:

If printAll = true, you'll see:

MNNormalSpec Debug Output: 22 normals, 6 faces

Normal (Non ) 0: 0.577350, -0.577350, 0.577350

Normal (Non ) 1: 0.000000, 0.000000, -1.000000  
Normal (Non ) 2: 0.000000, 0.000000, -1.000000  
Normal (Non ) 3: 0.000000, 0.000000, -1.000000  
Normal (Non ) 4: 0.000000, 0.000000, -1.000000  
Normal (Non ) 5: 0.000000, 0.000000, 1.000000  
Normal (Non ) 6: 0.000000, 0.000000, 1.000000  
Normal (Non ) 7: 0.000000, 0.000000, 1.000000  
Normal (Non ) 8: 0.000000, -1.000000, 0.000000  
Normal (Non ) 9: 0.000000, -1.000000, 0.000000  
Normal (Non ) 10: 0.000000, -1.000000, 0.000000  
Normal (Non ) 11: 1.000000, 0.000000, 0.000000  
Normal (Non ) 12: 1.000000, 0.000000, 0.000000  
Normal (Non ) 13: 1.000000, 0.000000, 0.000000  
Normal (Non ) 14: 0.000000, 1.000000, 0.000000  
Normal (Non ) 15: 0.000000, 1.000000, 0.000000  
Normal (Non ) 16: 0.000000, 1.000000, 0.000000  
Normal (Non ) 17: 0.000000, 1.000000, 0.000000  
Normal (Non ) 18: -1.000000, 0.000000, 0.000000  
Normal (Non ) 19: -1.000000, 0.000000, 0.000000  
Normal (Non ) 20: -1.000000, 0.000000, 0.000000  
Normal (Non ) 21: -1.000000, 0.000000, 0.000000

Normal Selection:

0 6 13 17

Normal faces: \_ before normal index means non-specified.

Face 0: \_1 \_2 \_3 \_4

Face 1: \_5 0 \_6 \_7

Face 2: \_8 \_9 0 \_10

Face 3: \_11 \_12 \_13 0

Face 4: \_14 \_15 \_16 \_17

Face 5: \_18 \_19 \_20 \_21

If printAll is false, you'll just see:

MNNormalSpec Debug Output: 22 normals, 6 faces

Normal Selection:

0 6 13 17

Face 1: \_ 0 \_ \_

Face 2: \_ \_ 0 \_

Face 3: \_ \_ \_ 0

**Prototype:**

**bool CheckAllData (int numParentFaces);**

**Remarks:**

Performs a series of internal checks to verify that the normal data is consistent. If there are any problems, messages are sent out via DebugPrint.

**Parameters:**

**int numParentFaces**

The number of faces in the parent MNMesh. (This method does not require an accurate "parent" pointer, but it does require this number to be accurate.

**Return Value:**

True if everything checks out ok, false if not.

**Prototype:**

**IOResult Save (ISave \*isave);**

**Remarks:**

Called by the system. Saves the MNNormalSpec to the buffer.

**Prototype:**  
**IOResult Load (ILoad \*iload);**

**Remarks:**  
Called by the system. Loads the MNNormalSpec from the buffer.

**Prototype:**  
**bool Transform (Matrix3 & xfm, BOOL useSel=false, BitArray  
\*normalSelection=NULL);**

**Remarks:**  
Transforms the normals. Note that since normals are always considered to be unit length vectors, scales and translations are not effective. Translations drop out because we use the VectorTransform method to transform the normals, and scales drop out because we renormalize the normals to a length of 1 afterwards.

**Parameters:**  
**Matrix3 & xfm**  
The desired transform.

**BOOL useSel=false**  
Indicates whether all normals should be translated, or only selected ones.

**BitArray \*normalSelection=NULL**  
The desired transform.

**BOOL useSel=false**

If non-NULL, this represents a selection of normals that should be used instead of the usual selection, when deciding which normals to transform.

**Return Value:**

True if something was modified. False would indicate that no normals were changed, perhaps because there are no normals in the spec or because none were selected.

**Prototype:**

**bool Translate (Point3 & translate, BOOL useSel=true, BitArray \*normalSelection=NULL);**

**Remarks:**

This is used to give a translation-like effect to normals. It's used in the Edit

Normals "Move" mode. Essentially it drags the "top" of the normals by the amount

given, and then renormalizes the vectors to unit length. It uses the current

display length as well, so the formula is basically

$mpNormal[i] = \text{Normalize}(mpNormal[i]*mDisplayLength + \text{translate});$

This gives a fairly natural result in Edit Normals Move.

**Parameters:**

**Point3 & translate**

The desired translation.

**BOOL useSel=false**

Indicates whether all normals should be translated, or only selected ones.

**BitArray \*normalSelection=NULL**

If non-NULL, this represents a selection of normals that should be used instead of the usual selection, when deciding which normals to translate.

**Return Value:**

True if something was modified. False would indicate that no normals were changed, perhaps because there are no normals in the spec or because none were selected.

**Prototype:**

```
bool BreakNormals (BOOL useSel=true, BitArray  
*normalSelection=NULL);
```

**Remarks:**

"Breaks" normals so that none are shared between faces. Broken normals are set to Specified (but not explicit.)

**Parameters:****BOOL useSel=false**

Indicates whether all normals should be affected, or only selected ones.

**BitArray \*normalSelection=NULL**

If non-NULL, this represents a selection of normals that should be used instead of

the usual selection, when deciding which normals to affect.  
(Irrelevant if  
useSel=false.)

**Return Value:**

True if something was modified. False would indicate that no normals were changed,  
perhaps because there are no normals present or because none were selected,  
or because selected normals were already fully broken and specified.

If the return value is true, the MNNORMAL\_NORMALS\_BUILT and MNNORMAL\_NORMALS\_COMPUTED flags are cleared, because the newly broken normals need to be rebuilt and computed.

**Prototype:**

**bool UnifyNormals (BOOL useSel=true, BitArray  
\*normalSelection=NULL);**

**Remarks:**

Requires an accurate "parent" pointer (see SetParent).

This method unifies selected normals so that there's a maximum of one per vertex. For instance, a default box has 3 normals at every vertex. You can select 2 or 3 of them and click "Unify" in Edit Normals, and the normals will be shared across the faces that use them. See Edit Normals documentation for more information.

This method does not unify normals that are based at different vertices. If you want separate vertices to use the same normal, you must use more direct, low-level methods like SetNormalIndex.

Unified normals are set to specified (but not explicit).

**Parameters:**

**BOOL useSel=false**

Indicates whether all normals should be affected, or only selected ones.

**BitArray \*normalSelection=NULL**

If non-NULL, this represents a selection of normals that should be used instead of the usual selection, when deciding which normals to affect. (Irrelevant if useSel=false.)

**Return Value:**

True if something was modified. False would indicate that no normals were changed, perhaps because there are no normals present or because none were selected, or because selected normals were already fully unified and specified.

If the return value is true, the `MNNORMAL_NORMALS_BUILT` and `MNNORMAL_NORMALS_COMPUTED` flags are cleared, because the newly unified normals need to be rebuilt and computed.

**Prototype:**

**bool SpecifyNormals (BOOL useSel=true, BitArray \*normalSelection=NULL);**

**Remarks:**

Specifies the indicated normals. DOESN'T remove the explicitness of the normals.

(That should be done separately with `MakeNormalsExplicit`, `value=false`.)

**Parameters:**

**BOOL useSel=false**

Indicates whether all normals should be affected, or only selected ones.

**BitArray \*normalSelection=NULL**

If non-NULL, this represents a selection of normals that should be used instead of the usual selection, when deciding which normals to affect. (Irrelevant if `useSel=false`.)

**Return Value:**

True if something was modified. False would indicate that no normals were changed, perhaps because there are no normals present or because none were selected, or because they were all already specified.

**Prototype:**

**bool MakeNormalsExplicit (BOOL useSel=true, BitArray \*normalSelection=NULL, bool value=true);**

**Remarks:**

Makes the indicated normals both specified and explicit.

**Parameters:**

**BOOL useSel=false**

Indicates whether all normals should be affected, or only selected

ones.

**BitArray \*normalSelection=NULL**

If non-NULL, this represents a selection of normals that should be used instead of the usual selection, when deciding which normals to affect. (Irrelevant if useSel=false.)

**bool value=true**

I Indicates whether the normals should be set to explicit, or non-explicit.

**Return Value:**

True if something was modified. False would indicate that no normals were changed, perhaps because there are no normals present or because none were selected.

If value=false and the return value is true, the MNNORMAL\_NORMALS\_COMPUTED flag is cleared, because the newly nonexplicit normals need to be computed

.

**Prototype:**

**bool ResetNormals (BOOL useSel=true, BitArray \*normalSelection=NULL);**

**Remarks:**

Resets the indicated normals to be completely non-explicit and non-specified.

**Parameters:****BOOL useSel=false**

Indicates whether all normals should be affected, or only selected ones.

**BitArray \*normalSelection=NULL**

If non-NULL, this represents a selection of normals that should be used instead of

the usual selection, when deciding which normals to affect.

(Irrelevant if useSel=false.)

**Return Value:**

True if something was modified. False would indicate that no normals were changed,

perhaps because there are no normals present or because none were selected.

If the return value is true, the MNNORMAL\_NORMALS\_BUILT and MNNORMAL\_NORMALS\_COMPUTED flags are cleared, because the newly nonspecified normals need to be rebuilt and computed.

## List of Expression Types

One of the following values:

### **SCALAR\_EXPR**

Scalar expression. This is a single floating point value.

### **VECTOR\_EXPR**

Vector expression. The **eval()** method returns the result as an array of floating point values:

**ans[0] = x, ans[1] = y, ans[2] = z.**

The method **getExprType()** may be used to determine the type of the expression result.

## List of Expression Variable Types

See Also: [Class Point3](#).

One of the following values:

### **SCALAR\_VAR**

Scalar variable. This is a single floating point value.

### **VECTOR\_VAR**

Vector variable. This is a **Point3** value when passed to the **eval()** method. The **x**, **y**, **z** public data members of the **Point3** are the values representing the vector. Vectors are specified in an expression by enclosing the three values in square brackets, i.e. "**[]**". For example, this is a unit vector parallel to the world Z axis: **[0,0,1]**.

## List of Expression Return Codes

One of the following values:

### **EXPR\_NORMAL**

No problems, expression evaluated successfully.

### **EXPR\_INST\_OVERFLOW**

Expression caused an instruction stack overflow during parsing.

### **EXPR\_UNKNOWN\_TOKEN**

Unknown function, constant, or register found during parsing.

### **EXPR\_TOO\_MANY\_VARS**

Expression caused a value stack overflow.

### **EXPR\_TOO\_MANY\_REGS**

Register array overflow, or register number too big.

### **EXPR\_CANT\_EVAL**

Function can't be evaluated with given arguments.

### **EXPR\_CANT\_PARSE**

Expression can't be parsed syntactically.

## List of FPStatus Values

See Also: [Class FPInterface](#), [Function Publishing System](#).

One of the following values are returned from the various FPInterface methods:

### **FPS\_FAIL**

This status indicates a failure.

### **FPS\_NO\_SUCH\_FUNCTION**

The function called does not exist.

### **FPS\_ACTION\_DISABLED**

The action is disabled.

### **FPS\_OK**

Indicates a success.

## List of ControlType2 Choices

See Also: [Class IParamBlock2](#), [Structure ParamDef](#), [Class ParamBlockDesc2](#).

### Description:

This data is available in release 3.0 and later only.

This is the list of user interface control choices available for use with parameter block2s. There is one of these associated with each parameter definition in a **ParamBlockDesc2**.

### enum ControlType2

{

#### **TYPE\_SPINNER**

A custom spinner control.

#### **TYPE\_RADIO**

A radio button control.

#### **TYPE\_SINGLECHECKBOX**

A single checkbox control.

#### **TYPE\_MULTICHECKBOX**

A multiple checkbox control. This control type is not currently supported.

#### **TYPE\_COLORSWATCH**

A color swatch color control.

#### **TYPE\_EDITBOX**

An edit box control.

#### **TYPE\_CHECKBUTTON**

A check button control. This button's state is either pressed or not.

#### **TYPE\_PICKNODEBUTTON**

A node picker button.

#### **TYPE\_TEXMAPBUTTON**

A texture map selector button. This button supports drag and drop of texmaps.

#### **TYPE\_MTLBUTTON**

A material selector button. This button supports drag and drop of materials.

#### **TYPE\_FILEOPENBUTTON**

A file open button.

#### **TYPE\_FILESAVEBUTTON**

A file save button.

**TYPE\_INTLISTBOX**

A list box control for integers.

**TYPE\_FLOATLISTBOX**

A list box control for floats.

**TYPE\_STRINGLISTBOX**

A list box control for strings.

**TYPE\_NODELISTBOX**

A list box control for nodes.

**TYPE\_MAPLISTBOX**

A list box control for maps.

**TYPE\_SLIDER**

A custom slider control.

};

# Class MaxNetCallback

See Also: [Class MaxNetManager](#)

class MaxNetCallback

## Description:

This class is available in release 4.0 and later only.

If you want to use the call back mechanisms provided by the MaxNetManager class, you create your own class derived from MaxNetCallback and pass it as the argument for **MaxNetManager::SetCallback()**. All methods are optional. You need only to implement those you want.

## Methods:

### Prototype:

**bool Progress(int total, int current);**

### Remarks:

This method is called whenever a lengthy operation is under way. This includes large block transfers, file transfers, etc.

### Parameters:

**int total**

A total amount of information to process.

**int current**

From the total amount, this is the current position.

### Return Value:

TRUE if you want to cancel the operation, otherwise FALSE.

### Prototype:

**void ProgressMsg(const TCHAR \*message);**

### Remarks:

This method is called to provide a textual message regarding the current process which is under way (connecting to a Manager, waiting for a reply, etc.)

### Parameters:

**const TCHAR\* message**

The text message.

**Prototype:**

**void ManagerDown();**

**Remarks:**

This method is called for messages which are sent to indicate that the Manager was shut down and is no longer available to service any requests.

**Prototype:**

**void Update();**

**Remarks:**

This method is called to indicate something has changed. This message is sent whenever a new job is sent to the queue, a Server changed its state, an error occurred, etc. This allows you to be kept up to date without the need to poll the Manager from time to time. Note that these messages are queued up. That is, if 10 jobs are deleted at once or any number of changes occur within a short period, you will only receive one **Update()** call.

**Prototype:**

**void QueryControl(TCHAR\* station);**

**Remarks:**

This method is called if someone requests control over the queue. If you do not have control over the queue, you do not need to respond to this message. If you have control over the queue and do nothing when this call is received, the Manager will timeout and take control over the queue. If you have control over the queue, you should respond to this message using **MaxNetManager::GrantManagerControl()** passing either a true value, indicating you are granting control, or a false value, indicating you do not want to relinquish control.

**Parameters:**

**TCHAR\* station**

The name of the computer requesting control.

**Prototype:**

```
void QueueControl();
```

**Remarks:**

This method is called in order to notify that someone took control of the queue.

## Structure ManagerInfo

See Also: [Class MaxNetManager](#), [Structure NetworkStatus](#), [Structure ConfigurationBlock](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about a Manager.

### typedef struct {

**DWORD size;**

The size of the structure, being **sizeof(ManagerInfo)**.

**DWORD version;**

The version information, defined by **\_MANAGER\_INFO\_VERSION**.

**ConfigurationBlock cfg;**

The network system configuration data. Refer to this structure for more information.

**NetworkStatus net\_status;**

The network status information.

**int servers;**

The number of servers registered.

**int jobs;**

The number of jobs.

**char reserved[32];**

Reserved for future use.

**} ManagerInfo;**

## Structure ClientInfo

See Also: [Class MaxNetManager](#), [Structure ConfigurationBlock](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about a Client.

### typedef struct {

**DWORD size;**

The size of the structure, being **sizeof(ClientInfo)**.

**DWORD version;**

The version information, defined by **\_CLIENTINFO\_VERSION**.

**ConfigurationBlock cfg;**

The network system configuration data. Refer to this structure for more information.

**bool controller;**

TRUE if the Client is currently controlling the queue.

**short udp\_port;**

The UDP port being used for network communications.

**char reserved[32];**

Reserved for future use.

**} ClientInfo;**

## Structure JobList

See Also: [Class MaxNetManager](#), [Structure Job](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information the state of a job.

### typedef struct {

#### **Job job;**

The structure containing the job details.

#### **HJOB hJob;**

The handle to the job.

#### **WORD state;**

The current state of the job, which is one of the following values:

#### **JOB\_STATE\_COMPLETE**

The job is complete.

#### **JOB\_STATE\_WAITING**

The job is waiting to be rendered.

#### **JOB\_STATE\_BUSY**

The job is busy rendering.

#### **JOB\_STATE\_ERROR**

The job experienced an error.

#### **JOB\_STATE\_SUSPENDED**

The job is suspended.

### **} JobList;**

## Structure Job

See Also: [Class MaxNetManager](#), [Structure AlertData](#), [Structure MaxJob](#), [Structure CombustionJob](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about a job.

### typedef struct {

#### **DWORD size;**

The size of the structure, being **sizeof(Job)**.

#### **DWORD version;**

The structure version information, defined by **\_JOB\_VERSION**.

#### **DWORD server\_pid;**

The server Process ID which is used by 3ds max to check server's health.

#### **DWORD flags;**

The job flags, defined as the following values:

#### **JOB\_VP**

Video Post (otherwise is Render Scene).

#### **JOB\_NONC**

Non concurrent driver (Accom DDR, AVI, etc.)

#### **JOB\_MAPS**

The Include Maps flag.

#### **JOB\_NONSTOP**

Uninterruptible driver (AVI, FLC, etc.)

#### **JOB\_SKIPEXST**

Skip Existing Frames.

#### **JOB\_ALLSERVERS**

Allow the use of all available servers.

#### **JOB\_INACTIVE**

This flag indicates the job is suspended

#### **JOB\_COMPLETE**

This read-only flag indicates that the job is complete.

**JOB\_IGNORESHARE**

Ignore the Manager's job share, always request archives.

**JOB\_SKIPOUTPUTTST**

This flag indicates that the server should not test the output path.

**JOB\_NONSEQFRAMES**

Non sequential frames such as 1,3,5-10, etc.

**JOB\_COMBUSTIONJOB**

This flag indicates the job is a Combustion specific job.

**JOB\_NOTARCHIVED**

This flag indicates an uncompressed file (i.e. not an archive).

**JOB\_ASSIGN\_VP**

This is a legacy support flag, defined as **JOB\_VP**.

**JOB\_ASSIGN\_RND**

This is a legacy support flag.

**HJOB hJob;**

Handle to the job, assigned by the Manager when a job is created/submitted.

This handle will be read-only after its creation.

**char name[MAX\_PATH];**

The name of the job.

**DWORD filesize;**

This variable is used internally when transferring an archive and specifies its size.

**DWORD filesizeextracted;**

This variable is used internally when transferring an archive and specifies its uncompressed size.

**SYSTEMTIME submission;**

The system time, set when a job is created/submitted.

**SYSTEMTIME startjob;**

The system time, set when a job starts.

**SYSTEMTIME endjob;**

The system time, set when a job is completed.

**int servercount;**

The number of servers defined for the job (can be 0 if

**JOB\_ALLSERVERS** is set).

**AlertData alerts;**

The alert notification data structure.

**int jobtextcount;**

The number of JobTextInfo records.

**int firstframe;**

The first frame in the range.

**int lastframe;**

The last frame in the range.

**int step;**

The frame step value (i.e. every n-th frame).

**int width;**

The frame output width.

**int height;**

The frame output height.

**int frames\_completed;**

The number of frames completed.

**char priority;**

The job priority level.

**char reserved[32];**

Reserved for future use.

**union {**

**MaxJob maxJob**

    Specific information about a job pertaining to 3ds max.

**CombustionJob combustionJob**

    Specific information about a job pertaining to Combustion.

**}**

**} Job;**

## Class CJobText

See Also: [Class MaxNetCallBack](#), [Structure JobText](#), [List of Job Text Types](#)

class CJobText

### Description:

This class is available in release 4.0 and later only.

The CJobText class stores job information which is of a dynamic nature or of variable length.

### Methods:

public:

### Prototype:

**~CJobText();**

### Remarks:

Destructor. The CJobText buffers will be deallocated.

### Prototype:

**int Count();**

### Remarks:

This method return the number of JobText buffers in the list.

Note: Developers should use Job.jobtextcount to find out how many elements there are.

### Prototype:

**int Add(JobText\* jt);**

### Remarks:

This method will add another JobText buffer.

### Parameters:

**JobText\* jt**

A pointer to the JobText buffer to add.

### Return Value:

The number of JobText buffers.

**Prototype:**

**void Delete(int idx, int count = 1);**

**Remarks:**

This method will delete one or a sequence of buffers.

**Parameters:**

**int idx**

The position of the first index to be deleted.

**int count**

The number of entries to delete.

**Prototype:**

**void Reset();**

**Remarks:**

This method will reset and deallocate the CJobText buffers.

**Prototype:**

**JobText\* Buffer();**

**Remarks:**

This method will return a pointer to the actual JobText buffer.

**Prototype:**

**int BufferSize();**

**Remarks:**

This method returns the total size of the JobText buffer.

**Prototype:**

**int FindJobText(JOB\_TEXT\_TYPE tp, int start = 0);**

**Remarks:**

This method allows you to search for a JobText entry by its type. Refer to the list of Job Text types for details.

**Parameters:****JOB\_TEXT\_TYPE tp**

The Job Text type you wish to find.

**int start**

The start position from which to initiate the search process.

**Return Value:**

The index of the entry which was found, or -1 if not found.

**Prototype:**

```
bool GetTextItem(TCHAR* text, JOB_TEXT_TYPE type, int  
start = 0, int* idx = 0);
```

**Remarks:**

This method retrieves an index based on its Text Type. Refer to the [List of Job Text Types](#) for details.

**Parameters:****TCHAR\* text**

A pointer to the text buffer which will be filled in by the method.

**JOB\_TEXT\_TYPE type**

The Job Text type you wish to find.

**int start**

The start position from which to retrieve the text item.

**int\* idx**

A pointer to the index found.

**Return Value:**

TRUE if the method was successful, otherwise FALSE.

**Prototype:**

```
bool GetUser(TCHAR* user);
```

**Remarks:**

This method will retrieve the user name associated with a job.

**Parameters:****TCHAR\* user**

The user name which was retrieved.

**Return Value:**

TRUE if the user name was retrieved, otherwise FALSE.

**Prototype:**

**bool GetComputer(TCHAR\* computer);**

**Remarks:**

This method will retrieve the computer name associated with a job.

**Parameters:**

**TCHAR\* computer**

The computer name which was retrieved.

**Return Value:**

TRUE if the computer name was retrieved, otherwise FALSE.

**Prototype:**

**bool GetFrames(TCHAR\* frames);**

**Remarks:**

This method will retrieve the frame sequence (such as the "1,2,4,5-40" types).

**Parameters:**

**TCHAR \*frames**

The frames retrieved.

**Return Value:**

TRUE if the frame sequence string was retrieved, otherwise FALSE.

**Prototype:**

**bool GetShare(TCHAR\* share);**

**Remarks:**

This method will retrieve the Manager's network share associated with a job.

**Parameters:**

**TCHAR\* share**

The network share which was retrieved.

**Return Value:**

TRUE if the network share was retrieved, otherwise FALSE.

**Operators:****Prototype:**

**JobText& operator[](const int i);**

**Remarks:**

This access operator returns a reference to a JobText entry.

**Parameters:**

**const int i**

The index of the JobText buffer to return.

## Structure HSERVER

See Also: [Class MaxNetManager](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API as a handle to a server.

**typedef struct {**

**BYTE addr[8];**

    The handle to a server by it's mac (Ethernet) address.

**} HSERVER;**

## Structure JOBFAMES

See Also: [Class MaxNetManager](#), [Structure HSERVER](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store a job's frame progress information.

### typedef struct {

#### **char state;**

The current state of this frame which is one of the following values:

#### **FRAME\_WAITING**

The frame is waiting to be assigned and rendered.

#### **FRAME\_ASSIGNED**

The frame is assigned to the server.

#### **FRAME\_COMPLETE**

The frame has completed rendering.

#### **NO\_FRAME**

There are no frames to be rendered.

#### **int frame;**

The frame number.

#### **HSERVER hServer;**

The handle to the server rendering this frame.

#### **DWORD elapsed;**

The time it took to render this frame, in milliseconds.

### } JOBFAMES;

## Structure JobServer

See Also: [Class MaxNetManager](#), [Structure HSERVER](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about the servers in the job queue.

### typedef struct {

**HSERVER hServer;**

The handle to the server.

**char status;**

The current state of the server, which is one of the following values:

**JOB\_SRV\_IDLE**

The server is idle.

**JOB\_SRV\_BUSY**

The server is busy.

**JOB\_SRV\_FAILED**

The server is has failed (rendering error).

**JOB\_SRV\_ABSENT**

The server is absent.

**JOB\_SRV\_SUSPENDED**

The server is suspended, out of work schedule.

**JOB\_SRV\_BUSYOTHER**

The server is busy with another job.

**JOB\_SRV\_ERROR**

The server experienced a connection error.

**JOB\_SRV\_COOL\_OFF**

The server is "cooling off" (i.e. in error recovery).

**bool failed;**

This variable is used internally.

**bool active;**

Indicates that the server is active in the job.

**int cur\_frame;**

The frame which is currently being rendered.

**float thours;**

The total hours the server has spent rendering.

**int frames;**

The total number of frames the server has rendered.

**} JobServer;**

## Structure ServerList

See Also: [Class MaxNetManager](#), [Structure HSERVER](#), [Structure ServerInfo](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store global server state information.

If **hJob** is a valid handle (i.e. non 0) and **frame** holds a **NO\_FRAME**, this means this server has just be assigned a job or it is in between frames (no frames assigned). Most likely it is loading 3ds max. The transition in between frames is in the nanosecond level, but seeing as it is possible, it cannot be discarded.

### typedef struct {

**HSERVER hServer;**

The handle to the server.

**HJOB hJob;**

The handle to the job that the server is currently working on, if there is one.

**int frame;**

The frame the server is currently rendering, if there is one.

**SYSTEMTIME frame\_started;**

The time the server had the frame assigned.

**WORD state;**

The current state of the server which is one of the following values:

**SERVER\_STATE\_ABSENT**

The server is absent.

**SERVER\_STATE\_IDLE**

The server is idle.

**SERVER\_STATE\_BUSY**

The server is busy.

**SERVER\_STATE\_ERROR**

The server is experiencing an error.

**SERVER\_STATE\_SUSPENDED**

The server has been suspended..

**ServerInfo info;**

The server information structure containing the server details.

**} ServerList;**

## Structure WeekSchedule

See Also: [Class MaxNetManager](#), [Structure Schedule](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store weekly scheduling information.

**typedef struct {**

**Schedule day[7];**

The hourly schedule configuration for each day of the week.

**int AttendedPriority;**

The attended priority value which is one of the following values:

**HIGHPRIORITY, LOWPRIORITY, or IDLEPRIORITY.**

**int UnattendedPriority;**

The unattended priority value which is one of the following values:

**HIGHPRIORITY, LOWPRIORITY, or IDLEPRIORITY.**

**} WeekSchedule;**

## Structure NetworkStatus

See Also: [Class MaxNetManager](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store network status information.

**typedef struct {**

**DWORD dropped\_packets;**

The number of packets dropped due to buffer overflows.

**DWORD bad\_packets;**

The number of bad formed packets.

**DWORD tcprequests;**

The total number of TCP requests since boot time.

**DWORD udprequests;**

The total number of UDP requests since boot time.

**SYSTEMTIME boot\_time;**

The system boot time.

**char reserved[32];**

Reserved for future use.

**} NetworkStatus;**

## List of MaxNet Errors

See Also: [Class MaxNetManager](#), [Class CJobText](#), [Structure JobText](#)

The list of the various MaxNet error codes available, defined as the enum `maxnet_error_t`.

### **MAXNET\_ERR\_NONE**

No error.

### **MAXNET\_ERR\_CANCEL**

Cancellation Error.

### **MAXNET\_ERR\_NOMEMORY**

An out of memory error has occurred.

### **MAXNET\_ERR\_FILEIO**

A file IO error has occurred.

### **MAXNET\_ERR\_BADARGUMENT**

Bad arguments were passed along.

### **MAXNET\_ERR\_NOTCONNECTED**

A connection was not established.

### **MAXNET\_ERR\_NOTREADY**

Windows Network Not Installed or Not Initialized.

### **MAXNET\_ERR\_IOERROR**

An IO error has occurred.

### **MAXNET\_ERR\_CMDERROR**

A command error has occurred.

### **MAXNET\_ERR\_HOSTNOTFOUND**

The specific host could not be found.

### **MAXNET\_ERR\_BADSOCKETVERSION**

Winsock.dll is obsolete.

### **MAXNET\_ERR\_WOULDBLOCK**

An internal blocking error has occurred.

### **MAXNET\_ERR\_SOCKETLIMIT**

No more available TCP/IP Sockets.

### **MAXNET\_ERR\_CONNECTIONREFUSED**

A connection has been refused. Service not installed on host computer.

**MAXNET\_ERR\_ACCESSDENIED**

Access to a host was denied.

**MAXNET\_ERR\_TIMEOUT**

A network time-out has occurred.

**MAXNET\_ERR\_BADADDRESS**

A bad network address was supplied.

**MAXNET\_ERR\_UNKNOWN**

An unknown error has occurred.

## List of Dimension Types

See Also: [Class ParamDimension](#).

One of the following values:

**DIM\_NONE**

**DIM\_WORLD**

**DIM\_ANGLE**

**DIM\_COLOR**

**DIM\_COLOR255**

**DIM\_PERCENT**

**DIM\_NORMALIZED**

**DIM\_SEGMENTS**

**DIM\_TIME**

**DIM\_CUSTOM**

## Class ParamBlockDesc

See Also: [Class ParamBlockDescID](#).

class ParamBlockDesc

### Description:

The parameter block descriptor describes each parameter in a parameter block.

```
class ParamBlockDesc {  
    public:  
        ParamType type;  
        UserType *user;  
        BOOL animatable;  
};
```

### Data Members:

#### **ParamType type**

The parameter type. See [List of Parameter Types](#).

#### **UserType \*user**

This value is not used -- it must always be passed as NULL.

#### **BOOL animatable**

This is a flag indicating if the parameter may be animated or not. Pass TRUE if the value may be animated and FALSE if it is constant.

## Class ParamVersionDesc

### Description:

This structure describes a version of the parameter block.

### Data Members:

**ParamBlockDescID \*desc;**

This is an array of parameter block descriptors.

**int count;**

This is the number of items in the array.

**DWORD version;**

This is the version number.

### Methods:

### Prototype:

**ParamVersionDesc(ParamBlockDescID \*d,int c,int v);**

### Remarks:

Constructor.

### Parameters:

**ParamBlockDescID \*d**

This is an array of parameter block descriptors.

**int c**

This is the number of items in the array.

**int v**

This is the version number.

## Structure DispInfo

Below is the display information structure. This structure holds the information describing 3ds max's current system of measurement. This includes the type of units used, how they are displayed and any custom unit name and scale.

**typedef struct {**

**int dispType;**

Unit Display Type. One of the following values:

**UNITDISP\_GENERIC**

**UNITDISP\_METRIC**

**UNITDISP\_US**

**UNITDISP\_CUSTOM**

**int metricDisp;**

Metric display option. One of the following values:

**UNIT\_METRIC\_DISP\_MM**

**UNIT\_METRIC\_DISP\_CM**

**UNIT\_METRIC\_DISP\_M**

**UNIT\_METRIC\_DISP\_KM**

**int usDisp;**

US display option. One of the following values:

**UNIT\_US\_DISP\_FRAC\_IN**

**UNIT\_US\_DISP\_DEC\_IN**

**UNIT\_US\_DISP\_FRAC\_FT**

**UNIT\_US\_DISP\_DEC\_FT**

**UNIT\_US\_DISP\_FT\_FRAC\_IN**

**UNIT\_US\_DISP\_FT\_DEC\_IN**

**int usFrac;**

US fraction option. One of the following values:

**UNIT\_FRAC\_1\_1**

**UNIT\_FRAC\_1\_2**

**UNIT\_FRAC\_1\_4**

**UNIT\_FRAC\_1\_8**

**UNIT\_FRAC\_1\_10**

**UNIT\_FRAC\_1\_16**  
**UNIT\_FRAC\_1\_32**  
**UNIT\_FRAC\_1\_64**  
**UNIT\_FRAC\_1\_100**

**const TCHAR \*customName;**

Custom unit name.

**float customValue;**

Custom unit value.

**int customUnit;**

Custom unit reference.

Note: The three custom settings above are related as follows: If for example the **customName** is set to "FL" for furlongs (which is equal to 660 feet), **customValue** should equal 660.0 and **customUnit** should equal **UNITS\_FEET**.

**} DispInfo;**

## Class AnimProperty

See Also: [Class Animatable](#).

class AnimProperty

### Description:

This is the base class for classes that can be hung off an animatable's property list. When an animatable is deleted, its properties will be deleted and their virtual destructor will be called.

### Methods:

#### Prototype:

**virtual BOOL DontDelete();**

#### Remarks:

Implemented by the Plug-In.

When the animatable is destroyed it will normally delete all the properties. If a plug-in wants to add a property to the list that it does not want deleted it can implement this method to return TRUE.

#### Return Value:

TRUE if the item should not be deleted; otherwise FALSE.

#### Default Implementation:

**{return FALSE;}**

#### Prototype:

**virtual DWORD ID()=0;**

#### Remarks:

Implemented by the Plug-In.

Returns the ID of the property. Values above **PROPID\_USER** can be used by plug-ins. Note: that a plug-in should only put user defined properties on its own list. So IDs only have to be unique within a plug-in. If a plug-in needs to attach data to another object, it can do so via APP\_DATA.

## Class InterpCtrlUI

See Also: [Class AnimProperty](#), [Class JointParams](#).

```
class InterpCtrlUI : public AnimProperty
```

### Description:

This is simply a container class to hold some data while the controllers parameters are being edited. All methods of this class are implemented by the system.

### Data Members:

public:

**HWND hParams;**

The window handle of the rollup page.

**IObjParam \*ip;**

The interface pointer.

**Control \*cont;**

The controller that is being edited.

### Methods:

### Prototype:

**InterpCtrlUI(HWND h,IObjParam \*i,Control \*c)**

### Remarks:

Constructor. The data members are initialized to the values passed.

### Prototype:

**~InterpCtrlUI();**

### Remarks:

Destructor.

### Prototype:

**DWORD ID();**

### Remarks:

Returns the property list of id: **PROPID\_INTERPUI**

## Class MtlBaseList

See Also: [Template Class Tab](#), [Class MtlBase](#), [Class Interface](#).

```
class MtlBaseList: public Tab<MtlBaseHandle>
```

### Description:

A simple list of MtlBases. All methods of this class are implemented by the system.

Note the following typedefs:

```
typedef MtlBase* MtlBaseHandle;  
typedef Mtl* MtlHandle;  
typedef Texmap* TexmapHandle;
```

### Methods:

#### Prototype:

```
int AddMtl(MtlBase *m, BOOL checkUnique=TRUE);
```

#### Remarks:

Adds the specified MtlBase to the list.

#### Parameters:

**MtlBase \*m**

The MtlBase to add.

**BOOL checkUnique=TRUE**

If TRUE this method checks to make sure the MtlBase is unique, and will only add it if so.

#### Return Value:

Nonzero if the MtlBase was added; otherwise zero.

#### Prototype:

```
int FindMtl(MtlBase *m);
```

#### Remarks:

Finds the specified MtlBase in this material list and returns its index. Returns -1 if not found.

#### Parameters:

**MtlBase \*m**

The MtlBase to find.

**Prototype:**

**int FindMtlByName(TSTR& name);**

**Remarks:**

Finds the specified material by name and returns its index. Returns -1 if not found.

**Parameters:**

**TSTR& name**

The name to find.

**Prototype:**

**void RemoveEntry(int n);**

**Remarks:**

Removes the specified MtlBase from the list.

**Parameters:**

**int n**

The index of the MtlBase to remove.

**Prototype:**

**void Empty();**

**Remarks:**

Removes all MtlBases from the list.

## Class DWORDTab

See Also: [Class Tab](#), [Class AdjEdgeList](#).

```
class DWORDTab : public Tab<DWORD>
```

### **Description:**

This class is simply a table of DWORDs (32-bit values.)

## Class MEdge

See Also: [Class AdjEdgeList](#), [Class Mesh](#).

class MEdge

### Description:

This class describes a single edge of a mesh object that is adjacent to a vertex. This is an edge that is coming out of the vertex. This is used in adjacency lists.

### Data Members:

public:

**DWORD f[2];**

The indices into the meshes face table of the two faces that share this edge.

**DWORD v[2];**

The indices into the meshes vertex table of the two vertices of this edge.

### Methods:

public:

### Prototype:

**int EdgeIndex(Face \*faces, int side);**

### Remarks:

Returns the index of the edge in the face on side **side**. So: given a Mesh mesh and an MEdge \*me, int eid = me->EdgeIndex (mesh.faces, 0); then mesh.faces[me->f[0]].v[eid] and mesh.faces[me->f[0]].v[(eid+1)%3] are the endpoints of the edge.

In particular, mesh.edgeSel[me->f[0]\*3+eid] tells whether this edge is selected.

### Parameters:

**Face \*faces**

The list of faces from the mesh.

**int side**

Either 0 or 1, indicating whether we should find this result for the face on side 0 or on side 1.

### Prototype:

**BOOL Selected(Face \*faces, BitArray &esel);**

**Remarks:**

Returns TRUE if this edge is selected on either side; or FALSE if it is not selected on either.

**Parameters:**

**Face \*faces**

The list of faces from the mesh.

**BitArray &esel**

The edge selection BitArray from the mesh.

**Prototype:**

**BOOL Visible(Face \*faces);**

**Remarks:**

Returns TRUE if this edge is visible on either side; or FALSE if it is not visible on either.

**Parameters:**

**Face \*faces**

The list of faces from the mesh.

**Prototype:**

**BOOL Hidden(Face \*faces);**

**Remarks:**

Returns TRUE if all the faces using this edge are hidden; otherwise FALSE.

**Parameters:**

**Face \*faces**

The list of faces from the mesh.

**Prototype:**

**Point3 Center(Point3 \*verts);**

**Remarks:**

Returns the center of this edge.

**Parameters:**

**Point3 \*verts**

The list of vertices from the mesh.

**Prototype:****BOOL AFaceSelected(BitArray &fsel);****Remarks:**

Returns TRUE if one (or both) of the faces sharing the edge is selected; otherwise FALSE.

**Parameters:****BitArray &fsel**

The face selection bit array.

**Prototype:****Point3 ButterflySubdivide(Mesh \*mesh, AdjFaceList \*af, float tens);****Remarks:**

Returns a point suitable for use in standard tessellation.

**Parameters:****Mesh \*mesh**

A pointer to the associated mesh.

**AdjFaceList \*af**

A pointer to the associated AdjFaceList.

**float tens**

The tension parameter, as seen in the Tessellate modifier.

**Prototype:****UVVert ButterflyMapSubdivide (Mesh \*mesh, AdjFaceList \*af, float tens, int mp, bool & seam, UVVert & side2);****Remarks:**

This method is available in release 4.0 and later only.

This method creates a map vertex for the middle of this edge using the Butterfly tessellation scheme. Designed to create map vertices to go with the

vertex created by **ButterFlySubdivide()**.

**Parameters:**

**Mesh \*mesh**

A pointer to the associated mesh.

**AdjFaceList \*af**

A pointer to the associated AdjFaceList.

**float tens**

The tension parameter, as seen in the Tessellate modifier.

**int mp**

The map channel we want to get a result for.

**bool & seam**

If this is set to true by the algorithm, then there's a mapping seam on this edge, and different map vertices should be used on each side.

**UVVert & side2**

If there's a seam, this contains the mapping result for the second side. The return value matches the mapping scheme on face f[0]; side2 matches the scheme on face f[1].

**Return Value:**

The desired mapping coordinates.

**Prototype:**

**DWORD OtherVert(DWORD vv);**

**Remarks:**

Returns the index of the other vertex using this edge.

**Parameters:**

**DWORD vv**

The index of a vertex using the edge.

**Prototype:**

**DWORD OtherFace(DWORD ff);**

**Remarks:**

Returns the index of the other face using this edge.

**Parameters:**

**DWORD ff**

The index of a face using this edge.

## Class AdjFace

See Also: [Class AdjFaceList](#).

class AdjFace

### Description:

This class describes a single face for use in AdjFaceLists.

### Data Members:

public:

**DWORD f[3];**

The indices of the faces adjacent to this one. These are indices into the mesh face table.

### Methods:

### Prototype:

**AdjFace();**

### Remarks:

Constructor. The face indices are set to **UNDEFINED**.

## List of NURBSObject Types

See Also: [Class NURBSObject](#), [Class NURBSSurface](#), [Class NURBSControlVertex](#), [Class NURBSPoint](#), [Class NURBSCurve](#).

One of the following enum values describes the type of **NURBSObject** this is. The sub-classed **NURBSObject** returns one of these values in **GetType()**.

### **kNPoint**

Specifies a [NURBSIndependentPoint](#) object.

### **kNPointCPoint**

Specifies a [NURBSPointConstPoint](#) object.

### **kNCurveCPoint**

Specifies a [NURBSCurveConstPoint](#) object.

### **kNCurveCurveIntersectionPoint**

Specifies a [NURBSCurveCurveIntersectionPoint](#) object.

### **kNSurfaceCPoint**

Specifies a [NURBSSurfConstPoint](#) object.

### **kNCurveSurfaceIntersectionPoint**

Specifies a [NURBSCurveSurfaceIntersectionPoint](#) object.

### **kNTexturePoint**

This option is available in release 3.0 and later only.

Specifies a [NURBSTexturePoint](#) object.

### **kNVCV**

Specifies a [NURBSControlVertex](#) object.

### **kNVCVCurve**

Specifies a [NURBSCVCurve](#) object.

### **kNPointCurve**

Specifies a [NURBSPointCurve](#) object.

### **kNBlendCurve**

Specifies a [NURBSBlendCurve](#) object.

### **kNOffsetCurve**

Specifies a [NURBSOffsetCurve](#) object.

### **kNXFormCurve**

Specifies a [NURBSXFormCurve](#) object.

**kNMirrorCurve**

Specifies a [NURBSMirrorCurve](#) object.

**kNFilletCurve**

Specifies a [NURBSFilletCurve](#) object.

**kNChamferCurve**

Specifies a [NURBSChamferCurve](#) object.

**kNIsoCurve**

Specifies a [NURBSIsoCurve](#) object.

**kNProjectVectorCurve**

Specifies a [NURBSProjectVectorCurve](#) object.

**kNProjectNormalCurve**

Specifies a [NURBSProjectNormalCurve](#) object.

**kNSurfSurfIntersectionCurve**

Specifies a [NURBSSurfSurfIntersectionCurve](#) object.

**kNCurveOnSurface**

Specifies a [NURBSCurveOnSurface](#) object.

**kNPointCurveOnSurface**

Specifies a [NURBSPointCurveOnSurface](#) object.

**kNSurfaceNormalCurve**

Specifies a [NURBSSurfaceNormalCurve](#) object.

**kNSurfaceEdgeCurve**

This option is available in release 3.0 and later only.

Specifies a [NURBSSurfaceEdgeCurve](#) object.

**kNCVSurface**

Specifies a [NURBSCVSurface](#) object.

**kNPointSurface**

Specifies a [NURBSPointSurface](#) object.

**kNBlendSurface**

Specifies a [NURBSBlendSurface](#) object.

**kNOffsetSurface**

Specifies a [NURBSOffsetSurface](#) object.

**kNXFormSurface**

Specifies a [NURBSXFormSurface](#) object.

**kNMirrorSurface**

Specifies a [NURBSMirrorSurface](#) object.

**kNRuledSurface**

Specifies a [NURBSRuledSurface](#) object.

**kNULoftSurface**

Specifies a [NURBSULoftSurface](#) object.

**kNExtrudeSurface**

Specifies a [NURBSExtrudeSurface](#) object.

**kNLatheSurface**

Specifies a [NURBSLatheSurface](#) object.

**kNUVLoftSurface**

Specifies a [NURBSUVLoftSurface](#) object.

**kNNBlendSurface**

Specifies a [NURBSNBlendSurface](#) object.

**kN1RailSweepSurface**

Specifies a [NURBS1RailSweepSurface](#) object.

**kN2RailSweepSurface**

Specifies a [NURBS2RailSweepSurface](#) object.

**kNCapSurface**

Specifies a [NURBSCapSurface](#) object.

**kNMultiCurveTrimSurface**

Specifies a [NURBSMultiCurveTrimSurface](#) object.

**kNFilletSurface**

This option is available in release 3.0 and later only.

Specifies a [NURBSFilletSurface](#) object.

## List of NURBSObject Kinds

See Also: [Class NURBSObject](#).

One of the following enum values describes the kind of **NURBSObject** this is (sub-classed from which base class). The sub-classed **NURBSObject** returns one of these values in **GetKind()**.

### **kNURBSPoint**

Specifies a [NURBSPoint](#) object.

### **kNURBSTexturePoint**

This option is available in release 3.0 and later only.

Specifies a [NURBSTexturePoint](#) object.

### **kNURBSCV**

Specifies a [NURBSControlVertex](#) object.

### **kNURBSCurve**

Specifies a [NURBSCurve](#) object.

### **kNURBSSurface**

Specifies a [NURBSSurface](#) object.

## List of NURBS Trim Directions

See Also: [Class NURBSTrimPoint](#).

One of the following enum values describes a direction for a trim to the curve. This defines the side of the curve to keep relative to the trim point.

### **kNone = 0**

Specifies the curve on both sides of the point should be kept -- no trimming is done.

### **kPositive = 1**

Specifies the curve on the positive side of the curve should be kept. The positive side is the side between the point towards greater values in the curve parameter space.

### **kNegative = 2**

Specifies the curve on the negative side of the curve should be kept. The negative side is the side between the point towards lesser values in the curve parameter space.

## List of NURBSConst Types

See Also: [Class NURBSPointConstPoint](#), [Class NURBSCurveConstPoint](#), [Class NURBSSurfConstPoint](#).

One of the following enum values describes the type of **NURBSConst** this is. For example, these correspond to the types of dependent points available in the NURBS Surface user interface in the 'Surface Point' rollup.

Constrained objects are those that exist in the parameter space of other objects.

### **kNConstOnObject**

Indicates the point is actually on the surface of the object.

### **kNConstOffset**

Indicates the points is offset some distance (specified in object space) from the surface of the object.

### **kNConstNormal**

Indicates the point is offset some distance along the normal to the curve or surface.

### **kNConstTangent**

Indicates the point is offset some U and/or V distance along the tangent from the curve or surface. If the value is positive it's the tangent that heads in the direction of increasing parameter value; if negative it's the tangent that heads in the direction of decreasing parameter value.

## List of NURBSAutoParam Types

See Also: [Class NURBSCVCurve](#), [Class NURBSCVSurface](#).

One of the following enum values describes how automatic reparameterization is handled.

### **kNotAutomatic**

Specifies that the reparameterization is **not** automatic. When it is automatic, curves are reparameterized as they are edited.

### **kAutoCentripetal**

Chooses the chord-length algorithm for reparameterization. Chord-length reparameterization spaces knots (in parameter space) based on the square root of the length of each curve segment.

### **kAutoUniform**

Spaces the knots uniformly. A uniform knot vector has the advantage that the curve or surface changes only locally when you edit it. With chord-length parameterization, moving any CV can potentially change the entire sub-object.

## List of NURBSParamaterization Types

See Also: [Class NURBSCVCurve](#), [Class NURBSCVSurface](#).

One of the following enum values describes the types or reparameterization.

### **kCentripetal**

Chooses the chord-length algorithm for reparameterization. Chord-length reparameterization spaces knots (in parameter space) based on the square root of the length of each curve segment.

### **kUniform**

Spaces the knots uniformly. A uniform knot vector has the advantage that the curve or surface changes only locally when you edit it. With chord-length parameterization, moving any CV can potentially change the entire sub-object.

## List of NURBSMirrorAxis Types

See Also: [Class NURBSMirrorCurve](#), [Class NURBSMirrorSurface](#).

One of the following enum values describe the axis of reflection for a mirror curve or surface.

**kMirrorX**

**kMirrorY**

**kMirrorZ**

**kMirrorXY**

**kMirrorXZ**

**kMirrorYZ**

## List of NURBS Texture Surface Types

See Also: [Class NURBSTextureSurface](#), [Class NURBSControlVertex](#), [Class NURBSSurface](#).

Determines the type of texture surface generated. One of the following values:

### **kNMapDefault**

Automatically generates a texture surface. This method evenly distributes the texture, and attempts to compensate for stretching of the surface.

### **kNMapUserDefined**

Generates a texture surface that the user can edit. A user may edit the user-defined texture surface either by using an Edit Texture Surface dialog (as in versions of 3ds max prior to release 3), or by editing texture points directly in the viewports.

### **kNMapSurfaceMapper**

Generates the texture surface by projecting the texture of another NURBS surface sub-object in the NURBS model. The projection travels along the direction of the normals of the source surface. Projected texture surfaces are relational.

## Class PatchCapInfo

See Also: [Working with Shapes and Splines](#).

class PatchCapInfo

### Description:

This is the information class for patch capping. All methods of this class are implemented by the system. Developers must only declare an instance of this class and then call **MakeCap()** on the shape.

### Data Members:

public:

**CapPatchTab patches;**

This is used internally.

**PatchCapVertTab verts;**

This is used internally.

**PatchCapVecTab vecs;**

This is used internally.

**Point3Tab newVerts;**

This is used internally.

**Point3Tab newVecs;**

This is used internally.

### Methods:

#### Prototype:

**void Init(BezierShape \*shape);**

#### Remarks:

This method is used internally.

#### Prototype:

**void FreeAll();**

#### Remarks:

This method is used internally.

### Operators:

**Prototype:**

**PatchCapInfo &operator=(PatchCapInfo &from)**

**Remarks:**

Assignment operator used internally.

## Class ShapeVSel

See Also: [Class BitArray](#), [Class PolyShape](#).

class ShapeVSel

### Description:

This class stores and provides access to shape vertex selection data. All methods of this class are implemented by the system.

### Data Members:

public:

**int polys;**

The number of splines in the shape.

**BitArray \*sel;**

An array of BitArrays, one for each spline.

### Methods:

### Prototype:

**ShapeVSel();**

### Remarks:

Constructor. The number of splines is set to 0. The BitArray pointer is set to NULL.

### Prototype:

**void Insert(int where, int count=0);**

### Remarks:

Creates and inserts a new BitArray into **sel**.

### Parameters:

**int where**

The index into **sel** indicating where to insert the new BitArray.

**int count=0**

The number of bits in the new BitArray.

### Prototype:

**void Delete(int where);**

**Remarks:**

Deletes the specified BitArray from the **sel** list.

**Parameters:**

**int where**

The index into **sel** indicating which BitArray to delete.

**Prototype:**

**void SetSize(BezierShape& shape, BOOL save=FALSE);**

**Remarks:**

Sets the number of splines and allocates the corresponding number of BitArrays based on the shape passed. The size of each BitArray is set to the number of vertices in each polyline.

**Parameters:**

**BezierShape& shape**

The shape whose splines determine the sizes set.

**BOOL save=FALSE**

TRUE to keep the previous BitArray contents. FALSE to discard it.

**Prototype:**

**void SetSize(PolyShape& shape, BOOL save=FALSE);**

**Remarks:**

Sets the number of splines and allocates the corresponding number of BitArrays based on the shape passed. The size of each BitArray is set to the number of vertices in each spline.

**Parameters:**

**PolyShape& shape**

The shape whose lines determine the sizes set.

**BOOL save=FALSE**

TRUE to keep the previous BitArray contents. FALSE to discard it.

**Prototype:**

**void ClearAll();**

**Remarks:**

Clears every bit for every poly.

**Prototype:**

**void Empty();**

**Remarks:**

Sets the size of every poly BitArray to 0.

**Prototype:**

**IOResult Save(ISave\* isave);**

**Remarks:**

Saves the ShapeVSEL to disk.

**Prototype:**

**IOResult Load(ILoad\* iload);**

**Remarks:**

Loads the ShapeVSEL from disk.

**Operators:**

**Prototype:**

**ShapeVSEL& operator=(ShapeVSEL& from);**

**Remarks:**

Assignment operator.

## Class ShapeSSel

See Also: [Class BitArray](#), [Class PolyShape](#).

class ShapeSSel

### Description:

This class stores and provides access to shape segment selection data. All methods of this class are implemented by the system.

### Data Members:

public:

**int polys;**

The number of splines in the shape.

**BitArray \*sel;**

An array of BitArrays, one for each spline.

### Methods:

#### Prototype:

**ShapeSSel();**

#### Remarks:

Constructor. Initialize the class members.

**Constructor. The number of splines is set to 0. The BitArray pointer is set to NULL.**Prototype:

**~ShapeSSel();**

#### Remarks:

Destructor. Any BitArrays are freed.

#### Prototype:

**void Insert(int where, int count=0);**

#### Remarks:

Creates and inserts a new BitArray into **sel**.

#### Parameters:

**int where**

The index into **sel** indicating where to insert the new BitArray.

**int count=0**

The number of bits in the new BitArray.

**Prototype:**

**void Delete(int where);**

**Remarks:**

Deletes the specified BitArray from the **sel** list.

**Parameters:**

**int where**

The index into **sel** indicating which BitArray to delete.

**Prototype:**

**void SetSize(BezierShape& shape, BOOL save=FALSE);**

**Remarks:**

Sets the number of splines and allocates the corresponding number of BitArrays based on the shape passed. The size of each BitArray is set to the number of segments in each polyline.

**Parameters:**

**BezierShape& shape**

The shape whose splines determine the sizes set.

**BOOL save=FALSE**

TRUE to keep the previous BitArray contents. FALSE to discard it.

**Prototype:**

**void SetSize(PolyShape& shape, BOOL save=FALSE);**

**Remarks:**

Sets the number of splines and allocates the corresponding number of BitArrays based on the shape passed. The size of each BitArray is set to the number of segments in each spline.

**Parameters:**

**PolyShape& shape**

The shape whose lines determine the sizes set.

**BOOL save=FALSE**

TRUE to keep the previous BitArray contents. FALSE to discard it.

**Prototype:**

**void ClearAll();**

**Remarks:**

Clears every bit for every poly.

**Prototype:**

**void Empty();**

**Remarks:**

Sets the size of every poly BitArray to 0.

**Prototype:**

**IOResult Save(ISave\* isave);**

**Remarks:**

Saves the ShapeSSel to disk.

**Prototype:**

**IOResult Load(ILoad\* iload);**

**Remarks:**

Loads the ShapeSSel from disk.

**Operators:**

**Prototype:**

**ShapeSSel& operator=(ShapeSSel& from);**

**Remarks:**

Assignment operator.

## Class ShapePSel

See Also: [Class BitArray](#), [Class PolyShape](#).

class ShapePSel

### Description:

This class stores and provides access to shape polygon (spline) selection data. All methods of this class are implemented by the system.

### Data Members:

public:

**int polys;**

The number of splines in the shape.

**BitArray sel;**

One bit for each spline in the shape.

### Methods:

#### Prototype:

**ShapePSel();**

#### Remarks:

Constructor. Initialize the class members.

#### Prototype:

**~ShapePSel();**

#### Remarks:

Destructor.

#### Prototype:

**void Insert(int where);**

#### Remarks:

Resizes the BitArray **sel** to include a new bit at the specified location.

#### Parameters:

**int where**

The location for the new bit in the BitArray.

**Prototype:**

**void Delete(int where);**

**Remarks:**

Deletes the specified bit from the BitArray.

**Parameters:**

**int where**

Indicates which bit to delete.

**Prototype:**

**void SetSize(BezierShape& shape, BOOL save=FALSE);**

**Remarks:**

Sets the number of splines and resizes the BitArray based on the shape passed.

**Parameters:**

**BezierShape& shape**

The shape whose splines determine the sizes set.

**BOOL save=FALSE**

TRUE to keep the previous BitArray contents. FALSE to discard it.

**Prototype:**

**void SetSize(PolyShape& shape, BOOL save=FALSE);**

**Remarks:**

Sets the number of splines and resizes the BitArray based on the shape passed.

**Parameters:**

**PolyShape& shape**

The shape whose lines determine the sizes set.

**BOOL save=FALSE**

TRUE to keep the previous BitArray contents. FALSE to discard it.

**Prototype:**

**void Set(int index);**

**Remarks:**

Sets the bit specified by the index to 1.

**Parameters:**

**int index**

The bit to set.

**Prototype:**

**void Set(int index, int value);**

**Remarks:**

Sets the bit specified by the index to the value passed.

**Parameters:**

**int index**

The bit to set or clear.

**int value**

The value to set, either 0 or 1.

**Prototype:**

**void Clear(int index);**

**Remarks:**

Clears the bit specified by the index to 1.

**Parameters:**

**int index**

The bit to clear.

**Prototype:**

**void ClearAll();**

**Remarks:**

Clears all the bits in the array (sets them to 0).

**Prototype:**

**void Empty();**

**Remarks:**

Sets the size of **sel** to 0.

**Prototype:**

**IOResult Save(ISave\* isave);**

**Remarks:**

Saves the BitArray to disk.

**Prototype:**

**IOResult Load(ILoad\* iload);**

**Remarks:**

Loads the BitArray from disk.

**Operators:**

**Prototype:**

**ShapePSel& operator=(ShapePSel& from);**

**Remarks:**

Assignment operator.

**Prototype:**

**int operator[](int index) const;**

**Remarks:**

Array access operator.

## List of BezierShape Display Flags

See Also: [Class BezierShape](#).

One or more of the following values:

### **DISP\_VERTTICKS**

Display vertices as tick marks.

### **DISP\_BEZHANGLES**

Display bezier handles.

### **DISP\_SELVERTS**

Display selected vertices.

### **DISP\_SELSEGMENTS**

Display selected segments.

### **DISP\_SELPOLYS**

Display selected polygons.

### **DISP\_UNSELECTED**

Used by the lofter. Indicates the shape is unselected. The shape is drawn in white, overriding any colors that the BezierShape class would have used.

### **DISP\_SELECTED**

Used by the lofter. Indicate the shape is selected. The shape is drawn using the selection color, overriding any colors that the BezierShape class would have used.

### **DISP\_ATSHAPELEVEL**

Used by the lofter. Indicates the shape is at the current level. The shape is drawn in green, overriding any colors that the BezierShape class would have used.

### **DISP\_VERT\_NUMBERS**

This flag is available in release 3.0 and later only.

When this bit is set, and vertex ticks are being displayed, the shape is drawn with vertex numbers in addition to the ticks.

### **DISP\_VERT\_NUMBERS\_SELONLY**

This flag is available in release 3.0 and later only.

When this bit is set and the **DISP\_VERT\_NUMBERS** bit is set, only the numbers of selected vertices are displayed.

### **DISP\_SPLINES\_ORTHO**

This bit is reserved for internal use.

## Class MeshCapInfo

See Also: [Working with Shapes and Splines](#).

class MeshCapInfo

### Description:

The information class for mesh capping (MORPH or GRID). All the data members and methods of this class are used internally. Developers must only declare an instance of this class and then call **MakeCap()** on the shape.

### Data Members:

public:

**CapFaceTab faces;**

This is used internally.

**MeshCapVertTab verts;**

This is used internally.

**Point3Tab newVerts;**

This is used internally.

### Methods:

#### Prototype:

**void Init(PolyShape \*shape);**

#### Remarks:

This method is used internally.

#### Prototype:

**void FreeAll();**

#### Remarks:

This method is used internally.

### Operators:

#### Prototype:

**MeshCapInfo &operator=(MeshCapInfo &from);**

#### Remarks:

Assignment operator used internally.

## Class BezierShapeTopology

See Also: [Class BezierShape](#), [Template Class Tab](#), [Class BitArray](#).

class BezierShapeTopology

### Description:

This class is available in release 3.0 and later only.

This class has data members and methods used to build and store topology information on a BezierShape. This class is used with the

**BezierShape::GetTopology()** method.

### Data Members:

public:

#### **BOOL ready;**

TRUE if the data has been built; otherwise FALSE.

#### **IntTab kcount;**

A table of integers containing the knot count for each bezier spline in the shape.

#### **BitArray closed;**

A bit array containing a 1 for each closed spline or a 0 for each open one in the shape.

### Methods:

public:

### Prototype:

**BezierShapeTopology();**

### Remarks:

Constructor. The data members are initialized as follows:

**ready = FALSE;**

### Prototype:

**void Build(BezierShape &shape);**

### Remarks:

Builds the topology data for the specified shape.

**Parameters:**

**BezierShape &shape**

The shape whose topology data will be built.

**Prototype:**

**IOResult Save(ISave \*isave);**

**Remarks:**

Used internally to save the shape topology data.

**Prototype:**

**IOResult Load(ILoad \*iload);**

**Remarks:**

Used internally to load the shape topology data.

**Prototype:**

**int operator==(const BezierShapeTopology& t);**

**Remarks:**

Assignment operator.

## Class ShapeHitData

See Also: [Class HitData](#).

```
class ShapeHitData : public HitData
```

### Description:

This is a storage class for hit records used in hit testing to know which specific shape object was hit. All methods of this class are implemented by the system.

### Data Members:

public:

**BezierShape \*shape;**

The shape that was hit.

**int poly;**

The polygon of the shape that was hit.

**int index;**

The index of the sub-object entity that was hit.

### Methods:

### Prototype:

**ShapeHitData(BezierShape \*shape, int poly, int index)**

### Remarks:

Constructor. The data members are initialized to the values passed.

## Class ShapeContextCallback

See Also: [Class BezierShape](#), [Class ModContext](#).

class ShapeContextCallback

### Description:

This class is available in release 3.0 and later only.

This class has a method used for retrieving other shapes in the current editing context. This class provides a way for the

**BezierShape::PerformTrimOrExtend** method to access the shapes being trimmed.

### Methods:

public:

### Prototype:

```
virtual BezierShape *GetShapeContext(ModContext *context)=0;
```

### Remarks:

This method will be called with a ModContext pointer; the function should return the shape for that context. This is only used in modifier applications, where more than one shape object is being modified. See **\MAXSDK\SAMPLES\MODIFIERS\EDITSP.L.CPP** for an example of its use.

### Parameters:

**ModContext \*context**

Points to the ModContext for the shape the modifier is applied to.

### Return Value:

A pointer to the BezierShape for the context.

## Class PolyPt

See Also: [Class PolyLine](#).

class PolyPt

### Description:

This class represents a single point of a **PolyLine**. All methods of this class are implemented by the system.

### Data Members:

public:

#### **Point3 p;**

The location of the point.

#### **DWORD flags;**

Predefined **PolyPt** flags. Note that developers can use bits 0-7 of the flags for their own use. Bits 8 and above are used by the system.

If you are converting some other type of shape or spline to a **PolyShape** (and thus **PolyLines**) you can set certain flags to make things operate smoother.

These flags are described below:

#### **POLYPT\_KNOT**

This indicates if this point in the **PolyLine** corresponds to a knot in the original spline. For example if you had a circle that was a bezier spline it should have four of the points in the **PolyLine** designated as **POLYPT\_KNOT**. These are the points at the 12 o'clock, 3 o'clock, 6 o'clock and 9 o'clock positions. Then all the other points would be **POLYPT\_INTERPOLATED**. This is used to make capping more efficient, for example, the system generally tries to attach to a knot when making connections between polygons.

#### **POLYPT\_INTERPOLATED**

This indicates the point is an interpolated point (not a knot).

#### **POLYPT\_SMOOTH**

If you convert to a **PolyLine**, use this bit to control smoothing of the resulting shape. If this bit is set, it means that any mesh generated will share smoothing across the edge. For example, all the points on a curved section of a spline between knots will have this flag set. Then depending on the

knot type (if it's a corner or bezier corner) then this smooth flag will not be set. If this shape is then extruded or lofted this information is used to determine smoothing groups. If this flag is set you'll get a smooth transition however if this bit is not set you'll get a nice sharp corner.

### **POLYPT\_SEG\_SELECTED**

The segment that starts with this point is selected. This is used in the drawing routines so that PolyShapes generated from BezierShapes will still have the selected segments drawn in the selected color.

### **POLYPT\_BRIDGE**

This flag is used internally by the capping code.

### **POLYPT\_SPLICE**

This flag is used internally by the capping code.

### **POLYPT\_VISEEDGE**

This flag is used internally by the capping code.

### **POLYPT\_NO\_SPLICE**

This flag is used internally by the capping code.

### **int aux;**

Auxiliary data attached to this point (usually a mesh vertex number for capping).

### **DWORD flags2;**

This flag contains the material information for the segments. The mat ID is stored in the HIWORD.

```
#define POLYPT_MATID_SHIFT 16
```

```
#define POLYPT_MATID_MASK 0xFFFF
```

### **Methods:**

### **Prototype:**

```
PolyPt();
```

### **Remarks:**

Constructor. The point **p** is set to 0,0,0, **flags** and **flags2** are set to 0, **aux** is set to 0.

### **Prototype:**

**PolyPt(Point3 ip, DWORD f = 0, int a=0, DWORD f2=0);**

**Remarks:**

Constructor. The point, flags, aux and flags2 data members are initialized to the data passed.

**Prototype:**

**inline MtlID GetMatID();**

**Remarks:**

This method is available in release 3.0 and later only.

This method gets the material ID on a per-segment basis within the spline or polyline.

**Prototype:**

**inline void SetMatID(MtlID id);**

**Remarks:**

This method is available in release 3.0 and later only.

This method sets the material ID on a per-segment basis within the spline or polyline.

**Parameters:**

**MtlID id**

The ID to set.

## List of Parameter Types for PolyLine Interpolation

See Also: [Class PolyLine](#).

One of the following values:

### **POLYSHP\_INTERP\_SIMPLE**

Parameter space based on segments. This simple interpolation is interpolating based on parameter space -- If a polyline has 4 segments, the first segment is parameter values 0-0.25, the second 0.25-0.5, the third 0.5-0.75 and the fourth 0.75-1.0. This is regardless of the length of each segment.

### **POLYSHP\_INTERP\_NORMALIZED**

Parameter space normalized to curve length. This interpolation normalizes the parameter space to distance along the length of a polyline. So parameter space 0 is the start, 1.0 is the end and 0.5 is halfway along the actual length of the curve.

# Class EventRouter

See Also: [Class EventUser](#).

class EventRouter

## Description:

Event router functionality. All methods of this class are implemented by the system.

## Methods:

### Prototype:

**void Register(EventUser \*user);**

### Remarks:

Register and activate an event user.

### Parameters:

**EventUser \*user**

The EventUser to activate.

### Prototype:

**void UnRegister(EventUser \*user);**

### Remarks:

Remove an event user from the list (automatically re-activates the previous user).

### Parameters:

**EventUser \*user**

The EventUser to remove.

### Prototype:

**BOOL Process();**

### Remarks:

Process the event.

### Return Value:

TRUE if the event was handed off to a user.

# Class ArcballCallback

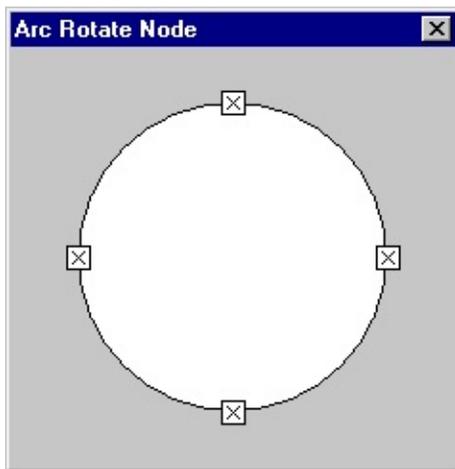
See Also: [Class ArcballDialog](#), [Class Quat](#).

class ArcballCallback

## Description:

This class is available in release 2.0 and later only.

This class provides methods to work with a general arcball dialog box for doing 3D rotations. This dialog appears below:



To use these APIs you'll need to **#include "arcdlg.h"**

All methods of this class are implemented by the plug-in.

Sample code can be found in

**\\MAXSDK\\SAMPLES\\HOWTO\\CUSTCTRL\\CUSTCTRL.CPP.**

## Function:

```
ArcballDialog *CreateArcballDialog(ArcballCallback *cb, HWND  
hWndOwner, TCHAR* title=NULL);
```

## Remarks:

This global function is provided by 3ds max and is used to create the arcball dialog box. Then the methods of your callback class are called based on the user's use of the dialog. Be sure to call the **ArcballDialog::DeleteThis()** method when done.

## Parameters:

**ArcballCallback \*cb**

The callback whose methods are called based on the user's interaction with the

dialog controls.

**HWND hwndOwner**

The window handle of the dialog owner.

**TCHAR\* title=NULL**

The title string to be displayed in the dialog.

**Return Value:**

A new instance of the **ArcballDialog** class. Be sure to call its **DeleteThis()** method when done.

**Methods:**

**Prototype:**

**virtual void StartDrag()=0;**

**Remarks:**

This method is called when a drag operation begins. The developer may want to save the start state at this point.

**Prototype:**

**virtual void EndDrag()=0;**

**Remarks:**

This method is called when a drag operation ends.

**Prototype:**

**virtual void Drag(Quat q, BOOL buttonUp)=0;**

**Remarks:**

This method is called during a drag operation.

**Parameters:**

**Quat q**

The relative rotation from the start rotation.

**BOOL buttonUp**

If TRUE this indicates if the mouse button is up (has been released); if FALSE the button is down.

**Prototype:**

**virtual void CancelDrag()=0;**

**Remarks:**

This method is called when the right mouse button is clicked during a drag operation to cancel it.

**Prototype:**

**virtual void BeingDestroyed()=0;**

**Remarks:**

This method is called if the dialog box window was closed. Note that developers should **not** call **ArcballDialog::DeleteThis()** from inside this method.

## Class IMergeManager

See Also: [Class InterfaceServer](#), [List of Reference Messages](#) , [Class ExclList](#)

```
class IMergeManager : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This class facilitates taking care of node handles when merging and xref'ing scenes.

When nodes are merged, their handles will be reassigned so that their handles will not conflict with existing nodes in the scene. After the merge process has completed, all merged objects will receive a reference notification;

**REFMSG\_NODE\_HANDLE\_CHANGED**. The PartID will be a pointer to a merge manager interface that you can use to see if a specific handle was converted and convert between the old and the new handle. **IMergeManager\* pMergeManager = (IMergeManager\*)partID**; If a plug-in uses node handles in a persistent manner it has to intercept this reference message and convert the previous handles to the newly generated handles.

### Methods:

public:

### Prototype:

```
virtual ULONG GetNewHandle(ULONG oldHandle) = 0;
```

### Remarks:

This method allows you to obtain a new handle.

### Parameters:

**ULONG oldHandle**

The old handle you wish to obtain a new one for.

### Return Value:

The new handle.

### Prototype:

**virtual bool HandleExist(ULONG handle) = 0;**

**Remarks:**

This method allows you to check if a handle already exists.

**Parameters:**

**ULONG handle**

The handle you wish to check for.

**Return Value:**

TRUE if the handle exists; otherwise FALSE.

## Class DllDesc

See Also: [Class DllDir](#), [Class ClassDesc](#), [Class Interface](#), [DLL Functions and Class Descriptors](#).

class DllDesc

### Description:

This class is available in release 2.0 and later only.

This class provides information about a DLL. Every DLL in 3ds max may implement any number of classes. This class has methods which return the number of classes implemented in the DLL, a description string for the DLL, and a method to unmap the DLL from memory.

All methods of this class are implemented by the system.

### Data Members:

public:

**HINSTANCE handle;**

The DLL instance handle.

**TSTR fname;**

The file name the DLL was loaded from.

### Methods:

#### Prototype:

**const TCHAR \*Description();**

#### Remarks:

Returns the Description defined in the **LibDescription()** function.

#### Prototype:

**int NumberOfClasses();**

#### Remarks:

Returns the number of classes implemented in the DLL. This is the value returned by the library function **LibNumberClasses()**.

#### Prototype:

**void Free();**

**Remarks:**

This method decrements the reference count of the DLL. When the reference count reaches zero, the module is unmapped from the address space of the calling process.

**Operators:**

**Prototype:**

**ClassDesc \*operator[](int i);**

**Remarks:**

Returns a pointer to the 'i-th' class descriptor.

**Parameters:**

**int i**

The zero based index of the ClassDesc to return.

**Prototype:**

**int operator==(const DllDesc& dd) const;**

**Remarks:**

Equality operator. Returns nonzero if the **DllDesc** passed matches this one; otherwise zero.

## List of System Error Log Message Types

See Also: [Class LogSys](#).

There are four types of log entries. In the preference dialog, the user can select what types of log entries they want to be generated. This is the way a user can control the verbosity of the log file in some meaningful way. Developers are encouraged to be very verbose about information and debug messages -- the **Log()** functions should be used to record any events out of the ordinary. As the user can elect to ignore these messages they are perfect for troubleshooting.

One or more of the following values (these values may be ORed together ( i.e. **SYSLOG\_ERROR|SYSLOG\_WARN**):

### **SYSLOG\_ERROR**

An error message. An example of this type is a fatal error.

### **SYSLOG\_WARN**

A warning message. An example of this type is a message telling the user the MAX file just loaded is obsolete and needs to be resaved. This option may not be selected by the MAX user via the UI but it is available for use (it's used internally often).

### **SYSLOG\_INFO**

An information message. An example of this is a message indicating a new MAX file has been loaded.

### **SYSLOG\_DEBUG**

A debugging message. This message type is for anything you think might help trace problems that the user is having with your program.

## Class TextureInfo

See Also: [Class MtlMakerCallback](#), [Class Material](#), [Class Matrix3](#).

```
class TextureInfo : public BaseInterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This class describes a texture used by the interactive renderer. This includes all the information about the mapping channel, tiling, etc. A table of these is maintained by the **texture** data member of class **Material**.

There are data members related to maps which specify how the texture should be applied. These are specified independently for color and alpha and include a scale. For example, for normal multiplication (modulation) application of a texture, the entries would be:

```
colorOp = GW_TEX_MODULATE  
colorAlphaSource = GW_TEX_TEXTURE  
colorScale = GW_TEX_SCALE_1X  
alphaOp = GW_TEX_MODULATE  
alphaAlphaSource = GW_TEX_TEXTURE  
alphaScale = GW_TEX_SCALE_1X
```

For applying a texture with alpha blending, the entry would be:

```
colorOp = GW_TEX_ALPHA_BLEND  
colorAlphaSource = GW_TEX_TEXTURE  
colorScale = GW_TEX_SCALE_1X  
alphaOp = GW_TEX_LEAVE  
alphaAlphaSource = GW_TEX_TEXTURE  
alphaScale = GW_TEX_SCALE_1X
```

For applying an opacity map, the entry would be:

```
colorOp = GW_TEX_LEAVE  
colorAlphaSource = GW_TEX_TEXTURE  
colorScale = GW_TEX_SCALE_1X  
alphaOp = GW_TEX_REPLACE  
alphaAlphaSource = GW_TEX_TEXTURE  
alphaScale = GW_TEX_SCALE_1X
```

## Data Members:

public:

**int useTex;**

Indicates if the material uses textures. Nonzero indicates textures are used.

**int faceMap;**

Indicates if the material is face mapped. Nonzero indicates it is.

**DWORD\_PTR textHandle;**

The texture handle.

**int uvwSource;**

The UVW source used.

**UVSOURCE\_MESH**

Use UVW coordinates from a standard map channel.

**UVSOURCE\_XYZ**

Compute UVW from object XYZ.

**UVSOURCE\_MESH2**

Use UVW2 (Vertex Color) coordinates.

**UVSOURCE\_WORLDXYZ**

Use World XYZ as UVW.

**int mapChannel;**

The mapping channel used.

**Matrix3 textTM;**

The texture transformation matrix.

**UBYTE tiling[3];**

The UVW tiling. One of the following values:

**GW\_TEX\_REPEAT**

**GW\_TEX\_MIRROR**

**GW\_TEX\_NO\_TILING**

**UBYTE colorOp;**

The color texture operation. One of the following values:

**GW\_TEX\_LEAVE**

Use the source pixel value

**GW\_TEX\_REPLACE**

Use the texture pixel value

**GW\_TEX\_MODULATE**

Multiply the source with the texture

**GW\_TEX\_ADD**

Add the source and texture

**GW\_TEX\_ADD\_SIGNED**

Add the source and texture with an 0.5 subtraction

**GW\_TEX\_SUBTRACT**

Subtract the source from the texture

**GW\_TEX\_ADD\_SMOOTH**

Add the source and the texture then subtract their product

**GW\_TEX\_ALPHA\_BLEND**

Alpha blend the texture with the source

**GW\_TEX\_PREMULT\_ALPHA\_BLEND**

Alpha blend the source with a premultiplied alpha

**UBYTE colorAlphaSource;**

The color blend alpha source. One of the following values:

**GW\_TEX\_ZERO**

Use no alpha value

**GW\_TEX\_SOURCE**

Use the source alpha

**GW\_TEX\_TEXTURE**

Use the texture alpha

**GW\_TEX\_CONSTANT**

Use a constant BGRA color as an alpha

**GW\_TEX\_PREVIOUS**

Use the previous texture stage alpha

**UBYTE colorScale;**

The color scale factor. One of the following values:

**GW\_TEX\_SCALE\_1X**

Multiply the tex op result by 1

**GW\_TEX\_SCALE\_2X**

Multiply the tex op result by 2

**GW\_TEX\_SCALE\_4X**

Multiply the tex op result by 4

### **UBYTE alphaOp;**

The alpha texture operation. One of the following values:

#### **GW\_TEX\_LEAVE**

Use the source pixel value

#### **GW\_TEX\_REPLACE**

Use the texture pixel value

#### **GW\_TEX\_MODULATE**

Multiply the source with the texture

#### **GW\_TEX\_ADD**

Add the source and texture

#### **GW\_TEX\_ADD\_SIGNED**

Add the source and texture with an 0.5 subtraction

#### **GW\_TEX\_SUBTRACT**

Subtract the source from the texture

#### **GW\_TEX\_ADD\_SMOOTH**

Add the source and the texture then subtract their product

#### **GW\_TEX\_ALPHA\_BLEND**

Alpha blend the texture with the source

#### **GW\_TEX\_PREMULT\_ALPHA\_BLEND**

Alpha blend the source with a premultiplied alpha

### **UBYTE alphaAlphaSource;**

The alpha blend alpha source. One of the following values:

#### **GW\_TEX\_ZERO**

Use no alpha value

#### **GW\_TEX\_SOURCE**

Use the source alpha

#### **GW\_TEX\_TEXTURE**

Use the texture alpha

#### **GW\_TEX\_CONSTANT**

Use a constant BGRA color as an alpha

#### **GW\_TEX\_PREVIOUS**

Use the previous texture stage alpha

## **UBYTE alphaScale;**

The alpha scale factor. One of the following values:

**GW\_TEX\_SCALE\_1X**

Multiply the tex op result by 1

**GW\_TEX\_SCALE\_2X**

Multiply the tex op result by 2

**GW\_TEX\_SCALE\_4X**

Multiply the tex op result by 4

## **Methods:**

public:

## **Prototype:**

**TextureInfo();**

## **Remarks:**

Constructor. The data members are initialized as follows:

**useTex = 1;**

**faceMap = 0;**

**textHandle = 0;**

**uvwSource = UVSOURCE\_MESH;**

**mapChannel = 1;**

**tiling[0] = tiling[1] = tiling[2] = GW\_TEX\_REPEAT;**

**colorOp = GW\_TEX\_MODULATE;**

**colorAlphaSource = GW\_TEX\_TEXTURE;**

**colorScale = GW\_TEX\_SCALE\_1X;**

**alphaOp = GW\_TEX\_LEAVE;**

**alphaAlphaSource = GW\_TEX\_TEXTURE;**

**alphaScale = GW\_TEX\_SCALE\_1X;**

## **Prototype:**

**~TextureInfo();**

## **Remarks:**

Destructor.

## Class CustomParticleDisplay

See Also: [Class ParticleSys](#), [Class GraphicsWindow](#).

class CustomParticleDisplay

### Description:

This class allow a plug-in particle system to provide its own custom drawing routine. Implement the **DrawParticle()** method of this class and register this callback with the **SetCustomDraw()** method of class **ParticleSys**.

### Methods:

### Prototype:

```
virtual void DrawParticle(GraphicsWindow *gw,  
    ParticleSys &parts, int i)=0;
```

### Remarks:

Implemented by the Plug-In.

Draws the 'i-th' particle of the specified particle system.

### Parameters:

**GraphicsWindow \*gw**

The window into which to draw the particle.

**ParticleSys &parts**

The particle system whose 'i-th' particle is to be drawn.

**int i**

The index of the particle to draw.

## Class ImpNode

See Also: [Class ImpInterface](#), [Class INode](#), [Class Matrix3](#), [Class Point3](#).

class ImpNode

### Description:

Import Node class. Methods of this class may be used to set various properties of the node. All methods of this class are implemented by the system.

### Sample Code:

The following sample code fragment (from `\MAXSDK\SAMPLES\IMPEXP\DXFIMP.CPP`) demonstrates the use of many of the methods of this class.

```
ImpNode *node = iface->CreateNode();
if (node) {
    TriObject *tri = CreateNewTriObject();
    // Now find the center of the vertices and use that as the pivot
    int verts = m->getNumVerts();
    Point3 accum(0,0,0);
    for(int i = 0; i < verts; ++i)
        accum += m->verts[i];
    Point3 delta = accum / (float)verts;
    for(i = 0; i < verts; ++i)
        m->verts[i] -= delta;
    tri->mesh = *m;
    node->Reference(tri);
    Matrix3 tm;
    tm.IdentityMatrix();// Reset initial matrix to identity
    tm.SetTrans(delta); // Add in the center point
    node->SetTransform(0,tm);
    iface->AddNodeToScene(node);
    node->SetName(_T(n->name));
}
```

Methods:

Prototype:

```
virtual RefResult Reference(ObjectHandle obj) = 0;
```

**Remarks:**

Sets the object that this node references.

**Parameters:****ObjectHandle obj**

The object to reference.

**Return Value:**

One of the following values:

**REF\_FAIL**

The operation failed.

**REF\_SUCCEED**

The operation succeeded.

**Prototype:**

```
virtual void SetTransform( TimeValue t, Matrix3 tm ) = 0;
```

**Remarks:**

Sets the transformation matrix of the node.

**Parameters:****TimeValue t**

The time to set the matrix.

**Matrix3 tm**

The new transformation matrix of the node.

**Prototype:**

```
virtual void SetName(const TCHAR *newname) = 0;
```

**Remarks:**

Sets the name of the node.

**Parameters:****const TCHAR \*newname**

The new name for the node.

**Prototype:**

```
virtual void SetPivot(Point3 p) = 0;
```

**Remarks:**

Sets the pivot point of the node.

**Parameters:****Point3 p**

The pivot point of the node.

**Prototype:****virtual INode \*GetINode()=0;****Remarks:**

Returns the INode pointer for the node.

## Class RendPickProc

See Also: [Class IRendParams](#).

class RendPickProc

### Description:

An instance of this class is passed to **IRendParams::SetPickMode()**. This is a callback that gets called as the user tries to pick objects in the scene.

### Methods:

#### Prototype:

```
virtual BOOL Pick(INode *node)=0;
```

#### Remarks:

Implemented by the Plug-In.

Called when the user picks something.

#### Parameters:

**INode \*node**

The node that was selected.

#### Return Value:

TRUE to end the pick mode; FALSE to continue.

#### Prototype:

```
virtual BOOL Filter(INode *node)=0;
```

#### Remarks:

Implemented by the Plug-In.

Return TRUE if this is an acceptable hit; otherwise FALSE.

#### Parameters:

**INode \*node**

The node that was selected.

#### Prototype:

```
virtual void EnterMode()
```

#### Remarks:

Implemented by the Plug-In.

This method is called as the mode is entered.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual void ExitMode()
```

**Remarks:**

Implemented by the Plug-In.

This method is called when the mode is exited.

**Default Implementation:**

```
{}
```

**Prototype:**

```
virtual HCURSOR GetDefCursor()
```

**Remarks:**

Implemented by the Plug-In.

Returns the handle of the default cursor. This is the cursor to use when the user is NOT over a pickable object.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

```
virtual HCURSOR GetHitCursor()
```

**Remarks:**

Implemented by the Plug-In.

Returns the handle of the hit cursor. This is the cursor to use when the user IS over a pickable object.

**Default Implementation:**

```
{return NULL;}
```

**Prototype:**

**virtual BOOL AllowMultiSelect();**

**Remarks:**

This method is available in release 2.0 and later only.

Implement this method to return TRUE to allow the user to pick more than one thing. In that case the **Pick()** method may be called more than once.

**Return Value:**

TRUE to allow multiple picks; otherwise FALSE.

**Default Implementation:**

**{return FALSE;}**

## Class TexHandle

See Also: [Class TexHandleMaker](#).

class TexHandle

### Description:

This class defines a texture handle. A pointer to an instance of this class is returned from the methods of **TexHandleMaker**. Methods of this class allow the handle to be retrieved and to delete the handle.

### Methods:

#### Prototype:

**virtual DWORD GetHandle() = 0;**

#### Remarks:

Implemented by the Plug-In.

This method is called to retrieve the texture handle.

#### Prototype:

**virtual void DeleteThis() = 0;**

#### Remarks:

Implemented by the Plug-In.

This method is called to delete the instance of the texture handle.

## Class ShaderParamDlg

See Also: [Class ParamDlg](#), [Class StdMat2](#), [Class Shader](#).

class ShaderParamDlg : public ParamDlg

### Description:

This class is available in release 3.0 and later only.

A pointer to an instance of this class is returned by a Shader when it is asked to put up its rollup page.

### Methods:

public:

### Prototype:

**virtual Class\_ID ClassID()=0;**

### Remarks:

Implemented by the Plug-In.

Returns the unique Class\_ID of this object.

### Prototype:

**virtual void SetThing(ReferenceTarget \*m)=0;**

### Remarks:

Implemented by the Plug-In.

This method sets the current shader being edited to the shader passed.

### Parameters:

**ReferenceTarget \*m**

The Shader to set as current.

### Prototype:

**virtual void SetThings(StdMat2\* pMtl, Shader\* pShader)=0;**

### Remarks:

Implemented by the Plug-In.

This method sets the current Standard material (and its shader) being edited to the ones passed.

**Parameters:**

**StdMtl2\* pMtl**

The Standard material to set as current.

**Shader\* pShader**

The Shader to set as current.

**Prototype:**

**virtual ReferenceTarget\* GetThing()=0;**

**Remarks:**

Returns the a pointer to the current **material** being edited. Note that in most of the Get/SetThing() methods in the SDK the 'Thing' is the actual plug-in. In this case it's not. It the material which is using this Shader.

**Prototype:**

**virtual Shader\* GetShader()=0;**

**Remarks:**

This method returns a pointer to the current Shader.

**Prototype:**

**virtual void SetTime(TimeValue t);**

**Remarks:**

This method is called when the current time has changed. This gives the developer an opportunity to update any user interface data that may need adjusting due to the change in time.

**Parameters:**

**TimeValue t**

The new current time.

**Default Implementation:**

**{}**

**Prototype:**

**virtual void DeleteThis()=0;**

**Remarks:**

This method is called to delete this instance of the class.

For dynamically created global utility plugins, this method has to be implemented and should have a implementation like **{ delete this; }**

**Prototype:**

**virtual BOOL PanelProc(HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM lParam)=0;**

**Remarks:**

This is the dialog procedure for the user interface controls of the Shader.

**Parameters:****HWND hwndDlg**

The window handle of the rollup page.

**UINT msg**

The message to process.

**WPARAM wParam**

The first dialog parameter.

**LPARAM lParam**

The second dialog parameter.

**Return Value:**

Except in response to the WM\_INITDIALOG message, the procedure should return nonzero if it processes the message, and zero if it does not. In response to a WM\_INITDIALOG message, the dialog box procedure should return zero if it calls the SetFocus function to set the focus to one of the controls in the dialog. Otherwise, it should return nonzero, in which case the system sets the focus to the first control in the dialog that can be given the focus.

**Prototype:**

**virtual void LoadDialog(int draw)=0;**

**Remarks:**

This method is used to load the user interface controls with their current values.

**Parameters:**

**int draw**

This parameter is not currently used.

**Prototype:**

**virtual HWND GetHwnd()=0;**

**Remarks:**

This method returns the window handle of the rollup panel.

**Prototype:**

**virtual int FindSubTexFromHWND(HWND hw)=0;**

**Remarks:**

This method returns the index of the sub-texmap corresponding to the window whose handle is passed. If the handle is not valid return -1.

**Parameters:**

**HWND hw**

The window handle to check.

**Prototype:**

**virtual void UpdateOpacity()=0;**

**Remarks:**

This method is called to update the opacity parameter of the plug-in in the user interface.

**Prototype:**

**virtual void UpdateMapButtons()=0;**

**Remarks:**

This method is called to update the map buttons in the user interface. For example it can put a " " or "m" or "M" on the button face based on the state of the map.

## Class StdMat2

See Also: [Class StdMat](#), [Class Shader](#), [Class Sampler](#), [Class Class\\_ID](#).

class StdMat2 : public StdMat

### Description:

This class is available in release 3.0 and later only.

This is the base class for all materials supporting the plug-in shader mechanism. The 3ds max Standard material is derived from this class.

### Methods:

public:

### Prototype:

**virtual BOOL KeyAtTime(int id, TimeValue t)=0;**

### Remarks:

Returns TRUE if the specified parameter whose ID is passed has a key at the time passed; otherwise FALSE.

### Parameters:

**int id**

The ID of the parameter to check.

**TimeValue t**

The time to check.

### Prototype:

**virtual int GetMapState(int indx)=0;**

### Remarks:

Returns a value to indicate the state of the specified map. One of the following values:

**0:** No map present.

**1:** Map present but disabled.

**2:** Map present and on.

### Parameters:

**int indx**

The index of the map to check. See [List of Texture Map Indices](#).

**Prototype:**

```
virtual TSTR GetMapName(int indx)=0;
```

**Remarks:**

Returns the name of the map whose index is passed.

**Parameters:****int indx**

The index of the map to check. See [List of Texture Map Indices](#).

**Prototype:**

```
virtual void SyncADTexLock(BOOL lockOn)=0;
```

**Remarks:**

This method is called when the state of the Ambient/Diffuse Texture lock is toggled. The material should store the setting and update the UI as required.

**Parameters:****BOOL lockOn**

TRUE for on; FALSE for off.

**Prototype:**

```
virtual BOOL SwitchShader(Class_ID id)=0;
```

**Remarks:**

This method is called when a new Shader has been selected.

**Parameters:****Class\_ID id**

The Class\_ID of the new shader to switch to.

**Prototype:**

```
virtual Shader* GetShader()=0;
```

**Remarks:**

Returns a pointer to the Shader in use. See [Class Shader](#) for details on this

plug-in type.

**Prototype:**

**virtual BOOL IsFaceted()=0;**

**Remarks:**

Returns TRUE if the shader is faceted; otherwise FALSE. The pre-R3 Constant shader is faceted. The other shaders are not.

**Prototype:**

**virtual void SetFaceted(BOOL on)=0;**

**Remarks:**

Sets the faceted setting of the Shader.

**Parameters:**

**BOOL on**

TRUE if it is faceted; FALSE if not.

**Prototype:**

**virtual long StdIDToChannel(long id)=0;**

**Remarks:**

Returns the index of the mapping channels which corresponds to the specified Standard materials texture map ID.

**Parameters:**

**long id**

The ID whose corresponding channel to return. See [List of Texture Map Indices](#).

**Return Value:**

The zero based index of the channel. If there is not a corresponding channel return -1.

**Prototype:**

**virtual void SetShading(int s);**

**Remarks:**

Sets the active shader to the one specified. The supported types are the pre-R3 shaders.

**Parameters:**

**int s**

One of the following values (all other values are a NOOP):

**SHADE\_CONST** (Phong, faceted).

**SHADE\_PHONG**

**SHADE\_METAL**

**SHADE\_BLINN**

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual int GetShading();
```

**Remarks:**

Returns one of the pre-R3 shader types. If an R3 shader type is active, SHADE\_BLINN is returned.

**Return Value:**

One of the following values:

**SHADE\_CONST** (Phong, faceted).

**SHADE\_PHONG**

**SHADE\_METAL**

**SHADE\_BLINN**

**Default Implementation:**

```
{ return -1; }
```

**Prototype:**

```
virtual BOOL SwitchSampler(Class_ID id)=0;
```

**Remarks:**

This method is called when the active Sampler is switched.

**Parameters:**

**Class\_ID id**

The Class\_ID of the new Sampler.

**Prototype:**

**virtual Sampler \*GetPixelSampler()=0;**

**Remarks:**

Returns a pointer to the sampler used.

**Prototype:**

**virtual BOOL GetSelfIllumColorOn(int mtlNum=0, BOOL backFace=FALSE)=0;**

**Remarks:**

Returns the Self Illumination Color On setting. TRUE if on; FALSE if off.

**Parameters:**

These parameters are not used and may be ignored.

**Prototype:**

**virtual Color GetSelfIllumColor(int mtlNum, BOOL backFace)=0;**

**Remarks:**

Returns the Self Illumination Color setting.

**Parameters:**

These parameters are not used and may be ignored.

**Prototype:**

**virtual Color GetSelfIllumColor(TimeValue t)=0;**

**Remarks:**

Returns the Self Illumination Color setting at the specified time.

**Parameters:**

**TimeValue t**

The time at which to get the color.

**Prototype:**

**virtual void SetSelfIllumColorOn(BOOL on)=0;**

**Remarks:**

Sets the Self Illumination Color On setting

**Parameters:**

**BOOL on**

TRUE for on; FALSE for off.

**Prototype:**

**virtual void SetSelfIllumColor(Color c, TimeValue t)=0;**

**Remarks:**

Sets the Self Illumination Color setting at the specified time.

**Parameters:**

**Color c**

The color to set.

**TimeValue t**

The time at which to set the color.

## List of Shader Standard Parameter Flags

See Also: [Class Shader](#), [Class IllumParams](#).

These are the flags that are returned in **Shader::SupportStdParams()** or stored in **IllumParams::stdParams**.

One or more of the following values:

### **STD\_PARAM\_NONE**

This indicates none of the flags below.

### **STD\_PARAM\_ALL**

This indicates all of the flags below.

### **STD\_PARAM\_METAL**

This bit is only used by the Metal shader.

### **STD\_PARAM\_LOCKDS**

Indicate support for the Diffuse / Specular lock.

### **STD\_PARAM\_LOCKAD**

Indicates support for the Ambient / Diffuse lock.

### **STD\_PARAM\_LOCKADTEX**

Indicates support for the Ambient / Diffuse texture lock.

### **STD\_PARAM\_SELFILLUM**

Indicates support for the Self Illumination parameter.

### **STD\_PARAM\_SELFILLUM\_CLR**

Indicates support for the Self Illumination color parameter.

### **STD\_PARAM\_AMBIENT\_CLR**

Indicates support for the Ambient color parameter.

### **STD\_PARAM\_DIFFUSE\_CLR**

Indicates support for the Diffuse color parameter.

### **STD\_PARAM\_SPECULAR\_CLR**

Indicates support for the Specular color parameter.

### **STD\_PARAM\_FILTER\_CLR**

Indicates support for the Filter color parameter.

### **STD\_PARAM\_GLOSSINESS**

Indicates support for the Glossiness parameter.

### **STD\_PARAM\_SOFTEN\_LEV**

Indicates support for the Soften Level parameter.

### **STD\_PARAM\_SPECULAR\_LEV**

Indicates support for the Specular Level parameter.

### **STD\_PARAM\_DIFFUSE\_LEV**

Indicates support for the Diffuse Level parameter.

### **STD\_PARAM\_DIFFUSE\_RHO**

Indicates support for the Roughness parameter.

### **STD\_PARAM\_ANISO**

Indicates support for the Specular Highlight Anisotropy parameter.

### **STD\_PARAM\_ORIENTATION**

Indicates support for the Specular Highlight Orientation parameter.

### **STD\_PARAM\_REFL\_LEV**

This is reserved for future use.

### **STD\_PARAM\_SELFILLUM\_CLR\_ON**

Indicates support for the Self Illumination Color On/Off checkbox.

### **STD\_BASIC2\_DLG**

This bit is only set by the three pre-R3 shaders (Phong, Blinn, and Metal). If this bit is not set then the Basic Parameters dialog is replaced by the one provided by the plug-in shader.

### **STD\_EXTRA\_DLG**

Indicates support for the Extended Parameters rollout. If this bit is not set one provided by the plug-in shader will be used instead.

The following three flags, when set, enable the specified controls in the Extended Parameters rollout.

#### **STD\_EXTRA\_REFLECTION**

Indicates support for Reflection Dimming parameters (Apply, Dim Level, Refl Level).

#### **STD\_EXTRA\_REFRACTION**

Indicates support for Index of Refraction parameter.

#### **STD\_EXTRA\_OPACITY**

Indicates support for Opacity parameters (Amount, In/Out, Type).

## List of Color Conversion Utilities

The following global functions are available for color conversion. These are defined in `\MAXSDK\INCLUDE\HSV.H`.

**Prototype:**

```
void RGBtoHSV(DWORD rgb, int *ho, int *so, int *vo);
```

**Remarks:**

Converts the specified color in RGB to HSV.

**Parameters:**

**DWORD rgb**

The RGB color to convert.

**int \*ho**

The hue output.

**int \*so**

The saturation output.

**int \*vo**

The value output.

**Prototype:**

```
DWORD HSVtoRGB(int H, int S, int V);
```

**Remarks:**

Converts the specified color in HSV color to RGB.

**Parameters:**

**int H**

The input hue.

**int S**

The input saturation.

**int V**

The input value.

**Return Value:**

The RGB color as a DWORD. See [COLORREF](#).

**Prototype:**

```
void HSVtoHWBt(int h, int s, int v, int *ho, int *w, int *bt);
```

**Remarks:**

Converts the specified color in HSV color to Hue Whiteness and Blackness (HWBt).

**Parameters:**

**int h**

The input hue.

**int s**

The input saturation.

**int v**

The input value.

**int \*ho**

The hue output.

**int \*w**

The whiteness output.

**int \*bt**

The blackness output.

**Prototype:**

```
void HWBttoHSV(int h, int w, int bt, int *ho, int *s, int *v);
```

**Remarks:**

Converts the specified color in Hue Whiteness and Blackness (HWBt) color to HSV.

**Parameters:**

**int h**

The hue input.

**int w**

The whiteness input.

**int bt**

The blackness input.

**int \*ho**

The hue output.

**int \*s**

The saturation output.

**int \*v**

The value output.

## Class IMCParamDlg

See Also: [Class ReferenceMaker](#), [Class MCDeviceBinding](#), [Class IRollupWindow](#).

```
class IMCParamDlg : public ReferenceMaker
```

### Description:

This class is an interface to allow the plug-in to provide a user interface in the command panel. It has two data members.

### Data Members:

public:

**MCDeviceBinding \*binding;**

Returns a pointer to the device binding.

**IRollupWindow \*iRoll;**

This is an interface into the command panel. Its methods may be used to work with rollup pages and alter UI controls.

## Class MeshSubHitRec

See Also: [Class Mesh](#), [Class BitArray](#).

class MeshSubHitRec

### Description:

This class allows access to the sub-object hit records used in Mesh hit testing. All methods of this class are implemented by the system.

### Data Members:

public:

#### **DWORD dist;**

The distance of the hit. If the user is in wireframe mode, this is the distance in pixels to the item that was hit. If the user is in shaded mode, this is the Z depth distance. Smaller numbers indicate a closer hit.

#### **int index;**

The index of the sub-object component. For example, if faces were being hit tested, this would be the index of the mesh's BitArray **faceSel**. For edges, this is the index into the **edgeSel** BitArray, where the index is

**3\*faceIndex+edgeIndex.**

#### **DWORD flags;**

These are not currently used.

### Methods:

#### Prototype:

**MeshSubHitRec(DWORD dist, int index, MeshSubHitRec \*next);**

#### Remarks:

Constructor. The data members are initialized to the data members passed.

#### Prototype:

**MeshSubHitRec(DWORD dist, int index, DWORD flags,  
MeshSubHitRec \*next)**

#### Remarks:

Constructor. The data members are initialized to the data members passed.

**Prototype:**

**MeshSubHitRec \*Next();**

**Remarks:**

Returns the next mesh sub hit record.

# Class GenericHierarchy

See Also: [Class HierarchyEntry](#).

class GenericHierarchy

## Description:

This is a utility class for describing hierarchies of shapes. All methods of this class are implemented by the system.

This is used in generating mesh objects from shapes. In order for a mesh object to be generated correctly, nested shapes must be oriented clockwise or counter-clockwise depending on their level of nesting. For example, a donut shape with two circular curves will have the outer shape going counter-clockwise and the inner shape going clockwise. If a third shape was nested inside both of these, its points would be going counter-clockwise.

## Methods:

### Prototype:

**GenericHierarchy()**

### Remarks:

Constructor. The hierarchy is set as initially empty.

### Prototype:

**void AddEntry(int data, int parent = -1);**

### Remarks:

This method adds one entry given its parent.

### Parameters:

**int data**

The polygon index of the entry to add.

**int parent = -1**

The index of the parent of the entry.

### Prototype:

**int Entries();**

**Remarks:**

Returns the total number of members in the hierarchy.

**Prototype:**

**HierarchyEntry\* GetStart();**

**Remarks:**

Retrieves the first item under the root.

**Return Value:**

The first **HierarchyEntry** under the root.

**Prototype:**

**HierarchyEntry\* FindEntry(int data, HierarchyEntry\* start = NULL);**

**Remarks:**

Finds the specified entry in the hierarchy.

**Parameters:**

**int data**

The polygon index of the entry to find.

**HierarchyEntry\* start = NULL**

The entry at which to begin the search. If NULL is specified the search starts at the root.

**Return Value:**

A pointer to the **HierarchyEntry** of the found entry. If not found, NULL is returned.

**Prototype:**

**int NumberOfChildren(int data);**

**Remarks:**

Returns the number of children for this item.

**Parameters:**

**int data**

The index of the polygon to return the number of children of.

**Prototype:**

**int GetChild(int data, int index);**

**Remarks:**

Returns the specified child of the specified entry.

**Parameters:**

**int data**

The index of the polygon whose child is to be returned.

**int index**

Specifies which child to return.

**Return Value:**

The specified child of the entry.

**Prototype:**

**void New();**

**Remarks:**

Clear out the hierarchy tree.

**Prototype:**

**void Sort();**

**Remarks:**

Sorts the hierarchy tree by children / siblings. This is used internally as all the sorting is done automatically as the hierarchy is generated.

**Prototype:**

**BOOL IsCompatible(GenericHierarchy& hier);**

**Remarks:**

Determines if this hierarchy and the specified hierarchy are compatible.

**Parameters:**

**GenericHierarchy& hier**

The hierarchy to check for compatibility.

**Return Value:**

TRUE if the hierarchies are compatible; otherwise FALSE.

**Prototype:**

**void Dump(HierarchyEntry\* start = NULL);**

**Remarks:**

This method is used internally to **DebugPrint()** the tree. See [Debugging](#).

**Prototype:**

**TSTR& SortKey();**

**Remarks:**

Returns the sort key for the hierarchy. This is used internally.

**Operators:**

**Prototype:**

**GenericHierarchy& operator=(GenericHierarchy& from);**

**Remarks:**

Copy operator.

**Parameters:**

**GenericHierarchy& from**

The hierarchy to copy from.

## Class NoteKeyTab

See Also: [Class NoteKey](#), [Template Class Tab](#), [Class DefNoteTrack](#), [Class Animatable](#).

```
class NoteKeyTab : public Tab<NoteKey*>
```

### Description:

This class is table of pointers to **NoteKey** objects which store data about a Note Track in Track View. See **Template Class Tab** for details on manipulating this table.

### Methods:

public:

### Prototype:

```
~NoteKeyTab();
```

### Remarks:

Destructor. Deletes all the keys in the table.

### Prototype:

```
void Clear();
```

### Remarks:

Deletes all the keys in the table.

### Prototype:

```
void DelKey(int i);
```

### Remarks:

Deletes the specified key.

### Parameters:

```
int i
```

The zero based index of the key to delete.

### Prototype:

```
void KeysChanged();
```

**Remarks:**

This method is used internally to sort the keys by time.

**Operators:**

public:

**Prototype:**

**NoteKeyTab &operator=(NoteKeyTab &keys);**

**Remarks:**

Assignment operator.

**Parameters:**

**NoteKeyTab &keys**

The table of keys to assign.

## Structure SubRenderParams

See Also: [Class RenderMapsContext](#).

```
struct SubRenderParams : public BaseInterfaceServer
```

### Description:

This structure contains information on rendering for Mirror and Automatic Cubic materials. This is used by the methods of the **RenderMapsContext** class.

### Structure:

```
struct SubRenderParams {
```

#### **RenderType** renderType;

The rendering type being done. See the [List of Render Types](#) for more information.

#### **BOOL** fieldRender;

TRUE if field rendering is being used; otherwise FALSE.

#### **BOOL** evenLines;

This is used when field rendering. TRUE if doing even numbered scanlines; FALSE for odd numbered.

#### **BOOL** doingMirror;

This is used as part of implementing the Mirror material. It should be FALSE in all other cases.

#### **int** devWidth, devHeight;

The dimensions in pixels of Bitmap **tobm**.

#### **float** devAspect;

The aspect ratio of Bitmap **tobm**.

#### **int** xorg, yorg;

The location on the screen of the upper left corner of the output bitmap.

#### **int** xmin,xmax,ymin,ymax;

The area of the screen being rendered.

#### **Point2** blowupCenter;

This parameter is available in release 4.0 and later only.

The 2D point at the center of the render blowup region.

#### **Point2** blowupFactor;

This parameter is available in release 4.0 and later only.  
The X and Y scale factors for render blowup.

**Prototype:**

```
virtual INT_PTR Execute(int cmd, ULONG arg1=0, ULONG  
arg2=0, ULONG arg3=0);
```

**Remarks:**

This method is available in release 2.0 and later only.

This is a general purpose function that allows the API to be extended in the future. The 3ds max development team can assign new **cmd** numbers and continue to add functionality to this class without having to 'break' the API.

**Parameters:**

**int cmd**

The index of the command to execute.

**ULONG arg1=0**

Optional argument 1. See the documentation where the **cmd** option is discussed for more details on these parameters.

**ULONG arg2=0**

Optional argument 2.

**ULONG arg3=0**

Optional argument 3.

**Return Value:**

An integer return value. See the documentation where the **cmd** option is discussed for more details on the meaning of this value.

```
};
```

## Class Point4

See Also: [Class Point3](#).

class Point4

### Description:

This class describes a point using float x, y, z and w coordinates. Methods are provided to add and subtract points, multiply and divide by scalars, and element by element multiply and divide two points. All methods are implemented by the system.

This class is available in release 2.0 and later only.

### Data Members:

public:

**float x, y, z, w;**

The x, y, z and w components of the point.

**static const Point4 Origin;**

This is equivalent to **Point4(0.0f, 0.0f, 0.0f, 0.0f);**

**static const Point4 XAxis;**

This is equivalent to **Point4(1.0f, 0.0f, 0.0f, 0.0f);**

**static const Point4 YAxis;**

This is equivalent to **Point4(0.0f,1.0f, 0.0f, 0.0f);**

**static const Point4 ZAxis;**

This is equivalent to **Point4(0.0f, 0.0f,1.0f, 0.0f);**

**static const Point4 WAxis;**

This is equivalent to **Point4(0.0f, 0.0f, 0.0f,1.0f);**

### Methods:

### Prototype:

**Point4 ()**

### Remarks:

Constructor. No initialization is performed.

**Prototype:**

**Point4(float X, float Y, float Z, float W)**

**Remarks:**

Constructor. x, y, z and w are initialized to the values specified.

**Prototype:**

**Point4(double X, double Y, double Z, double W)**

**Remarks:**

Constructor. x, y, z and w are initialized to the specified values (cast as floats).

**Prototype:**

**Point4(int X, int Y, int Z, int W)**

**Remarks:**

Constructor. x, y, z and w are initialized to the specified values (cast as floats).

**Prototype:**

**Point4(const Point3& a, float W=0)**

**Remarks:**

Constructor. x, y, z and w are initialized to the specified Point3 and W.

**Prototype:**

**Point4(float af[4])**

**Remarks:**

Constructor. x, y, z and w are initialized to af[0], af[1], af[2] and af[3] respectively.

**Prototype:**

**inline Point4& Set(float X, float Y, float Z, float W);**

**Remarks:**

This method is available in release 3.0 and later only.

Sets the x, y, z and w coordinate to the values passed and returns a reference to this Point4.

**Parameters:****float X**

The new x value.

**float Y**

The new y value.

**float Z**

The new z value.

**float W**

The new w value.

**Return Value:**A reference to this **Point4**.**Prototype:****int Equals(const Point4& p, float epsilon = 1E-6f);****Remarks:**

This method is available in release 3.0 and later only.

Compares this Point4 and the specified one to see if the x, y, z and w values are within plus or minus the specified tolerance.

**Parameters:****const Point4& p**

The point to compare.

**float epsilon = 1E-6f**

The tolerance to use in the comparison.

**Return Value:**

Nonzero if the points are 'equal'; otherwise zero.

**Operators:****Prototype:****float& operator[](int i)****const float& operator[](int i) const****Remarks:**

Allows access to x, y, z and w using the subscript operator.

**Return Value:**

An value for **i** of 0 will return x, 1 will return y, 2 will return z and 3 will return w.

**Prototype:**

**operator float\*()**

**Remarks:**

Conversion function. Returns the address of the Point4.x

**Prototype:**

**Point4 operator-() const**

**Remarks:**

Unary - operator. Negates x, y, z and w.

**Prototype:**

**Point4 operator+() const**

**Remarks:**

Unary +. Returns the Point4.

**Prototype:**

**inline Point4& operator-=(const Point4&);**

**Remarks:**

Subtracts a Point4 from this Point4.

**Return Value:**

A Point4 that is the difference between two Point4s.

**Prototype:**

**inline Point4& operator+=(const Point4&);**

**Remarks:**

Adds a Point4 to this Point4.

**Return Value:**

A Point4 that is the sum of two Point4s.

**Prototype:**

**inline Point4& operator\*=(float);**

**Remarks:**

Multiplies this Point4 by a floating point value.

**Return Value:**

A Point4 multiplied by a float.

**Prototype:**

**inline Point4& operator/=(float);**

**Remarks:**

Divides this Point4 by a floating point value.

**Return Value:**

A Point4 divided by a float.

**Prototype:**

**inline Point4& operator\*=(const Point4&);**

**Remarks:**

Element-by-element multiplication of two Point4s:  
(**x\*x, y\*y, z\*z, w\*w**).

**Return Value:**

A Point4 element-by-element multiplied by another Point4.

**Prototype:**

**int operator==(const Point4& p) const**

**Remarks:**

Equality operator. Test for equality between two Point4's.

**Return Value:**

Nonzero if the Point4's are equal; otherwise 0.

**Prototype:**

**inline Point4 operator-(const Point4&) const;**

**Remarks:**

Subtracts a Point4 from a Point4.

**Return Value:**

A Point4 that is the difference between two Point4s.

**Prototype:**

```
inline Point4 operator+(const Point4&) const;
```

**Remarks:**

Adds a Point4 to a Point4.

**Return Value:**

A Point4 that is the sum of two Point4s.

**Prototype:**

```
inline Point4 operator/(const Point4&) const;
```

**Remarks:**

Divides a Point4 by a Point4 element by element.

**Return Value:**

A Point4 resulting from dividing a Point4 by a Point4 element by element.

**Prototype:**

```
inline Point4 operator*(const Point4&) const;
```

**Remarks:**

Multiplies a Point4 by a Point4 element by element.

**(x\*x, y\*y, z\*z, w\*w).**

**Return Value:**

A Point4 resulting from the multiplication of a Point4 and a Point4.

**Prototype:**

```
inline Point4 operator*(float f, const Point4& a)
```

**Remarks:**

Returns a Point4 that is the specified Point4 multiplied by the specified float.

**Prototype:**

**inline Point4 operator\*(const Point4& a, float f)**

**Remarks:**

Returns a Point4 that is the specified Point4 multiplied by the specified float.

**Prototype:**

**inline Point4 operator/(const Point4& a, float f)**

**Remarks:**

Returns a Point4 that is the specified Point4 divided by the specified float.

**Prototype:**

**inline Point4 operator+(const Point4& a, float f)**

**Remarks:**

Returns a Point4 that is the specified Point4 with the specified floating point valued added to each component x, y, z and w.

## Class RefListItem

See Also: [Class RefList](#).

class RefListItem

### Description:

This class represents a single entry in a RefList.

### Data Members:

public:

**RefMakerHandle maker;**

**RefListItem \*next;**

### Methods:

### Prototype:

**RefListItem(RefMakerHandle hmaker, RefListItem \*list)**

### Remarks:

Constructor. The data members are assigned.

## List of ParamTags Choices

See Also: [Class ParamBlockDesc2](#), [List of ParamType Choices](#), [Class PBAccessor](#), [Class PBValidator](#), [Class ParamDimension](#).

These are the parameter definition optional information tags. These tags are used in the **ParamBlockDesc2** main constructor as part of the <optional\_tagged\_param\_specs>. The typical format is:

<tag>, <optional\_param\_spec>,

The following options are available. The hyperlinks take you to the start of each tag description.

**enum ParamTags**

{

[p\\_default](#),

[p\\_ms\\_default](#),

[p\\_range](#),

[p\\_ui](#),

[p\\_uix](#),

[p\\_validator](#),

[p\\_accessor](#),

[p\\_vals](#),

[p\\_refno](#),

[p\\_subtexno](#),

[p\\_submtlno](#),

[p\\_dim](#),

[p\\_classID](#),

[p\\_sclassID](#),

[p\\_enabled](#),

[p\\_enable\\_ctrls](#),

[p\\_prompt](#),

[p\\_caption](#),

[p\\_init\\_file](#),

```
p_file_types,  
end,  
};
```

## Option Descriptions:

### **p\_default**

The default value assigned when a block is first created. It must be of the correct type to match the **ParamType** of the parameter (for example float for TYPE\_FLOATs, int for TYPE\_INTs, Color(x,y,z) for TYPE\_RGBA, Point3(x,y,z) for TYPE\_POINT3s, etc.) Defaults can only be supplied for the following base types:

```
TYPE_ANGLE, TYPE_PCNT_FRAC,  
TYPE_COLOR_CHANNEL, TYPE_FLOAT,  
TYPE_TIMEVALUE, TYPE_INT, TYPE_BOOL,  
TYPE_RADIOBTN_INDEX, TYPE_POINT3, TYPE_RGBA,  
TYPE_STRING, TYPE_FILENAME, TYPE_MATRIX3
```

Examples:

```
p_default, FALSE,  
p_default, 1,  
p_default, Point3(0,0,0),  
p_default, 25.0,
```

### **p\_ms\_default**

This establishes the default value used during MAXScript creation. For example the MAXScript command **sphere.radius** defaults to 25.0 when created by the scripser, but **p\_default** is set to 0.0 so interactive creation starts out with a point-sized sphere when the first mouse click is made.

Example:

```
p_ms_default, 25.0,
```

### **p\_range**

This establishes allowable ranges used in MAXScript validation and spinner setup. Supplied as two values of the correct type (as described in **p\_default**, above). Ranges can only be supplied for the following types:

**TYPE\_ANGLE, TYPE\_PCNT\_FRAC,  
TYPE\_COLOR\_CHANNEL, TYPE\_FLOAT,  
TYPE\_TIMEVALUE, TYPE\_INT,  
TYPE\_RADIOBTN\_INDEX, TYPE\_POINT3, TYPE\_RGBA**

Example:

**p\_range, -99999999.0, 99999999.0,**

**p\_ui**

This is the user interface control specification.

This optional tag takes a variable list of arguments depending on the type of UI control specified. This sequence of arguments is similar in form to the ParamDescUI class constructors in the pre-release 3 ParamMap system.

Following the p\_ui tag, one of the following control types should be specified. Following that are further specs as defined in each type.

Note: If a p\_ui is supplied for a Tab<> parameter type and the control is \*not\* of on the ListBox types, the table size should be fixed and supplied in the <required\_param\_specs> table\_size field, and you should supply a set of dialog item ID's for each element in the table. See the example in TYPE\_SPINNER below.

Control Types Links

Single Controls

[\*\*TYPE\\_SPINNER\*\*](#)

[\*\*TYPE\\_SLIDER\*\*](#)

[\*\*TYPE\\_RADIO\*\*](#)

[\*\*TYPE\\_CHECKBUTTON\*\*](#)

[\*\*TYPE\\_SINGLECHEKBOX\*\*](#)

[\*\*TYPE\\_MULTICHEKBOX\*\*](#)

[\*\*TYPE\\_COLORSWATCH\*\*](#)

[\*\*TYPE\\_EDITBOX\*\*](#)

[\*\*TYPE\\_PICKNODEBUTTON\*\*](#)

[\*\*TYPE\\_TEXMAPBUTTON\*\*](#)

[\*\*TYPE\\_BITMAPBUTTON\*\*](#)

[\*\*TYPE\\_MTLBUTTON\*\*](#)

[TYPE\\_FILEOPENBUTTON](#)

[TYPE\\_FILESAVEBUTTON](#)

List Box Controls

[TYPE\\_NODELISTBOX](#)

[TYPE\\_INTLISTBOX](#)

[TYPE\\_FLOATLISTBOX](#)

[TYPE\\_STRINGLISTBOX](#)

[TYPE\\_POINT3LISTBOX](#)

**TYPE\_SPINNER:**

This is a standard MAX spinner control. It requires a spinner type, list of dialog item resource IDs and a display scale. The spinner type can be one of the **EditSpinnerType** values described below:

**EDITTYPE\_INT**

Any integer value.

**EDITTYPE\_FLOAT**

Any floating point value.

**EDITTYPE\_UNIVERSE**

This is a value in world space units. It respects the system's unit settings (for example feet and inches).

**EDITTYPE\_POS\_INT**

Any integer  $\geq 0$

**EDITTYPE\_POS\_FLOAT**

Any floating point value  $\geq 0.0$

**EDITTYPE\_POS\_UNIVERSE**

This is a positive value in world space units. It respects the system's unit settings (for example feet and inches) .

**EDITTYPE\_TIME**

This is a time value. It respects the system time settings (SMPTE for example).

The list of dialog item IDs depends on the ParamType of the parameter. For TYPE\_POINT3 and TYPE\_RGBA, you supply 3 pairs of IDs, one for

each coordinate, each pair specifying the editbox and spinner IDs. For the other you specify one editbox/spinner pair of IDs. The display scale can be a floating point value or the special value SPIN\_AUTOSCALE.

Eg:

**p\_ui, TYPE\_SPINNER, EDITTYPE\_UNIVERSE,  
IDC\_RADIUS, IDC\_RADSPINNER, SPIN\_AUTOSCALE,**

If the parameter is a table type then, as with all other control types, you must specify a fixed table size and supply a list of dialog item IDs, one set for each element in the tab. Eg, for a 3 element table:

**p\_ui, TYPE\_SPINNER, EDITTYPE\_UNIVERSE,  
IDC\_RADIUS1, IDC\_RADSPINNER1,  
IDC\_RADIUS2, IDC\_RADSPINNER2,  
IDC\_RADIUS3, IDC\_RADSPINNER3,  
SPIN\_AUTOSCALE,**

This control type can be used with any of the following ParamTypes:

**TYPE\_ANGLE, TYPE\_PCNT\_FRAC,  
TYPE\_COLOR\_CHANNEL, TYPE\_FLOAT,  
TYPE\_TIMEVALUE, TYPE\_INT, TYPE\_POINT3,  
TYPE\_RGBA**

#### **TYPE\_SLIDER:**

This is a standard 3ds max slider control. It requires a type, list of dialog item resource IDs and a number of ticks. The slider type can be one of the **EditSpinnerType** values described below:

**EDITTYPE\_INT**

Any integer value.

**EDITTYPE\_FLOAT**

Any floating point value.

**EDITTYPE\_UNIVERSE**

This is a value in world space units. It respects the system's unit settings (for example feet and inches).

**EDITTYPE\_POS\_INT**

Any integer  $\geq 0$

### **EDITTYPE\_POS\_FLOAT**

Any floating point value  $\geq 0.0$

### **EDITTYPE\_POS\_UNIVERSE**

This is a positive value in world space units. It respects the system's unit settings (for example feet and inches) .

### **EDITTYPE\_TIME**

This is a time value. It respects the system time settings (SMPTE for example).

The list of dialog item IDs depends on the ParamType of the parameter. For TYPE\_POINT3 and TYPE\_RGBA, you supply 3 pairs of IDs, one for each coordinate, each pair specifying the editbox and slider IDs. For the other you specify one editbox/slider pair of IDs. The segment count is the number of ticks in the slider.

Eg:

**p\_ui, TYPE\_SPINNER, EDITTYPE\_UNIVERSE,  
IDC\_EDITBOX, IDC\_SLIDER, numSegs**

For Point3 you can do the following:

**p\_ui, TYPE\_SPINNER, EDITTYPE\_UNIVERSE,  
IDC\_EDITBOX1, IDC\_SLIDER1, IDC\_EDITBOX2,  
IDC\_SLIDER2, IDC\_EDITBOX3, IDC\_SLIDER3, numSegs**

This control type can be used with any of the following ParamTypes:

**TYPE\_ANGLE, TYPE\_PCNT\_FRAC,  
TYPE\_COLOR\_CHANNEL, TYPE\_FLOAT,  
TYPE\_TIMEVALUE, TYPE\_INT, TYPE\_POINT3,  
TYPE\_RGBA**

### **TYPE\_RADIO:**

This is the standard Win32 radio button control. Following the **TYPE\_RADIO**, supply an int count of the number of radiobuttons in this group and then a list of dialog item IDs for each button. This can only be used with **TYPE\_INT** parameters. The value of the parameter defaults to the ordinal number of the radio button, starting at 0. You can

optionally supply a **p\_vals** tag immediately following the **TYPE\_RADIO p\_ui**, which should be followed by a list of numbers, one for each radio button. These numbers will become the (non-ordinal) parameter value corresponding to which button is set.

#### **TYPE\_CHECKBUTTON:**

This control functions just like **TYPE\_SINGLECHECKBOX** documented below but is represented by a button in either a pressed in or un-pressed state.

#### **TYPE\_SINGLECHECKBOX:**

This is the standard Win32 checkbox. Follow the **TYPE\_SINGLECHECKBOX** with the dialog item ID of the checkbox. This can only be used with **TYPE\_INT** or **TYPE\_BOOL** parameters.

#### **TYPE\_MULTICHECKBOX:**

This control type is not currently supported.

#### **TYPE\_COLORSWATCH:**

This is a 3ds max color-picker swatch. Follow the **TYPE\_COLORSWATCH** with the dialog item ID of the swatch custom control. This can only be used with **TYPE\_POINT3** or **TYPE\_RGBA** parameters.

#### **TYPE\_EDITBOX:**

This is a 3ds max custom editbox control. Follow the **TYPE\_EDITBOX** with the dialog item ID of the EditBox custom control. This can only be used with **TYPE\_STRING** and **TYPE\_FILENAMEAE** parameters.

#### **TYPE\_PICKNODEBUTTON:**

This is a 3ds max CustButton control used in a CBT\_CHECK mode with GREEN\_WASH highlight color, as per node picking button conventions. Follow the control type with the dialog item ID of the CustButton custom control. When the user presses this button a PickModeCallback command mode is entered and the user can pick a scene node under the filtering of any validation supplied (see tags **p\_validator**, **p\_classID** and **p\_sclassID**). This can only be used with **TYPE\_INODE** parameters. Use the **p\_prompt** tag to supply a status line prompt.

#### **TYPE\_TEXMAPBUTTON:**

This is a 3ds max CustButton control used in a CBT\_PUSH mode. Follow the control type with the dialog item ID of the CustButton custom control. This button throws up a Map selector dialog when pressed and is Map drag-and-drop sensitive. This can only be used with **TYPE\_TEXMAP** parameters. Use the **p\_prompt** tag to supply a status line prompt.

#### **TYPE\_MTLBUTTON:**

3ds max CustButton control used in a CBT\_PUSH mode. Follow the control type with the dialog item ID of the CustButton custom control. This button throws up a Material selector dialog when pressed and is Material drag-n-drop sensitive. This can only be used with **TYPE\_MTL** parameters. Use the **p\_prompt** tag to supply a status line prompt.

#### **TYPE\_BITMAPBUTTON:**

This is a 3ds max CustButton control used in a CBT\_PUSH mode. Follow the control type with the dialog item ID of the CustButton custom control. This button throws up a standard 3ds max Bitmap browser when pressed and is Bitmap drag-and-drop sensitive. Can only be used with **TYPE\_BITMAP** parameters. Use the **p\_prompt** tag to supply a status line prompt.

#### **TYPE\_FILEOPENBUTTON:**

This is a 3ds max CustButton control used in a CBT\_PUSH mode. Follow the control type with the dialog item ID of the CustButton custom control. This button throws up a standard Windows Open File dialog for selecting a file name. Can only be used with **TYPE\_STRING** and **TYPE\_FILENAME** parameters. Use any of the **p\_caption**, **p\_init\_file** and **p\_file\_types** tagsto further control the dialog.

#### **TYPE\_FILESAVEBUTTON:**

This is a 3ds max CustButton control used in a CBT\_PUSH mode. Follow control type with the dialog item ID of the CustButton custom control. This button throws up a standard Windows Save File dialog for selecting a file name. Can only be used with **TYPE\_STRING** and **TYPE\_FILENAME** parameters. Use any of the **p\_caption**, **p\_init\_file** and **p\_file\_types** tags to further control the dialog.

#### **TYPE\_INTLISTBOX:**

#### **TYPE\_FLOATLISTBOX:**

This specifies a series of controls for displaying and managing a ListBox

control containing an int or float Tab<> parameter. The setup consists of a ListBox control, 3 buttons for adding, replacing and deleting items in the list and a 3ds max spinner to supply source values for Add & Replace. After the control type, you supply 4 dialog item IDs. The first is the ListBox control, then dialog item IDs for 3 CustButton controls for an Add, Replace and Delete button, respectively. Follow these with a spinner type, editbox/spinner dialog item ID pair and a display scale, exactly as for TYPE\_SPINNER above. You can supply the value 0 for any of the Add, Replace, or Delete buttons if you don't need them in the dialog. This can only be used with the following parameter types:

For TYPE\_INTLISTBOX:

**TYPE\_TIMEVALUE\_TAB, TYPE\_INT, TYPE\_INT\_TAB**

For TYPE\_FLOATLISTBOX:

**TYPE\_ANGLE\_TAB, TYPE\_PCNT\_FRAC\_TAB,  
TYPE\_COLOR\_CHANNEL\_TAB, TYPE\_FLOAT\_TAB**

The Add/Replace/Delete buttons automatically keep the Tab<> parameter in step with the list.

The **TYPE\_INT** parameter type and **TYPE\_INTLISTBOX** control type combination is recognized specially and is used to allow a dropdown list to be associated with an int parameter such that the selection index in the dropdown becomes the integer parameter value. In this mode, after the control type you supply the ListBox control ID, then a count followed by that many string resource IDs. These strings are used to populate the dropdown.

Eg:

**p\_ui, TYPE\_INTLISTBOX, <list\_ctrl\_id>, <num\_items>, [  
<item1\_str\_id>, <item2\_str\_id>, ... ]**

These are the list control res ID, followed by a list of initial string items to load up given as a count (which can be 0 -- you can load them up dynamically in the dialog proc), and then a list of string resource IDs. See the **std2\_shader\_type** parameter in **\MAXSDK\SAMPLES\MATERIALS\STDMTL2.CPP** for an example.

**TYPE\_STRINGLISTBOX:**

A series a controls for displaying and managing a ListBox control

containing an string Tab<> parameter. The setup consists of a ListBox control, 3 buttons for adding, replacing and deleting items in the list and a 3ds max CustEdit box to supply source strings for Add & Replace. After the control type, you supply 4 dialog item IDs. The first is the ListBox control, then dialog item IDs for 3 CustButton controls for an Add, Replace and Delete button, respectively. Follow these with the CustEdit control dialog item ID. You can supply the value 0 for any of the Add, Replace, Delete buttons or CustEdit control IDs if you don't need them in the dialog. Can only use with TYPE\_STRING\_TAB parameters.

The Add/Replace/Delete buttons automatically keep the Tab<> parameter in step with the list.

#### **TYPE\_NODELISTBOX:**

A series a controls for displaying and managing a ListBox control containing an INode Tab<> parameter. The setup consists of a ListBox control, 3 buttons for picking, replacing and deleting items in the list. After the control type, you supply 4 dialog item IDs. The first is the ListBox control, then dialog item IDs for 3 CustButton controls for a Pick, Replace and Delete button, respectively. THE Pick and Replace buttons act exactly as TYPE\_PICKNODEBUTTONS to get nodes to add-to or replace-in the list. You can supply the value 0 for any of the Add, Replace, Delete buttons IDs if you don't need them in the dialog. Can only use with TYPE\_INODE\_TAB parameters.

The Pick/Replace/Delete buttons automatically keep the Tab<> parameter in step with the list, including managing References as needed.

#### **TYPE\_POINT3LISTBOX:**

A series a controls for displaying and managing a ListBox control containing an Point3 Tab<> parameter. The setup consists of a ListBox control, 3 buttons for adding, replacing and deleting items in the list, a source spinner type, three pairs of editboxes and spinners, and a display scale. After the control type, you supply 4 dialog item IDs. The first is the ListBox control, then dialog item IDs for 3 CustButton controls for an Add, Replace and Delete button, respectively. Follow this with the source spinner type, the editbox and spinner for the first item, the editbox and spinner for the second item, the editbox and spinner for the third item, and finally the display scale. Note that you can only use this with TYPE\_POINT3\_TAB parameters.

## **p\_uix**

This option is available in release 4.0 and later only.

This specifies which additional rollup/map the parameter is supposed to appear in.

## **p\_validator**

Specifies a validator object (see [Class PBValidator](#)). You supply a pointer to an instance of a class derived from **PBValidator**. This class has a **Validate()** method which can return TRUE if the **PB2Value** passed to it is valid and FALSE otherwise. This can be used for instance by a node pick button to check if a chosen node is acceptable.

Example:

**p\_validator, &fOutValidator,**

## **p\_accessor**

Specifies an accessor object (see [Class PBAccessor](#)). You supply a pointer to an instance of a class derived from **PBAccessor**. This class is used to provide a parameter Get/SetValue callback. The callback can be used to monitor parameter value changes, or to implement dynamically-computed virtual parameters. The class has two virtual methods, **Get()** and **Set()**, each given a **PB2Value&**, parameter ID, etc. In the case of a **Get()** you can modify the value in the **PB2Value&** to implement a virtual get.

Example:

**p\_accessor, &cmap\_accessor,**

## **p\_vals**

This defines radio button values in button order if button settings need to correspond to non-ordinal numbers. The value of the parameter defaults to the ordinal number of the radio button, starting at 0. You can optionally supply this tag immediately following the **TYPE\_RADIO p\_ui**. This tag should be followed by a list of numbers, one for each radio button. These numbers will become the (non-ordinal) parameter value corresponding to which button is set.

Example:

**p\_vals, 0,1,2,4,3 ,**

### **p\_refno**

This is used if the flag value in a <required\_param\_specs> includes **P\_OWNERS\_REF**. For these reftarg parameters this specifies the reference number in the block's owner for this reference. If the parameter is a Tab<>, then the reference number supplied is the base reference number of the 0'th element, with sequential reference numbers for the following elements.

Example:

**p\_refno, UVGEN\_REF,**

### **p\_subtexno**

For Texmap items in Mtls this defines the integer SubTexmap index for this Texmap in the owner Mtl. This is used by **TYPE\_TEXMAPBUTTON** ParamMap2 control types to give to the Material Editor for automatic button and Drag And Drop handling.

Example:

**p\_subtexno, 0,**

### **p\_submtlno**

Defines the sub-material integer index for this Mtl in an owner Mtl. This is used by **TYPE\_MTLBUTTON** ParamMap2 control types to give to the Material Editor for automatic button and Drag And Drop handling.

Example:

**p\_submtlno, 1,**

### **p\_dim**

This allows you to supply a dimension for this parameter. You specify a **ParamDimension\*** as the argument. Certain parameter types have an implied dimensions (see [List of ParamType Choices](#) for details). Defaults to **defaultDim**.

Example:

**p\_dim, stdWorldDim,**

### **p\_classID**

This specifies the class ID used as a validator for the various reftarg

parameters. This is used by the scripiter, picknode filter, etc. For example, if you supply this in a **TYPE\_INODE** parameter and use a **TYPE\_PICKNODE** parammap control, the picker uses this class ID in the picking filter. If you set **P\_CAN\_CONVERT** in the parameter's flag, it applies an **Object::CanConvertTo()** test to the node's world state using the Class\_ID supplied to perform the filter test.

### **p\_sclassID**

This specifies the super class ID used as a validator for the various reftarg parameters. For example, if you supply this in a **TYPE\_INODE** parameter and use a **TYPE\_PICKNODE** parammap control, the picker uses this super class ID in the picking filter.

Example:

**p\_sclassID, SHAPE\_CLASS\_ID,**

### **p\_enabled**

The associated UI controls are enabled by default.

supply TRUE or FALSE to indicate whether associated UI controls are enabled or disabled when the rollout dialog is first opened. If not supplied, defaults to enabled. Can be overridden by p\_enable\_ctrls below.

Example:

**p\_enabled, FALSE,**

### **p\_enable\_ctrls**

for TYPE\_BOOLs, lists which other params would be automatically UI enabled/disabled by this param

for TYPE\_BOOL parametes, lists which other params should be automatically UI enabled/disabled by changes in state of this parameter. This allows you to easily set up conditional enabling of other UI controls based on the state of the controlling parameter.

Example:

**p\_enable\_ctrls, 3, sel\_falloff, sel\_pinch, sel\_bubble,**

### **p\_prompt**

This sets the status line prompt string resource ID for various picker buttons.

Example:

**p\_prompt, IDS\_PICK\_CAM\_PROMPT,**

## **p\_caption**

This is a caption string resource ID for **TYPE\_FILEOPENBUTTON** or **TYPE\_FILESAVEBUTTON** open/save file dialogs.

## **p\_init\_file**

This establishes the initial filename for open/save file dlgs. Use a direct string for the argument, not a resource ID. The filename can be changed at runtime; do this by setting the `init_file` member of the `ParamDef` for the parameter, e.g.: **pbdesc->GetParamDef(file\_param).init\_file = new\_file\_name;**

## **p\_file\_types**

This is used by open/save file dialogs. The argument is a string resource ID. This string sets up the file type drop-down in the open/save dialog. It is in the following form:

**"<description1>|<pattern1>|<description2>|<pattern2>|...|"**

In other words, it is a sequence of file type descriptions and file type patterns each separated by a '|' vertical bar and terminated by a '|' vertical bar. For example:

**"Data(\*.dat)|\*.dat|Excel(\*.csv)|\*.csv|All|\*.\*|"**

specifies 3 types in the file type dropdown, the first reading "Data(\*.dat)" and matching \*.dat and the second reading "Excel(\*.csv)" and matching \*.csv and the third reading "All" and matching any file.

## **end**

Signals the end of the `<required_param_specs>` entry.

## Structure PB2Value

See Also: [Class IParamBlock2](#), [Class PBAccessor](#), [Class PBBitmap](#), [Class ReferenceTarget](#), [Class Control](#).

### Description:

This structure is available in release 3.0 and later only.

This structure holds the value in a ParamBlock2 or PBAccessor.

### typedef struct

{

**union**

{

**int i;**

This is used by: **TYPE\_INT**, **TYPE\_BOOL**.

**float f;**

This is used by: **TYPE\_FLOAT**, **TYPE\_ANGLE**,  
**TYPE\_PCNT\_FRAC**, **TYPE\_COLOR\_CHANNEL**.

**Point3\* p;**

This is used by: **TYPE\_POINT3**, **TYPE\_RGBA**.

**TimeValue t;**

This is used by **TYPE\_SPINNER** when the EditSpinnerType is  
**EDITTYPE\_TIME**.

**TCHAR\* s;**

This is used by **TYPE\_EDITBOX** or **TYPE\_STRING**.

**PBBitmap\* bm;**

This is used by **TYPE\_BITMAP**.

**ReferenceTarget\* r;**

A generic reference target pointer (**Mtl\***, **Texmap\***, **INode\***).

**Matrix3\* m;**

This is used by **TYPE\_MATRIX3**. This member is available in release  
4.0 and later only.

**Control\* control;**

This value replaces the **i,f,p** or **t** values if they are actually animated (and thus have a controller assigned).

**};**

**BYTE flags;**

These flags are for internal use only, do not alter them.

**} PB2Value;**

## Class ToolImageItem

See Also: [Class ToolItem](#).

```
class ToolImageItem : public ToolItem
```

### Description:

This class allows a developer to use an image in the toolbar. This is used internally as part of the object snap code. All methods of this class are implemented by the system.

### Methods:

### Prototype:

```
ToolImageItem(int w,int h,int k,int id, int  
y=CENTER_TOOL_VERTICALLY,DWORD hID=0);
```

### Remarks:

Constructor. The data members are initialized to the values passed. The type parameter of ToolItem is set to CTB\_IMAGE.

## List of Tool Item Types

See Also: [Class ToolItem](#).

One of the following values:

### **CTB\_PUSHBUTTON**

A push button. These buttons pop back out as soon as they are released by the user.

### **CTB\_CHECKBUTTON**

A check button. These buttons stay pressed in until the user presses them again.

### **CTB\_MACROBUTTON**

This option is available in release 3.0 and later only.

A macro button. These may contain icons or text.

### **CTB\_SEPARATOR**

A separator or spacer. This is used to separate groups of items in the toolbar.

### **CTB\_STATUS**

A status control. These may be used to display text.

### **CTB\_IMAGE**

An image control.

### **CTB\_OTHER**

A user defined tool type.

## Class MNFaceClusters

See Also: [Class MNTempData](#), [Class MNMesh](#)

class MNFaceClusters

### Description:

This class is available in release 4.0 and later only.

This class may be used for grouping faces in an **MNMesh** into clusters for applying transformations. Depending on the constructor used, it may group faces into clusters based on minimal angles between faces, on face selections, or on both. The class contains a list of face "clusters" for a given mesh. A typical application would be in Editable Poly, where the user has selected two separate groups of faces on different parts of the mesh and wants to extrude them both, or rotate both around their local centers. Each "cluster" is a contiguous group of selected faces. This class is only defined in relation to some **MNMesh**.

For convenient caching, it is recommended that you use this class through the **MNTempData** class.

All methods of this class are implemented by the system.

### Data Members:

public:

**Tab<int> clust;**

The cluster number, one for each face. Note that non-selected faces have UNDEFINED for their id.

**int count;**

The total number of clusters in the MNMesh.

### Methods:

public:

### Prototype:

**MNFaceClusters (MNMesh &mesh, DWORD clusterFlags);**

### Remarks:

Constructor.

This method will create face cluster lists based on the specified MNMesh.

Each contiguous group of selected faces is grouped into a cluster.

**Parameters:**

**MNMesh &mesh**

The mesh these clusters are based on.

**DWORD clusterFlags**

The face flags to cluster the faces by.

For instance, if this value was set to **MN\_SEL**, then faces would be clustered by their selection.

**Prototype:**

**MNFaceClusters (MNMesh & mesh, float angle, DWORD clusterFlags);**

**Remarks:**

This method is available in release 4.0 and later only.

This method will create face cluster lists based on the specified MNMesh. Cluster boundaries will be determined by the angle between faces and optionally by the face flags.

**Parameters:**

**MNMesh &mesh**

The mesh these clusters are based on.

**float angle**

The minimum edge angle (in radians) used to define a separation between clusters.

**DWORD clusterFlags**

The face flags to cluster the faces by.

For instance, if this value was set to **MN\_SEL**, then faces would be clustered by their selection. If this value is set to 0, then the clusters are based only on edge angles.

**Prototype:**

**int operator[](int i);**

**Remarks:**

Index operator for accessing cluster data.

**Default Implementation:**

```
{ return clust[i]; }
```

**Prototype:**

```
void MakeVertCluster(MNMesh &mesh, Tab<int> & vclust);
```

**Remarks:**

This method will create a table indicating which face cluster each vertex in the mesh is in.

**Parameters:****MNMesh &mesh**

The mesh this face cluster is based on.

**Tab<int> & vclust**

The table of vertex clusters. This is set to size **mesh.VNum()**. Values of UNDEFINED (0xffffffff) in the table indicate that a vertex is not in any cluster. If a vertex is in two clusters (because it's a point where corners of two clusters touch), the higher-indexed face's cluster is dominant.

**Prototype:**

```
void GetNormalsCenters (MNMesh &mesh, Tab<Point3> & norm,  
Tab<Point3> & ctr);
```

**Remarks:**

Computes average normals and centers for each of the face clusters.

**Parameters:****MNMesh &mesh**

The mesh this face cluster is based on.

**Tab<Point3> & norm**

The tables where the normals should be put. Each of these tables has its size set to the number of clusters, and is indexed by cluster.

**Tab<Point3> & ctr**

The tables where the centers should be put. Each of these tables has its size set to the number of clusters, and is indexed by cluster.

**Prototype:**

```
void GetBorder (MNMesh &mesh, int clustID, Tab<int> &  
cbord);
```

**Remarks:**

This method will find the edge list that borders this cluster. This edge list is a set of closed loops of edges, which may be empty. For instance, if the mesh is a sphere, and all the faces are in the cluster, there are no border edges for the cluster. But if one horizontal row of faces, such as the faces just above the equator, are in the cluster, then the edges above those faces form one loop, while the edges below form another.

**Parameters:**

**MNMesh &mesh**

The mesh this face cluster is based on.

**int clustID**

The ID of the cluster we want to get the border of.

**Tab<int> &cbord**

The table for putting the border output. This table is set up as follows: each border loop is represented by a series of edge indices, followed by a -1 to indicate a separation between loops. So a result of size 10 with data 1, 4, 6, 9, -1, 2, 10, 15, 14, -1, would indicate two border loops consisting of four edges each. The order of the edges in the loops is chosen so that as you look from outside the mesh and follow the path of the edges, the face cluster will always be on the left.

**Prototype:**

```
void GetOutlineVectors (MNMesh & m, Tab<Point3> &cnorms,  
Tab<Point3> &odir);
```

**Remarks:**

This method will retrieve the "outline" direction for the border of the cluster. This is the direction used in the "Outline" feature in Editable Poly face level, as well as in the Bevel command mode.

**Parameters:**

**MNMesh &m**

The mesh this face cluster is based on.

### **Tab<Point3> & cnorms**

The cluster normals, as computed in the GetNormalsCenters method. (This data is input, not output.)

### **Tab<Point3> & odir**

This is where the outline vectors are stored. This table is set to size **mesh.VNum()**, and stores one direction vector for each vertex. Most direction vectors are usually zero, since most vertices are not on the cluster's border. All vectors are scaled so that moving along them moves the cluster's border edges by one unit. (For instance, the length of a vector at a right angle between two border edges would be **sqrt(2)**, so that each edge can move by 1 unit "out" from the cluster.)

## Class MNEdgeClusters

See Also: [Class MNTempData](#), [Class MNMesh](#)

class MNEdgeClusters

### Description:

This class is available in release 4.0 and later only.

This class represents a list of edge "clusters" for a given **MNMesh**. A typical application would be in Editable Poly, where the user has selected a two separate groups of edges on different parts of the mesh and wants to rotate both around their local centers. Each "cluster" is a contiguous group of selected edges (ie they all touch each other). This class is only defined in relation to some **MNMesh**.

For convenient caching, it is recommended that you use this class through the **MNTempData** class.

### Data Members:

public:

**Tab<int> clust;**

The cluster IDs of all the edges – this table has size **MNMesh::nume**. **clust[i]** is UNDEFINED if edge i is not in any cluster.

**int count;**

The total number of clusters.

### Methods:

public:

### Prototype:

**MNEdgeClusters (MNMesh &mesh, DWORD clusterFlags);**

### Remarks:

Constructor.

### Parameters:

**MNMesh &m**

The mesh these clusters are based on.

**DWORD clusterFlags**

The edge flags to cluster the edges by.

For instance, if this value was set to **MN\_SEL**, then edges would be clustered by their selection.

**Prototype:**

```
int operator[](int i);
```

**Remarks:**

Index operator to access cluster data.

**Default Implementation:**

```
{ return clust[i]; }
```

**Prototype:**

```
void MakeVertCluster (MNMesh &mesh, Tab<int> & vclust);
```

**Remarks:**

This method will create a list of cluster IDs for vertices.

**Parameters:**

**MNMesh &m**

The mesh associated with these **MNEdgeClusters**.

**Tab<int> & vclust**

This is where the output goes: **vclust** is set to size **MNMesh::numv**, and the value of each entry in this table tells which cluster the vertex has been assigned to, based on the edges using it. If vertex "v" is not in any clusters (i.e. none of the edges that use it are in any clusters), **vclust[v]** is UNDEFINED.

**Prototype:**

```
void GetNormalsCenters (MNMesh &mesh, Tab<Point3> & norm,  
Tab<Point3> & ctr);
```

**Remarks:**

This method extracts normal and center information for each of the edge clusters.

**Parameters:**

## **MNMesh & m**

The mesh associated with these **MNEdgeClusters**.

## **Tab<Point3> & norm**

This table has its size set to the number of clusters in the cluster list. Normals are computed as the normalized average of the normal vectors of all edges in the cluster.

## **Tab<Point3> & ctr**

This table has its size set to the number of clusters in the cluster list. Centers are the average location of the edge centers -- thus a point on three edges in the same cluster has more weight than a point on only one edge in the cluster.

## Class MNFaceElement

See Also: [Class MNTempData](#), [Class MNMesh](#)

class MNFaceElement

### Description:

This class is available in release 4.0 and later only.

This class is used to assist in the process of sorting **MNMesh** faces into separate elements.

For convenient caching, it is recommended that you use this class through the **MNTempData** class.

### Data Members:

public:

**Tab<int> elem;**

The list indicating which element each face is in. The size is the number of faces in the associated MNMesh.

**int count;**

The total number of elements in the associated MNMesh.

### Methods:

public:

### Prototype:

**MNFaceElement (MNMesh &mesh);**

### Remarks:

Constructor.

This method will create an element list based on the specified MNMesh.

### Parameters:

**MNMesh &mesh**

A reference to the MNMesh for which to create the element list.

### Prototype:

**int operator[] (int i);**

**Remarks:**

Index operator for accessing elements.

**Default Implementation:**

```
{ return elem[i]; }
```

## Class PatchSubHitRec

See Also: [Class PatchMesh](#).

class PatchSubHitRec

### Description:

This class represents a single hit record for sub-patch level hit testing. All methods of this class are implemented by the system.

### Data Members:

public:

#### **DWORD dist;**

The distance of the hit. If the user is in wireframe mode, this is the distance in pixels to the item that was hit. If the user is in shaded mode, this is the Z depth distance. Smaller numbers indicate a closer hit.

#### **PatchMesh \*patch;**

The PatchMesh associated with this sub-patch hit.

#### **int index;**

The index of the sub-object component. For example, if vertices were being hit tested, this would be the index into the vertex table.

#### **int type;**

The type of the hit. One of the following values:

**PATCH\_HIT\_PATCH**  
**PATCH\_HIT\_EDGE**  
**PATCH\_HIT\_VERTEX**  
**PATCH\_HIT\_VECTOR**  
**PATCH\_HIT\_INTERIOR**

### Methods:

### Prototype:

**PatchSubHitRec(DWORD dist, PatchMesh \*patch, int index,  
int type, PatchSubHitRec \*next);**

### Remarks:

Constructor. The data members are set to the values passed.

**Prototype:**

**PatchSubHitRec \*Next();**

**Remarks:**

Returns the next sub hit record.

## Class IObject

See Also: [Class BaseInterfaceServer](#), [Class FPInterface](#), [Class FPInterfaceDesc](#), [Function Publishing](#), [Class InterfaceServer](#),

```
class IObject : public BaseInterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This is the base class used to publish functions from those objects not derived from Animatable. A developer inherits from this class and implements the methods of this class to provide information about the interfaces published by the class.

There is a corresponding **ParamType2** type code, **TYPE\_IOBJECT**, that allows instances of these classes to be passed and returned in FPInterface methods. This provides a simple form of user-defined type, in the sense that these instance collections are passed as interfaces rather than pointers.

MAXScript has wrapper value classes for IObjects and so this mechanism provides a light-weight alternative to the MAXScript SDK facilities for adding new wrapper value classes to the scripter.

MAXScript also calls the **AcquireInterface()** and **ReleaseInterface()** methods on IObjects as it creates and collects these wrappers, so that IObject objects can keep track of MAXScript's extant references to them.

### Methods:

public:

### Prototype:

```
virtual TCHAR* GetIObjectName() = 0;
```

### Remarks:

Returns the object/class name.

### Default Implementation:

```
{ return _T(""); }
```

### Prototype:

```
virtual int NumInterfaces() const = 0;
```

**Remarks:**

Returns the number of interfaces published by this object.

**Default Implementation:**

```
{ return 0; }
```

**Prototype:**

```
virtual BaseInterface *GetInterfaceAt(int i) const = 0;
```

**Remarks:**

Returns a pointer to the 'i-th' interface.

**Parameters:**

**int i**

The zero based index of the interface to return.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual BaseInterface* GetInterface(Interface_ID id) = 0;
```

**Remarks:**

Returns a pointer to the interface whose ID is specified.

**Parameters:**

**Interface\_ID id**

The ID of the interface to return.

**Default Implementation:**

```
{ return NULL; }
```

**Prototype:**

```
virtual void AcquireIObject();
```

**Remarks:**

This method is called when MAXScript makes a reference to this object. This is part of the IObject reference management and can be implemented by dynamically allocated IObjects for ref-count based lifetime control.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void ReleaseIObject();
```

**Remarks:**

This method is called when MAXScript deletes a reference to this object. This is part of the IObject reference management and can be implemented by dynamically allocated IObjects for ref-count based lifetime control.

**Default Implementation:**

```
{ }
```

**Prototype:**

```
virtual void DeleteIObject();
```

**Remarks:**

This method is the virtual destructor for the IObject.

**Default Implementation:**

```
{ }
```

## Class IIRenderMgrSelector

See Also: [Class InterfaceServer](#), [Class IInteractiveRender](#) , [Class InterfaceServer](#), [Class INode](#)

```
class IIRenderMgrSelector : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

This class represents the abstract (interface) for an interactive rendering manager selector to assist in the process of determining the nodes that are selected by the interactive rendering manager.

### Methods:

public:

### Prototype:

```
virtual BOOL IsSelected(INode* pINode);
```

### Remarks:

This method returns whether the specified node is selected. By default all nodes are selected and should therefore be shaded.

### Parameters:

**INode\* pINode**

The node to test.

### Default Implementation:

```
{ return TRUE; }
```

## Class InterfaceNotifyCallback

See Also: [Class BaseInterface](#), [Function Publishing System](#).

class InterfaceNotifyCallback

### Description:

This class is available in release 4.0 and later only.

This class provides a callback mechanism which can be registered with an interface on Acquire() so that it can be notified when the interface goes away as the server controls the lifetime.

### Methods:

public:

### Prototype:

**virtual void InterfaceDeleted(BaseInterface\* bi);**

### Remarks:

This method gets called to notify the server is deleting the interface.

### Parameters:

**BaseInterface\* bi**

A pointer to the appropriate BaseInterface.

### Default Implementation:

{ }

### Prototype:

**virtual BaseInterface\* GetInterface(Interface\_ID id);**

### Remarks:

Returns a pointer to the interface whose ID is specified, for future notification extensions.

### Parameters:

**Interface\_ID id**

The ID of the interface to return.

### Return Value:

```
{ return NULL; }
```

# Class IMenuItemElement

See Also: [Class IMenuItem](#), [Class IPoint2](#), [Class Box2](#)

class IMenuItemElement

## Description:

This class is available in release 4.0 and later only.

This abstract class represents an interface for any menu element. Methods that are marked as internal should not be used.

## Methods:

public:

## Prototype:

```
virtual void SetOrigin(const IPoint2& origin, OriginLocation location) = 0;
```

## Remarks:

This method is used internally.

This method allows you to set the element's origin and origin location.

## Parameters:

**IPoint2& origin**

The x, y coordinates of the origin.

**OriginLocation location**

The origin location, either one of; **UPPER\_LEFT**, **LOWER\_LEFT**, **LOWER\_RIGHT**, or **UPPER\_RIGHT**.

## Prototype:

```
virtual const IPoint2& GetOrigin() const = 0;
```

## Remarks:

This method is used internally.

This method returns the x, y coordinates of the element's origin.

## Prototype:

**virtual void SetVisible(bool visible) = 0;**

**Remarks:**

This method allows you to set the visibility of the element.

**Parameters:**

**bool visible**

TRUE for visible, FALSE for invisible.

**Prototype:**

**virtual bool GetVisible() = 0;**

**Remarks:**

This method returns the visibility of the element. TRUE if the element is visible, otherwise FALSE.

**Prototype:**

**virtual void SetTitle(const TCHAR \*customTitle) = 0;**

**Remarks:**

This method allows you to set the item's title.

**Parameters:**

**TCHAR \*customTitle**

The title string.

**Prototype:**

**virtual const TSTR& GetTitle() = 0;**

**Remarks:**

This method returns the item's title string.

**Prototype:**

**virtual void SetEnabled(bool enabled) = 0;**

**Remarks:**

This method allows you to enable and disable the element.

**Parameters:**

**bool enabled**

TRUE to enable, FALSE to disable.

**Prototype:**

**virtual bool GetEnabled() = 0;**

**Remarks:**

This method returns the state of the element. TRUE if it's enabled, FALSE if it's disabled.

**Prototype:**

**virtual const IPoint2& GetSize() = 0;**

**Remarks:**

This method is used internally.

This method returns the element's size in the menu's coordinate space.

**Prototype:**

**virtual const Box2& GetRect() = 0;**

**Remarks:**

This method is used internally.

This method returns the element's rectangle size in the menu's coordinate space.

**Prototype:**

**virtual bool IsInRect(const IPoint2& point) = 0;**

**Remarks:**

This method is used internally.

This method determines if a specific point is inside the element's rectangle.

**Parameters:**

**IPoint2& point**

The point to test.

**Return Value:**

TRUE if the point is inside the rectangle, otherwise FALSE.

## Class IMenuItem

See Also: [Class IMenuItemElement](#), [Class IMenuItemGlobalContext](#), [Class IMenuItemLocalContext](#), [Class ActionItem](#), [Class IMenuItem](#), [Class MaxIcon](#) , [Class IQuadMenuSettings](#)

class IMenuItem : public IMenuItemElement, public FPMixinInterface

### Description:

This class is available in release 4.0 and later only.

This abstract class represents an interface for a menu item. Methods that are marked as internal should not be used.

### Methods:

public:

### Prototype:

```
virtual void SetMenuItemLocalContext(IMenuItemLocalContext*  
pMenuItemLocalContext) = 0;
```

### Remarks:

This method is used internally.

This method allows you to set a new (local) context for the menu, invalidating the menu's cache.

### Parameters:

**IMenuItemLocalContext\* pMenuItemLocalContext**

A pointer to the new local context object you wish to set.

### Prototype:

```
virtual ActionMode GetActionMode() const = 0;
```

### Remarks:

This method returns the current action mode. When item selection instigates an action item, calls functions, or displays a submenu the action mode changes to indicate the exact state the system is in.

### Return Value:

Either of the following; **AM\_INACTIVE**, **AM\_SEPARATOR**,

**AM\_ITEM, AM\_FN, AM\_SUBMENU, AM\_ITEM\_SUBMENU**

**Prototype:**

**virtual bool ExecuteAction() const = 0;**

**Remarks:**

This method will execute the current action.

**Return Value:**

TRUE if the action was executed successfully, otherwise FALSE.

**Prototype:**

**virtual void ActAsSeparator() = 0;**

**Remarks:**

This method allows you to make the item act as an item separator.

**Prototype:**

**virtual bool IsSeparator() const = 0;**

**Remarks:**

This method determines if the item is acting as a separator (TRUE) or not (FALSE).

**Prototype:**

**virtual void SetActionItem(ActionItem\* pActionItem) = 0;**

**Remarks:**

This method allows you to set the current action item. Note that **GetActionItem()** returns NULL if the ActionMode is not **AM\_ITEM**.

**Parameters:**

**ActionItem\* pActionItem**

The action item you wish to set.

**Prototype:**

**virtual ActionItem\* GetActionItem() const = 0;**

**Remarks:**

This method returns a pointer to the current action item, or NULL if the ActionMode is not **AM\_ITEM**.

**Prototype:**

```
virtual void SetActionFn(ActionFn actionFn) = 0;
```

**Remarks:**

This method allows you to set the current action function. Note that **GetActionFn()** returns NULL if the ActionMode is not **AM\_FN**. Also note: **typedef void (\* ActionFn)(void);**

**Parameters:**

**ActionFn actionFn**

The action function you wish to set.

**Prototype:**

```
virtual const ActionFn GetActionFn() const = 0;
```

**Remarks:**

This method returns the current action function, or NULL if the ActionMode is not **AM\_FN**.

Note: **typedef void (\* ActionFn)(void);**

**Prototype:**

```
virtual void SetSubMenu(IMenu* menu) = 0;
```

**Remarks:**

This method allows you to set the submenu. Note that **GetSubMenu()** returns NULL if the ActionMode is not **AM\_SUBMENU**.

**Parameters:**

**IMenu\* menu**

The submenu you wish to set.

**Prototype:**

```
virtual IMenu* GetSubMenu() = 0;
```

**Remarks:**

This method returns a pointer to the submenu, or NULL if the ActionMode is not **AM\_SUBMENU**.

**Prototype:**

```
virtual void SetPreDisplayCB(PreDisplayCB preDisplayCB) = 0;
```

**Remarks:**

This method allows you to set the pre-display callback.

Note: **typedef void (\* PreDisplayCB)(IMenuItem& menuItem);**

**Parameters:**

**PreDisplayCB preDisplayCB**

The callback to set.

**Prototype:**

```
virtual const PreDisplayCB GetPreDisplayCB() const = 0;
```

**Remarks:**

This method returns the pre-display callback.

**Prototype:**

```
virtual void Display(bool leftToRight) = 0;
```

**Remarks:**

This method is used internally.

**Prototype:**

```
virtual TCHAR GetAccelerator() = 0;
```

**Remarks:**

This method is used internally.

This method returns the item's accelerator, or 0 if none is assigned.

**Prototype:**

```
virtual void SetIcon(MaxIcon* pMaxIcon) = 0;
```

**Remarks:**

This method allows you to set the item's icon.

**Parameters:**

**MaxIcon\* pMaxIcon**

A pointer to a MaxIcon to set.

**Prototype:**

**virtual const MaxIcon\* GetIcon() const = 0;**

**Remarks:**

This method returns a pointer to the item's icon.

**Prototype:**

**virtual void SetChecked(bool checked) = 0;**

**Remarks:**

This method allows you to set the checked state of the item.

**Parameters:**

**bool checked**

TRUE to check the item, FALSE to uncheck the item.

**Prototype:**

**virtual bool GetChecked() = 0;**

**Remarks:**

This method returns TRUE if the item is checked or FALSE if it is unchecked.

**Prototype:**

**virtual void SetHighlighted(bool highlighted) = 0;**

**Remarks:**

This method allows you to set the highlighted state of the item.

**Parameters:**

**bool highlighted**

TRUE to highlight the item, FALSE if you do not want to highlight the item.

**Prototype:**

**virtual bool GetHighlighted() const = 0;**

**Remarks:**

This method returns TRUE if the item is highlighted, otherwise FALSE.

**Prototype:**

**virtual void SetUseCustomTitle(bool useCustomTitle) = 0;**

**Remarks:**

This method allows you to tell the item it should use a custom title, which is set through **SetTitle()**.

**Parameters:**

**bool useCustomTitle**

TRUE to use a custom title, FALSE to use the default.

**Prototype:**

**virtual bool GetUseCustomTitle() const = 0;**

**Remarks:**

This method returns TRUE if the item is using a custom title or FALSE if it's using a default.

**Prototype:**

**virtual void SetDisplayFlat(bool displayFlat) = 0;**

**Remarks:**

This method allows you to set whether the submenu-item should be displayed 'flat'.

**Parameters:**

**bool displayFlat**

TRUE to set to flat, otherwise FALSE.

**Prototype:**

**virtual bool GetDisplayFlat() const = 0;**

**Remarks:**

This method returns TRUE if the submenu-item should be displayed 'flat', otherwise FALSE.

**Prototype:**

**virtual void PostMenuInteraction() = 0;**

**Remarks:**

This method is used internally.

This method is called after the user/menu interaction is done after which it will clear the caches for ActionItem handler values.

**The following functions are not part of the class but are available for use.**

**Function:**

**IMenuItem \* GetIMenuItem();**

**Remarks:**

This method will return a pointer to the IMenuItem.

**Prototype:**

**void ReleaseIMenuItem(IMenuItem \*);**

**Remarks:**

This method will release the specified IMenuItem.

**Parameters:**

**IMenuItem \***

A pointer to the IMenuItem you wish to release.

## Class IMenuGlobalContext

See Also: [Class IMenuItem](#), [Class IMenuSettings](#), [Class IMenuTimer](#), [Class ItemID](#), [Class IPoint2](#)

class IMenuGlobalContext

### Description:

This class is available in release 4.0 and later only.

This abstract class represents an global context interface for all menus that might be displayed during a user's menuing action and is used internally. Methods that are marked as internal should not be used.

### Methods:

public:

### Prototype:

```
virtual void SetIMenuSettings(IMenuSettings* pIMenuSettings) = 0;
```

### Remarks:

This method is used internally.

This method allows you to set the menu settings associated with this global context.

### Parameters:

**IMenuSettings\* pIMenuSettings**

A pointer to a menu settings object.

### Prototype:

```
virtual IMenuSettings* GetIMenuSettings() const = 0;
```

### Remarks:

This method is used internally.

This method returns a pointer to the menu settings associated with this global context.

### Prototype:

**virtual void UpdateCursorPosition() = 0;**

**Remarks:**

This method is used internally.

This method updates the cursor position from Win32.

**Prototype:**

**virtual const IPoint2& GetCursorPosition() const = 0;**

**Remarks:**

This method is used internally.

This method returns the cached cursor position.

**Return Value:**

The x and y coordinates of the cursor.

**Prototype:**

**virtual const IPoint2& GetInitialCursorPosition() const = 0;**

**Remarks:**

This method is used internally.

This method returns the cached initial cursor position (i.e. the coordinate where the user clicked).

**Return Value:**

The x and y coordinates of the cached initial cursor position.

**Prototype:**

**virtual void SetInitialCursorPosition(IPoint2& initPos) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the cached initial cursor position (i.e. the coordinate where the user clicked).

**Parameters:**

**IPoint2& initPos**

The x and y coordinate of the cursor position.

**Prototype:**

**virtual void SetIMenuTimer(IMenuTimer\* pIMenuTimer) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set a global timer for the menu's global context.

**Parameters:**

**IMenuTimer\* pIMenuTimer**

A pointer to the menu timer object,

**Prototype:**

**virtual IMenuTimer\* GetIMenuTimer() const = 0;**

**Remarks:**

This method is used internally.

This method returns a pointer to the global timer for the menu's global context.

**Prototype:**

**virtual void SetHDisplayWnd(HWND hDisplayWnd) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the handle to the display window.

**Parameters:**

**HWND hDisplayWnd**

The handle to the display window.

**Prototype:**

**virtual HWND GetHDisplayWnd() const = 0;**

**Remarks:**

This method is used internally.

This method returns the handle to the display window.

**Prototype:**

**virtual void SetHMessageWnd(HWND hDisplayWnd) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the handle to the message window

**Parameters:**

**HWND hDisplayWnd**

The handle to the message window.

**Prototype:**

**virtual HWND GetHMessageWnd() const = 0;**

**Remarks:**

This method is used internally.

This method returns the handle to the message window.

**Prototype:**

**virtual void SetHDisplayDC(HDC hDisplayDC) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set a handle to the display device context.

**Parameters:**

**HDC hDisplayDC**

The handle to the display device context.

**Prototype:**

**virtual HDC GetHDisplayDC() const = 0;**

**Remarks:**

This method is used internally.

This method returns the handle to the display device context.

**Prototype:**

**virtual void SetTitleHFont(HFONT hTitleFont) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the handle to the title font.

**Parameters:**

**HFONT hTitleFont**

The handle to the title font.

**Prototype:**

**virtual HFONT GetTitleHFont() const = 0;**

**Remarks:**

This method is used internally.

This method returns the handle to the title font.

**Prototype:**

**virtual void SetItemHFont(HFONT hItemFont) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the handle to the item font.

**Parameters:**

**HFONT hItemFont**

The handle to the item font.

**Prototype:**

**virtual HFONT GetItemHFont() const = 0;**

**Remarks:**

This method is used internally.

This method returns the handle to the item font.

**Prototype:**

**virtual void SetAcceleratorHFont(HFONT hItemFont) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the handle to the accelerator font.

**Parameters:**

**HFONT hItemFont**

The handle to the accelerator font.

**Prototype:**

**virtual HFONT GetAcceleratorHFont() const = 0;**

**Remarks:**

This method is used internally.

This method returns the handle to the accelerator font.

**Prototype:**

**virtual void SetUniformItemSize(const IPoint2& itemSize) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the menu's maximum uniform item size.

**Parameters:**

**IPoint2& itemSize**

The size rectangle.

**Prototype:**

**virtual const IPoint2& GetUniformItemSize() const = 0;**

**Remarks:**

This method is used internally.

This method returns the menu's maximum uniform item size as a rectangle.

**Prototype:**

**virtual int GetTitleBarHeight() = 0;**

**Remarks:**

This method is used internally.

This method returns the height of the title bar, not counting the border.

**Prototype:**

**virtual ItemID& GetCurrentItemID() = 0;**

**Remarks:**

This method is used internally.

This method returns the ItemID of the menu/item triplet that's currently being traversed.

**Prototype:**

**virtual ItemID& GetSelectedItemID() = 0;**

**Remarks:**

This method is used internally.

This method returns the ItemID of the menu/item triplet that's currently selected.

**Prototype:**

**virtual bool HasSelection() = 0;**

**Remarks:**

This method is used internally.

This method will determine selection is available and will return TRUE if selection is available or FALSE if it is not.

**Prototype:**

**virtual bool IsCurrentMenuSelected() = 0;;**

**Remarks:**

This method is used internally.

This method will determine the selection status of the current menu and will return TRUE if the current menu is selected or FALSE if it is not.

**Prototype:**

**virtual bool IsCurrentItemSelected() = 0;**

**Remarks:**

This method is used internally.

This method will determine the selection status of the current menu and item and will return TRUE if the current menu and item are selected or FALSE if they are not.

**Prototype:**

**virtual void SelectCurrentItem() = 0;**

**Remarks:**

This method is used internally.

This method selects the current item.

## Class IMenuLocalContext

See Also: [Class IMenuItem](#), [Class IMenuColors](#), [Class IPoint2](#)

class IMenuLocalContext

### Description:

This class is available in release 4.0 and later only.

This abstract class represents a local context interface for a specific menu and is used internally. Methods that are marked as internal should not be used.

### Methods:

public:

### Prototype:

**virtual void SetHDrawDC(HDC hDrawDC) = 0;**

### Remarks:

This method is used internally.

This method allows you to set the handle to the drawing device context.

### Parameters:

**HDC hDrawDC**

The handle to the drawing device context.

### Prototype:

**virtual HDC GetHDrawDC() const = 0;**

### Remarks:

This method is used internally.

This method returns the handle to the drawing device context.

### Prototype:

**virtual void SetLocalCursorPosition(const IPoint2& localCursorPos) = 0;**

### Remarks:

This method is used internally.

This method allows you to set the cursor position in the local coordinate system of the menu.

**Parameters:**

**const IPoint2& localCursorPos**

The cursor position coordinates.

**Prototype:**

**virtual const IPoint2& GetLocalCursorPosition() const = 0;**

**Remarks:**

This method is used internally.

This method returns the cursor position in the local coordinate system of the menu.

**Prototype:**

**virtual void SetMenuItemWidth(int menuItemWidth) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the menu's current width.

**Parameters:**

**int menuItemWidth**

The menu width.

**Prototype:**

**virtual int GetMenuItemWidth() const = 0;**

**Remarks:**

This method is used internally.

This method returns the menu's current width.

**Prototype:**

**virtual void SetLevel(int level) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the menu's current level. Submenus are indexed by a level > 0.

**Parameters:**

**int level**

The current level to set.

**Prototype:**

**virtual int GetLevel() const = 0;**

**Remarks:**

This method is used internally.

This method returns the menu's current level.

**Prototype:**

**virtual void SetLastExecutedItemPath(Tab<IMenuItem \*>  
\*pExecutedItemPath) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the menu's last executed item path. The item path is a table of IMenuItem entries, listing the selected item at each menu level.

**Parameters:**

**Tab<IMenuItem \*> \*pExecutedItemPath**

A pointer to the item path.

**Prototype:**

**virtual Tab<IMenuItem \*> \*GetLastExecutedItemPath() = 0;**

**Remarks:**

This method is used internally.

This method returns the menu's last executed item path. The item path is a table of IMenuItem entries, listing the selected item at each menu level.

**Prototype:**

**virtual void SetMenuColors(const MenuColors \*pMenuColors) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the menu's current colors.

**Parameters:**

**MenuColors \*pMenuColors**

A pointer to the menu colors.

**Prototype:**

**virtual const MenuColors \*GetMenuColors() const = 0;**

**Remarks:**

This method is used internally.

This method returns a pointer to the menu's current colors.

**Prototype:**

**virtual void SetIMenuGlobalContext(IMenuGlobalContext\* pIMenuGlobalContext, int level, Tab<IMenuItem \*> \*pExecutedItemPath, const MenuColors \*pMenuColors) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the global menu context.

**Parameters:**

**IMenuGlobalContext\* pIMenuGlobalContext**

A pointer to the menu's global context object.

**int level**

The menu level.

**Tab<IMenuItem \*> \*pExecutedItemPath**

A pointer to the item path. The item path is a table of IMenuItem entries, listing the selected item at each menu level.

**MenuColors \*pMenuColors**

A pointer to the menu colors.

**Prototype:**

```
virtual IMenuGlobalContext* GetIMenuGlobalContext() const =  
0;
```

**Remarks:**

This method is used internally.

This method returns a pointer to the menu's global context object.

## Structure MenuEvent

See Also: [Class IMenu](#), [Class Interface](#).

Structure MenuEvent

### Description:

This structure is available in release 4.0 and later only.

This structure is used internally.

**struct MenuEvent**

```
{  
    Event mEvent;  
        EVENT_BEGIN_TRACK  
        EVENT_CURSOR_DOWN  
        EVENT_RIGHT_CURSOR_DOWN  
        EVENT_CURSOR_MOVED  
        EVENT_CURSOR_UP  
        EVENT_END_TRACK  
        EVENT_KEY  
        EVENT_RIGHT_CURSOR_UP  
    unsigned int mEventParam;  
        EP_NULL  
        EP_SHOW_SUBMENU  
        EP_HIDE_SUBMENU  
};
```

## Class IMenuContext

See Also: [Class InterfaceServer](#), [Class IMenuBarContext](#), [Class Interface](#).

class IMenuContext : public InterfaceServer

### Description:

This class is available in release 4.0 and later only.

This class represents the interface to the various menu types such as MenuBar, QuadMenus, and right-click Popup Menus. The class provides the fundamental basis for the ID, Type, and Name.

### Methods:

public:

### Prototype:

**virtual MenuContextId GetId() = 0;**

### Remarks:

This method returns the Menu Context ID.

### Prototype:

**virtual MenuContextType GetType() = 0;**

### Remarks:

This method returns the Menu Context Type. One of the following values:

**kMenuContextMenuBar**

The context for the main menu bar.

**kMenuContextQuadMenu**

The context for the quad menu.

**kMenuContextPopupMenu**

The context for the Ctrl-right click popup menu.

### Prototype:

**virtual TSTR& GetName() = 0;**

### Remarks:

This method returns the name of the Menu Context.

## Class DropType

See Also: [Class DropClipFormat](#), [Class FileDropType](#), [Class DropScriptDropType](#) , [List of DropTypes](#)

```
class DropType : public IDropSource, public IDataObject
```

### Description:

This class is available in release 4.0 and later only.

This is the base class for droppable content types. Distinguished instances of subclasses represent different types of drop content, such as a file distinguished by file suffix or a scene object. The active **DropClipFormat** parses dropped **IDataObject** into one of these dropped types, filling its data members with appropriate guff from the data object.

Each **DropClipFormat** can encompass multiple possible types of dropped data. For example, the **iDrop** package can drop max files, image files, script files, etc. The **DropClipFormat** classes parse raw dropped clipboard data into one of the **DropType** family of classes. These contain recognizer predicates, extracted data from the current drop, and utilities for working with the data, such as URL downloads, script compile & execution, etc. Custom drop-types can be created by subclassing one of the **DropType** base classes. The built-in **DropTypes** are listed in the [List of DropTypes](#)

### Data Members:

protected:

```
static IDragAndDropMgr* dndMgr;
```

Cached pointer to the DnD manager.

```
static bool dragging;
```

The drop source state.

```
static POINT startPt;
```

The drag and drop starting point.

```
static WPARAM startKeyState;
```

The starting state of the keyboard.

```
static HWND startWnd;
```

The handle to the starting window.

```
static bool loaded;
```

Flags if the current packages is already downloaded.

public:

**static IDataObject\* current\_dataobject;**

Currently dropping **IDataObject**. Filled in by the low-level **DragEnter()** code.

**Methods:**

public:

**Prototype:**

**DropType();**

**Remarks:**

Constructor.

**Prototype:**

**static void Init();**

**Remarks:**

This method clears the currently-parsed drop data.

**Default Implementation:**

**{ current\_dataobject = NULL; loaded = false; }**

**Prototype:**

**virtual int TypeCode()=0;**

**Remarks:**

This method returns the typecode of the DropType.

**Prototype:**

**virtual bool IsDropType(int code);**

**Remarks:**

This method returns TRUE if the DropType is of the specified DropType code, otherwise FALSE.

Each **DropType** subclass is given a unique integer code that can be used for

type-testing, switching, etc. The **TypeCode()** method must be implemented to return this code and **isDropType()** to test against the given code (this is provided to that intermediate **DropType** base classes with codes can effectively support superclass testing). The codes for the built-in **DropTypes** are given by the following defined symbols which are listed in the [List of DropTypes](#)

**Parameters:**

**int code**

The DropType code.

**Default Implementation:**

```
{ return code == TypeCode(); }
```

**Prototype:**

```
virtual bool Load(bool showProgress = true);
```

**Remarks:**

Subclasses should implement this method if they need to perform any droptype-specific loading prior to clipformat data use. For example, the URL package types all download any web-resident files in this method. Control any implemented progress dialog with the showProgress parameter.

**Parameters:**

**bool showProgress = true**

The download progress dialog can be displayed by passing true.

**Default Implementation:**

```
{ return true; }
```

**Prototype:**

```
virtual DWORD DropEffect();
```

**Remarks:**

This method returns the dropeffect currently supported by the accepted dropping type.

**Default Implementation:**

```
{ return DROPEFFECT_MOVE; }
```

## Structure Color24

See Also: [Structure Color48](#), [Structure Color64](#).

Note: **typedef uchar UBYTE;**

```
struct Color24 {
```

```
    uchar r,g,b;
```

8 bits for each of the Red, Green and Blue components.

```
};
```

## Class MNNormalFace

class MNNormalFace

### Description:

This class is available in release 5.0 and later only.

This class is a face used to store specified normal information for a given face. An array of these faces is used as a data member of class MNNormalSpec, in much the same way as an MNMapFace array is a data member of class MNMap.

.

### Methods:

### Prototype:

**MNNormalFace ()**

### Remarks:

Constructor - initializes mDegree to 0, mpNormalID to NULL.

### Prototype:

**MNNormalFace (int degree)**

### Remarks:

Constructor - initializes to the degree specified.

.

### Prototype:

**~MNNormalFace ()**

### Remarks:

Destructor - frees data members.

**Prototype:**  
**void Init();**

**Remarks:**

Initializes data members. This is useful in situations where the default constructor may not have been properly called, such as  
MNNormalFace \*fc = new MNNormalFace[10];

It should not be called on an MNNormalFace with data allocated, or that memory will leak.

**Prototype:**  
**void Clear();**

**Remarks:**

Frees all allocated data, sets degree to 0.

**Prototype:**  
**int GetDegree();**

**Remarks:**

Returns the face's degree.

**Prototype:**  
**int SetDegree (int degree);**

**Remarks:**

Sets the face's degree.

**Prototype:**

**int GetNormalID(int corner);**

**Remarks:**

Accessor for normal in a particular corner of the face.

**Parameters:**

**int corner**

The (zero-based) index of the corner of the face. (A quad face has corners 0, 1, 2, and 3.)

**Return Value:**

Index of normal (in parent MNNormalSpec's normal array), or -1 if "corner" is out of range.

**Prototype:**

**int SetNormalID (int corner, int norm);**

**Remarks:**

Sets the normal ID used in a particular corner.

**Parameters:**

**int corner**

The (zero-based) index of the corner of the face. (A quad face has corners 0, 1, 2, and 3.)

**int norm**

The index of the normal (in the parent MNNormalSpec's normal array).

**Prototype:**

**bool GetSpecified (int corner)**

**Remarks:**

Indicates whether the normal used in a particular corner is specified or not.

**Parameters:****int corner**

The (zero-based) index of the corner of the face. (A quad face has corners 0, 1, 2, and 3.)

**Prototype:**

**void SetSpecified (int corner, bool value=true);**

**Remarks:**

Controls whether the normal used in a particular corner of the face is specified or not.

**Parameters:****int corner**

The (zero-based) index of the corner of the face. (A quad face has corners 0, 1, 2, and 3.)

**Parameters:****bool value**

Whether the corner should have a specified (true) or unspecified (false) normal.

**Prototype:**

**void SpecifyNormalID (int corner, int norm);**

**Remarks:**

Specifies that a particular corner of the face should use a given normal index.

**Parameters:**

**int corner**

The (zero-based) index of the corner of the face. (A quad face has corners 0, 1, 2, and 3.)

**Parameters:**

**int norm**

The index of the normal (in the parent MNNormalSpec's normal array).

**Prototype:**

**void Insert (int pos, int num=1);**

**Remarks:**

Inserts space for more normals into an existing face. (Designed to be used in conjunction with MNFace::Insert, to increase the size of a polygon and add new vertices & normals.)

.

**Parameters:**

**int pos**

The location within the face where the new normals will be added.

**Parameters:**

**int num=1**

The number of new normals to add.

**Prototype:**

**void Delete (int pos, int num=1);**

**Remarks:**

Deletes normals from this map face. (Designed to be used in conjunction with MNFace::Delete, to decrease the size of a polygon and remove vertices & normals.).

**Parameters:****int pos**

The location within the face where the new normals will be deleted.

**Parameters:****int num=1**

The number of new normals to delete.

**Prototype:****void RotateStart (int newstart);****Remarks:**

Deletes normals from this map face. (Designed to be used in conjunction with MNFace::Delete, to decrease the size of a polygon and remove vertices & normals.).

**Parameters:****int newstart**

The new first normal

**Prototype:****void Flip ();****Remarks:**

Reverses order of normals, effectively inverting the face. (Designed

to be used in conjunction with MNFace::Flip.)

**Prototype:**

**MNNormalFace & operator= (const MNNormalFace & from);**

**Remarks:**

Typical = operator - calls SetDegree to make this face the same size as "from", then copies the specification data and the normalIDs.

**Prototype:**

**MNNormalFace & operator= (const MNFace & from);**

**Remarks:**

Sets the degree of this NormalFace to that of the MNFace given..

**Prototype:**

**void ShallowTransfer (MNNormalFace & from);**

**Remarks:**

"Shallow-copies" data from "from". This is dangerous to use - the pointer to the normal ID array is the same in both faces after this is called. It's typically used by MNNormal::FAlloc, when resizing the whole face array.

**Prototype:**

**void MNDebugPrint (bool printAll=false);**

**Remarks:**

Uses "DebugPrint" to output information about this MNNormalFace to the Debug buffer in DevStudio. Output is formatted as follows: suppose we have a 5-sided face, with normals {4,5,6,7,8}, and only the normals in corners 1 and 2 (i.e. normals 5 and 6) are specified.

MNDebugPrint (true) would generate:

```
_4 5 6 _7 _8
```

MNDebugPrint (false) would generate:

```
_ 5 6 _ _
```

This is mainly used as part of MNNormalSpec::MNDebugPrint.

**Prototype:**

**IOResult Save (ISave \*isave);**

**Remarks:**

Called by the system. Saves the face's data to the stream given.

**Prototype:**

**IOResult Load (ILoad \*iload);**

**Remarks:**

Called by the system. Loads the face's data from the stream given.

## Structure ConfigurationBlock

See Also: [Class MaxNetManager](#), [Structure ManagerInfo](#), [Structure ServerInfo](#), [Structure ClientInfo](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about the system.

### typedef struct {

**DWORD dwTotalPhys;**

The system's total physical memory, retrieved through **GlobalMemoryStatus()**.

**DWORD dwNumberOfProcessors;**

The number of processors in the system, retrieved through **GetSystemInfo()**.

**DWORD dwMajorVersion;**

The system's major version, retrieved through **GetVersionEx()**.

**DWORD dwMinorVersion;**

The system's minor version, retrieved through **GetVersionEx()**.

**DWORD dwBuildNumber;**

The system's build number, retrieved through **GetVersionEx()**.

**DWORD dwPlatformId;**

The system's platform ID, retrieved through **GetVersionEx()**.

**TCHAR szCSDVersion[128];**

The system's CSD version, retrieved through **GetVersionEx()**.

**char user[MAX\_PATH];**

The logged in user's name, retrieved through **GetUserName()**.

**char tempdir[MAX\_PATH];**

The temporary directory, retrieved through **ExpandEnvironmentStrings()**.

**char name[MAX\_PATH];**

The name of the computer, retrieved through **GetComputerName()**.

**char workDisk;**

The disk, indexed by drive letter, used for server files such as incoming jobs. The index is alphabetical, disk A = 0, disk B = 1, etc.

**DWORD disks;**

The bit map representing the available disks. Disk A = 0x1, B = 0x2, C = 0x4, etc.

**DWORD diskSpace[26];**

The space available on disks in MegaBytes, indexed by drive letter. Disk A = diskSpace[0], B = diskSpace[1], etc.

**BYTE mac[8];**

The computer NIC hardware address (00:00:00:00:00:00), which is 6 bytes and 2 padding bytes.

**char reserved[32];**

Reserved for future use.

**} ConfigurationBlock;**

## Structure AlertData

See Also: [Class MaxNetManager](#), [Structure Job](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about the alert notifications.

### typedef struct {

**bool alertEnabled;**

This flag specifies if the alerts are enabled.

**int nthFrames;**

Use every n-th frame for a progress report.

**DWORD notifications;**

This bit map represents the enabled alerts, which are following values:

**NOTIFY\_FAILURE**

Notify on failure.

**NOTIFY\_PROGRESS**

Notify on progress.

**NOTIFY\_COMPLETION**

Notify on completion.

**} AlertData;**

## Structure MaxJob

See Also: [Class MaxNetManager](#), [Structure Job](#), [Structure SceneInfo](#), [Structure MaxJobRenderElements](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about a 3ds max specific job.

### **typedef struct {**

#### **bool init;**

This flag determines if the structure is valid.

#### **bool gammacorrection;**

This flag determines if gamma correction is used.

#### **float gammavaluein;**

The input gamma value for maps.

#### **float gammavalueout;**

The output gamma value for output images.

#### **float pixelaspect;**

The pixel aspect ratio.

#### **SceneInfo sceneInfo;**

The scene information data structure.

#### **MaxJobRenderElements re;**

The render elements data structure.

#### **char reserved[64];**

Reserved for future use.

### **} MaxJob;**

## Structure CombustionJob

See Also: [Class MaxNetManager](#), [Structure Job](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about a Combustion specific job.

**typedef struct {**

**bool init;**

This flag determines if the structure is valid.

**char reserved[128];**

Reserved for future use.

**} CombustionJob;**

## Structure JobText

See Also: [Class MaxNetManager](#), [Class CJobText](#), [Structure TextBufferOutput](#), [Structure JobRenderElements](#), [List of Job Text Types](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store textural information regarding a job.

### typedef struct {

**JOB\_TEXT\_TYPE type;**

The Job Text type. See the [List of Job Text Types](#) for details.

### union {

**TCHAR text[256];**

The text buffer contents.

**TextBufferOutput output**

The output text buffer.

**JobRenderElement re**

The Render Elements details of the job.

}

**} JobText;**

## List of Job Text Types

See Also: [Class MaxNetManager](#), [Class CJobText](#), [Structure JobText](#)

The list of the various JobText types (**JOB\_TEXT\_TYPE**) available.

### **JOB\_TEXT\_USER**

The user name.

### **JOB\_TEXT\_COMPUTER**

The computer name for job submissions.

### **JOB\_TEXT\_MANAGER\_SHARE**

The manager's share where it can find jobs. Filled by the Manager.

### **JOB\_TEXT\_FRAMES**

The frame sequence (such as the "1,2,4,5-40" types). Otherwise frames are defined in Job.

### **JOB\_TEXT\_MAX\_OUTPUT**

The output image file name, for 3ds max jobs.

### **JOB\_TEXT\_CMB\_OUTPUT**

The output image file name, for Combustion jobs.

### **JOB\_TEXT\_RENDER\_ELEMENT**

The list of Render Elements.

### **JOB\_TEXT\_CAMERA**

The list of cameras.

### **JOB\_TEXT\_RESERVED**

This type is reserved for future use.

## Structure ServerInfo

See Also: [Class MaxNetManager](#), [Structure NetworkStatus](#), [Structure ConfigurationBlock](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about a Server.

### typedef struct {

**DWORD size;**

The size of the structure, being **sizeof(ServerInfo)**.

**DWORD version;**

The version information, defined by **\_SERVER\_INFO\_VERSION**.

**float total\_frames;**

The total number of frames rendered.

**float total\_time;**

The total time spent rendering, in hours.

**ConfigurationBlock cfg;**

The network system configuration data. Refer to this structure for more information.

**NetworkStatus net\_status;**

The network status information.

**char reserved[32];**

Reserved for future use.

**} ServerInfo;**

## Structure Schedule

See Also: [Class MaxNetManager](#), [Structure WeekSchedule](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store hourly scheduling information.

**typedef struct {**

**DWORD hour;**

This bit map represents the hourly schedule where 24 bits represent the hours. A bit set to 0 indicates it's allowed to work, a bit set to 1 indicates it's not allowed to work.

**} JOBFAMES;**

## Class HierarchyEntry

See Also: [Class GenericHierarchy](#).

class HierarchyEntry

### Description:

This class represents an entry in a **GenericHierarchy**. All methods of the class are implemented by the system.

### Data Members:

public:

#### **int data;**

This is the polygon number. For example a donut shape will have polygon 0 and polygon 1. This is an index into the list of polygons for the shape.

#### **int children;**

Number of children of this entry.

#### **HierarchyEntry \*parent;**

Points to the parent entry.

#### **HierarchyEntry \*sibling;**

Points to the first sibling.

#### **HierarchyEntry \*child;**

Points to the first child.

#### **TSTR sortKey;**

The sort key. This is used internally by the hierarchy so that it can determine whether two hierarchies are compatible. Developers shouldn't alter this value.

### Methods:

### Prototype:

**HierarchyEntry();**

### Remarks:

Constructor. The entry is initialized as empty.

### Prototype:

**HierarchyEntry(int d, HierarchyEntry \*p, HierarchyEntry \*s);**

**Remarks:**

Constructor. The entry is initialized to the data passed.

**Parameters:****int d**

The value for **data**. This is the polygon number.

**HierarchyEntry \*p**

The **parent** pointer is initialized to this value.

**HierarchyEntry \*s**

The **sibling** pointer is initialized to this value.

**Prototype:****int HierarchyLevel();****Remarks:**

Returns the level in the hierarchy this entry represents. This is the number of parents this item has. The root level is not counted.

**Return Value:**

The level in the hierarchy of this entry.

**Prototype:****void AddChild(int d);****Remarks:**

Adds a new child entry to this entry using the data specified.

**Parameters:****int d**

The **data** value of the child. This is the polygon number.

**Prototype:****int GetChild(int index);****Remarks:**

Returns the specified child of this entry.

**Parameters:**

**int index**

The child to retrieve.

**Return Value:**

The specified child of this entry. If the specified index is greater than or equal to the number of children **INVALID\_HIERARCHY** is returned.

**Prototype:**

**int Children()**

**Remarks:**

Returns the number of children of this entry.

**Prototype:**

**void Sort();**

**Remarks:**

Sorts the hierarchy using the **sortKeys**.

## Class NoteKey

See Also: [Class NoteKeyTab](#), [Class DefNoteTrack](#), [Class Animatable](#).

class NoteKey

### Description:

This class contains the data for a single note of a Note track in Track View. This includes the time, text and flags for the notes.

### Data Members:

public:

**TimeValue time;**

The time of the note.

**TSTR note;**

The text of the note.

**DWORD flags;**

The note flags. One or more of the following values:

**NOTEKEY\_SELECTED**

The key is selected.

**NOTEKEY\_LOCKED**

The key is locked.

**NOTEKEY\_FLAGGED**

The key is flagged.

### Methods:

public:

### Prototype:

**NoteKey(TimeValue t,const TSTR &n,DWORD f=0);**

### Remarks:

Constructor. The data members are initialized to the values passed.

### Prototype:

**NoteKey(NoteKey& n);**

### Remarks:

Constructor. The data members are initialized from the NoteKey passed.

**Prototype:**

**void SetFlag(DWORD mask);**

**Remarks:**

Sets the specified flags.

**Parameters:**

**DWORD mask**

The flags to set.

**Prototype:**

**void ClearFlag(DWORD mask);**

**Remarks:**

Clears (sets to zero) the specified flags.

**Parameters:**

**DWORD mask**

The flags to clear.

**Prototype:**

**BOOL TestFlag(DWORD mask);**

**Remarks:**

Tests the specified flags. Returns TRUE if set; otherwise FALSE.

**Parameters:**

**DWORD mask**

The flags to test.

## Class IQuadMenuSettings

See Also: [Class IMenuItem](#) , [Class IMenuSettings](#), [Class IMenuColors](#)

class IQuadMenuSettings : public IMenuSettings, public FPStaticInterface

### Description:

This class is available in release 4.0 and later only.

This abstract class represents an interface for quad menu settings. The methods contained in this class allow you to access and control all quad menu related settings and configuration parameters.

### Methods:

public:

### Prototype:

```
virtual void SetFirstQuadDisplayed(QuadIndex  
firstQuadDisplayed) = 0;
```

### Remarks:

This method allows you to set the first quad which will be displayed when a quad menu pops up.

### Parameters:

**QuadIndex firstQuadDisplayed**

The quad index, one of the following; **QUAD\_ONE**, **QUAD\_TWO**, **QUAD\_THREE**, or **QUAD\_FOUR**.

### Prototype:

```
virtual QuadIndex GetFirstQuadDisplayed() const = 0;
```

### Remarks:

This method returns the index of the first quad which will be displayed.

### Return Value:

The quad index, one of the following; **QUAD\_ONE**, **QUAD\_TWO**, **QUAD\_THREE**, or **QUAD\_FOUR**.

### Prototype:

**virtual void SetUseUniformQuadWidth(bool useUniformQuadWidth) = 0;**

**Remarks:**

This method allows you to set whether the quad menu has a uniform width.

**Parameters:**

**bool useUniformQuadWidth**

TRUE to set the uniform width, FALSE to set it to non-uniform.

**Prototype:**

**virtual bool GetUseUniformQuadWidth() const = 0;**

**Remarks:**

This method returns the status of the uniform width flag for the quad menu.

TRUE if the quad menu has been set to use uniform width, otherwise FALSE.

**Prototype:**

**virtual void SetUseUniformQuadWidthBOOL(BOOL useUniformQuadWidth) = 0;**

**Remarks:**

This method allows you to set whether the quad menu has a uniform width.

This version of **SetUseUniformQuadWidth()** is provided for the function publishing system.

**Parameters:**

**BOOL useUniformQuadWidth**

TRUE to set the uniform width, FALSE to set it to non-uniform.

**Prototype:**

**virtual BOOL GetUseUniformQuadWidthBOOL() const = 0;**

**Remarks:**

This method returns the status of the uniform width flag for the quad menu.

TRUE if the quad menu has been set to use uniform width, otherwise FALSE.

This version of **GetUseUniformQuadWidth()** is provided for the function publishing system.

**Prototype:**

**virtual void SetMirrorQuad(bool mirrorQuad) = 0;**

**Remarks:**

This method allows you to set whether the quad menus are mirrored left to right.

**Parameters:**

**bool mirrorQuad**

TRUE to mirror the menus, otherwise FALSE.

**Prototype:**

**virtual bool GetMirrorQuad() const = 0;**

**Remarks:**

This method returns TRUE if the quad menu is mirrored left to right, otherwise FALSE.

**Prototype:**

**virtual void SetMirrorQuadBOOL(BOOL mirrorQuad) = 0;**

**Remarks:**

This method allows you to set whether the quad menus are mirrored left to right. This version of **SetMirrorQuad()** is provided for the function publishing system.

**Parameters:**

**BOOL mirrorQuad**

TRUE to mirror the menus, otherwise FALSE.

**Prototype:**

**virtual BOOL GetMirrorQuadBOOL() const = 0;**

**Remarks:**

This method returns TRUE if the quad menu is mirrored left to right, otherwise FALSE. This version of **GetMirrorQuad()** is provided for the function publishing system.

**Prototype:**

**virtual void SetMoveCursorOnReposition(bool  
moveCursorOnReposition) = 0;**

**Remarks:**

This method allows you to set whether the cursor moves when the quad menu is repositioned because of clipping the edge of the screen.

**Parameters:**

**bool moveCursorOnReposition**  
TRUE to move the cursor, otherwise FALSE.

**Prototype:**

**virtual bool GetMoveCursorOnReposition() const = 0;**

**Remarks:**

This method returns TRUE if the cursor moves when the quad menu is repositioned because of clipping the edge of the screen, otherwise FALSE.

**Prototype:**

**virtual void SetMoveCursorOnRepositionBOOL(BOOL  
moveCursorOnReposition) = 0;**

**Remarks:**

This method allows you to set whether the cursor moves when the quad menu is repositioned because of clipping the edge of the screen. This version of **SetMoveCursorOnReposition()** is provided for the function publishing system.

**Parameters:**

**BOOL moveCursorOnReposition**  
TRUE to move the cursor, otherwise FALSE.

**Prototype:**

**virtual BOOL GetMoveCursorOnRepositionBOOL() const = 0;**

**Remarks:**

This method returns TRUE if the cursor moves when the quad menu is

repositioned because of clipping the edge of the screen, otherwise FALSE.  
This version of **GetMoveCursorOnReposition()** is provided for the function publishing system.

**Prototype:**

**virtual void SetReturnCursorAfterReposition(bool returnCursorAfterReposition) = 0;**

**Remarks:**

This method allows you to set whether the cursor is moved the opposite distance that it was automatically moved when the quad menu is repositioned because of clipping the edge of the screen.

**Parameters:**

**bool returnCursorAfterReposition**  
TRUE to set the flag, otherwise FALSE.

**Prototype:**

**virtual bool GetReturnCursorAfterReposition() const = 0;**

**Remarks:**

This method returns TRUE if the cursor is moved the opposite distance that it was automatically moved when the quad menu is repositioned because of clipping the edge of the screen, otherwise FALSE.

**Prototype:**

**virtual void SetReturnCursorAfterRepositionBOOL(BOOL returnCursorAfterReposition) = 0;**

**Remarks:**

This method allows you to set whether the cursor is moved the opposite distance that it was automatically moved when the quad menu is repositioned because of clipping the edge of the screen. This version of **GetReturnCursorAfterReposition()** is provided for the function publishing system.

**Parameters:**

**BOOL returnCursorAfterReposition**

TRUE to set the flag, otherwise FALSE.

**Prototype:**

**virtual BOOL GetReturnCursorAfterRepositionBOOL() const = 0;**

**Remarks:**

This method returns TRUE if the cursor is moved the opposite distance that it was automatically moved when the quad menu is repositioned because of clipping the edge of the screen, otherwise FALSE. This version of **GetReturnCursorAfterReposition()** is provided for the function publishing system.

**Prototype:**

**virtual void SetCursorLocInBox\_0to1(float x, float y) = 0;**

**Remarks:**

This method allows you to set the initial location of the cursor in the center quad box.

**Parameters:**

**float x, float y**

The location of the cursor, as a ratio of the box size, between 0.0 and 1.0.

**Prototype:**

**virtual float GetCursorLocXInBox\_0to1() const = 0;**

**Remarks:**

This method returns the initial x location of the cursor in the center quad box, as a ratio of the box size, between 0.0 and 1.0.

**Prototype:**

**virtual float GetCursorLocYInBox\_0to1() const = 0;**

**Remarks:**

This method returns the initial y location of the cursor in the center quad box, as a ratio of the box size, between 0.0 and 1.0.

**Prototype:**

```
virtual const MenuColors *GetMenuColors(int quadNum) const = 0;
```

**Remarks:**

This method returns the color array for a specific quad.

**Parameters:**

**int quadNum**

The quad to obtain the color array for, (numbered 1 through 4).

**Prototype:**

```
virtual void SetTitleBarBackgroundColor(int quadNum, const Color& color) = 0;
```

**Remarks:**

This method allows you to set the title bar background color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

```
virtual const Color& GetTitleBarBackgroundColor(int quadNum) const = 0;
```

**Remarks:**

This method returns the title bar background color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual COLORREF GetTitleBarBackgroundColorRef(int quadNum) const = 0;**

**Remarks:**

This method returns the title bar background color of a specific quad. This method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual void SetTitleBarTextColor(int quadNum, const Color& color) = 0;**

**Remarks:**

This method allows you to set the title bar text color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

**virtual const Color& GetTitleBarTextColor(int quadNum) const = 0;**

**Remarks:**

This method returns the title bar text color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual COLORREF GetTitleBarTextColorRef(int quadNum)**

**const = 0;**

**Remarks:**

This method returns the title bar text color of a specific quad. This method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual void SetItemBackgroundColor(int quadNum, const Color& color) = 0;**

**Remarks:**

This method allows you to set the item background color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

**virtual const Color& GetItemBackgroundColor(int quadNum) const = 0;**

**Remarks:**

This method returns the item background color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual COLORREF GetItemBackgroundColorRef(int quadNum) const = 0;**

**Remarks:**

This method returns the item background color of a specific quad. This method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual void SetItemTextColor(int quadNum, const Color& color) = 0;
```

**Remarks:**

This method allows you to set the item text color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

```
virtual const Color& GetItemTextColor(int quadNum) const = 0;
```

**Remarks:**

This method returns the item text color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual COLORREF GetItemTextColorRef(int quadNum) const = 0;
```

**Remarks:**

This method returns the item text color of a specific quad. This method returns

the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual void SetLastExecutedItemTextColor(int quadNum, const  
Color& color) = 0;
```

**Remarks:**

This method allows you to set the last executed item text color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

```
virtual const Color& GetLastExecutedItemTextColor(int  
quadNum) const = 0;
```

**Remarks:**

This method returns the last executed item text color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual COLORREF GetLastExecutedItemTextColorRef(int  
quadNum) const = 0;
```

**Remarks:**

This method returns the last executed item text color of a specific quad. This

method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual void SetHighlightedItemBackgroundColor(int quadNum, const Color& color) = 0;**

**Remarks:**

This method allows you to set the highlighted item background color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

**virtual const Color& GetHighlightedItemBackgroundColor(int quadNum) const = 0;**

**Remarks:**

This method returns the highlighted item background color of a specific quad.  
This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual COLORREF GetHighlightedItemBackgroundColorRef(int quadNum) const = 0;**

**Remarks:**

This method returns the highlighted item background color of a specific quad.

This method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual void SetHighlightedItemTextColor(int quadNum, const  
Color& color) = 0;
```

**Remarks:**

This method allows you to set the highlighted item text color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

```
virtual const Color& GetHighlightedItemTextColor(int quadNum)  
const = 0;
```

**Remarks:**

This method returns the highlighted item text color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual COLORREF GetHighlightedItemTextColorRef(int  
quadNum) const = 0;
```

**Remarks:**

This method returns the highlighted item text color of a specific quad. This

method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual void SetBorderColor(int quadNum, const Color& color) = 0;
```

**Remarks:**

This method allows you to set the border color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

```
virtual const Color& GetBorderColor(int quadNum) const = 0;
```

**Remarks:**

This method returns the border color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual COLORREF GetBorderColorRef(int quadNum) const = 0;
```

**Remarks:**

This method returns the border color of a specific quad. This method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual void SetDisabledShadowColor(int quadNum, const Color& color) = 0;**

**Remarks:**

This method allows you to set the disabled shadow color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

**virtual const Color& GetDisabledShadowColor(int quadNum)  
const = 0;**

**Remarks:**

This method returns the disabled shadow color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

**virtual COLORREF GetDisabledShadowColorRef(int quadNum)  
const = 0;**

**Remarks:**

This method returns the disabled shadow color of a specific quad. This method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual void SetDisabledHighlightColor(int quadNum, const  
Color& color) = 0;
```

**Remarks:**

This method allows you to set the disabled highlight color for a specific quad.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Color& color**

The color to set.

**Prototype:**

```
virtual const Color& GetDisabledHighlightColor(int quadNum)  
const = 0;
```

**Remarks:**

This method returns the disabled highlight color of a specific quad. This method returns the color as a **Color**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

**Prototype:**

```
virtual COLORREF GetDisabledHighlightColorRef(int  
quadNum) const = 0;
```

**Remarks:**

This method returns the disabled highlight color of a specific quad. This method returns the color as a **COLORREF**.

**Parameters:**

**int quadNum**

The quad (numbered 1 through 4).

# Class IMenuSettings

See Also: [Class IMenuGlobalContext](#)

class IMenuSettings

## Description:

This class is available in release 4.0 and later only.

This abstract class represents an interface for all general menu settings. Methods that are marked as internal should not be used.

## Methods:

public:

## Prototype:

```
virtual bool IsTokenValid(const ValidityToken& token) = 0;
```

## Remarks:

This method is used internally.

This method checks if a token is valid.

## Parameters:

**ValidityToken& token**

A reference to a token for which to check its validity.

## Return Value:

TRUE if the token is valid, otherwise FALSE.

## Prototype:

```
virtual void UpdateValidityToken(ValidityToken& token) const = 0;
```

## Remarks:

This method is used internally.

This method updates the validity token.

## Parameters:

**ValidityToken& token**

A reference to a token to update.

**Prototype:**

**virtual void ResetDefaults() = 0;**

**Remarks:**

This method will reset the menu settings to their defaults.

**Prototype:**

**virtual void SetBorderSz(int borderSz) = 0;**

**Remarks:**

This method allows you to set the menu border size.

**Parameters:**

**int borderSz**

The border size in pixels.

**Prototype:**

**virtual int GetBorderSz() const = 0;**

**Remarks:**

This method returns the menu border size.

**Prototype:**

**virtual void SetHorizontalMarginInPoints(int horizontalMarginInPoints) = 0;**

**Remarks:**

This method allows you to set the menu's horizontal margin size.

**Parameters:**

**int horizontalMarginInPoints**

The horizontal margin size in points.

**Prototype:**

**virtual int GetHorizontalMarginInPoints() const = 0;**

**Remarks:**

This method returns the menu's horizontal margin size (in points).

**Prototype:**

**virtual int GetHorizontalMargin(HDC hDC) const = 0;**

**Remarks:**

This method returns the menu's horizontal margin, in pixels.

**Parameters:**

**HDC hDC**

A handle to a device context.

**Prototype:**

**virtual void SetVerticalMarginInPoints(int verticalMarginInPoints) = 0;**

**Remarks:**

This method allows you to set the menu's vertical margin size.

**Parameters:**

**int verticalMarginInPoints**

The vertical margin size in points.

**Prototype:**

**virtual int GetVerticalMarginInPoints() const = 0;**

**Remarks:**

This method returns the menu's vertical margin size (in points).

**Prototype:**

**virtual int GetVerticalMargin(HDC hDC) const = 0;**

**Remarks:**

This method returns the menu's vertical margin, in pixels.

**Parameters:**

**HDC hDC**

A handle to a device context.

**Prototype:**

**virtual void SetItemFontFace(TCHAR\* szItemFontFace) = 0;**

**Remarks:**

This method allows you to set the menu item's font typeface.

**Parameters:**

**TCHAR\* szItemFontFace**

A string containing the typeface name.

**Prototype:**

**virtual const TCHAR\* GetItemFontFace() const = 0;**

**Remarks:**

This method returns the name of the menu item's font typeface.

**Prototype:**

**virtual void SetTitleFontFace(TCHAR\* szTitleFontFace) = 0;**

**Remarks:**

This method allows you to set the menu title's font typeface.

**Parameters:**

**TCHAR\* szTitleFontFace**

A string containing the typeface name.

**Prototype:**

**virtual const TCHAR\* GetTitleFontFace() const = 0;**

**Remarks:**

This method returns the name of the menu title's font typeface.

**Prototype:**

**virtual void SetItemFontSize(int itemFontSize) = 0;**

**Remarks:**

This method allows you to set the menu item's font size.

**Parameters:**

### **int itemFontSize**

The size of the font, in points.

#### **Prototype:**

```
virtual int GetItemFontSize() const = 0;
```

#### **Remarks:**

This method returns the menu item's font size, in points.

#### **Prototype:**

```
virtual void SetTitleFontSize(int titleFontSize) = 0;
```

#### **Remarks:**

This method allows you to set the menu title's font size.

#### **Parameters:**

### **int titleFontSize**

The size of the font, in points.

#### **Prototype:**

```
virtual int GetTitleFontSize() const = 0;
```

#### **Remarks:**

This method returns the menu title's font size, in points.

#### **Prototype:**

```
virtual void SetUseUniformItemHeight(bool  
useUniformItemHeight) = 0;
```

#### **Remarks:**

This method allows you to set the status of a menu item's uniform height flag.

#### **Parameters:**

### **bool useUniformItemHeight**

TRUE to set the uniform height flag ON, FALSE to set it to OFF.

#### **Prototype:**

**virtual bool GetUseUniformItemHeight() const = 0;**

**Remarks:**

This method returns TRUE or FALSE if the menu item's uniform height flag is set or not set, respectively.

**Prototype:**

**virtual void SetUseUniformItemHeight(BOOL useUniformItemHeight) = 0;**

**Remarks:**

This method allows you to set the status of a menu item's uniform height flag. This version of **SetUniformItemHeight()** is provided for the function publishing system.

**Parameters:**

**BOOL useUniformItemHeight**

TRUE to set the uniform height flag ON, FALSE to set it to OFF.

**Prototype:**

**virtual BOOL GetUseUniformItemHeight(BOOL) const = 0;**

**Remarks:**

This method returns TRUE or FALSE if the menu item's uniform height flag is set or not set, respectively. This version of **GetUniformItemHeight()** is provided for the function publishing system.

**Prototype:**

**virtual void SetOpacity(float opacity) = 0;**

**Remarks:**

This method allows you to set the menu's opacity value.

**Parameters:**

**float opacity**

The opacity value, ranging from 0.0 – 1.0.

**Prototype:**

**virtual float GetOpacity() const = 0;**

**Remarks:**

This method returns the menu's opacity value.

**Prototype:**

**virtual void SetDisplayMethod(DisplayMethod displayMethod) = 0;**

**Remarks:**

This method allows you to set a menu's display method.

**Parameters:**

**DisplayMethod displayMethod**

The display method (enum), which is either of the following;

**DM\_NORMAL, DM\_STRETCH, DM\_FADE,  
DM\_NUM\_METHODS**

**Prototype:**

**virtual DisplayMethod GetDisplayMethod() const = 0;**

**Remarks:**

This method returns the menu's display method, which is either of the following; **DM\_NORMAL, DM\_STRETCH, DM\_FADE, DM\_NUM\_METHODS**

**Prototype:**

**virtual void SetAnimatedSteps(unsigned int steps) = 0;**

**Remarks:**

This method allows you to set the menu's number of animated steps for the 'growing' effect.

**Parameters:**

**unsigned int steps**

The number of steps.

**Prototype:**

**virtual unsigned int GetAnimatedSteps() const = 0;**

**Remarks:**

This method returns the menu's number of animated steps used for the 'growing' effect.

**Prototype:**

**virtual void SetAnimatedStepTime(unsigned int ms) = 0;**

**Remarks:**

This method allows you to set the menu's animated step time.

**Parameters:**

**unsigned int ms**

The animated step time, in milliseconds.

**Prototype:**

**virtual unsigned int GetAnimatedStepTime() const = 0;**

**Remarks:**

This method returns the menu's animated step time, in milliseconds.

**Prototype:**

**virtual void SetSubMenuPauseTime(unsigned int ms) = 0;**

**Remarks:**

This method allows you to set the delay before a submenu is displayed.

**Parameters:**

**unsigned int ms**

The delay, in milliseconds.

**Prototype:**

**virtual unsigned int GetSubMenuPauseTime() const = 0;**

**Remarks:**

This method returns the delay before a submenu is displayed, in milliseconds.

**Prototype:**

```
virtual void SetUseLastExecutedItem(bool useLastExecutedItem)  
= 0;
```

**Remarks:**

This method allows you to set the "last executed item" flag which determines whether to use the menu's last executed item when the user clicks on the menu's titlebar.

**Parameters:**

**bool useLastExecutedItem**

TRUE to turn ON the flag, FALSE to turn the flag off.

**Prototype:**

```
virtual bool GetUseLastExecutedItem() const = 0;
```

**Remarks:**

This method returns whether the "last executed item" flag is set (TRUE) or not set (FALSE). The flag determines whether to use the menu's last executed item when the user clicks on the menu's titlebar.

**Prototype:**

```
virtual void SetUseLastExecutedItemBOOL(BOOL  
useLastExecutedItem) = 0;
```

**Remarks:**

This method allows you to set the "last executed item" flag which determines whether to use the menu's last executed item when the user clicks on the menu's titlebar. This version of **SetUseLastExecutedItem()** is provided for the function publishing system.

**Parameters:**

**BOOL useLastExecutedItem**

TRUE to turn ON the flag, FALSE to turn the flag off.

**Prototype:**

**virtual BOOL GetUseLastExecutedItemBOOL() const = 0;**

**Remarks:**

This method returns whether the "last executed item" flag is set (TRUE) or not set (FALSE). The flag determines whether to use the menu's last executed item when the user clicks on the menu's titlebar. This version of **GetUseLastExecutedItem()** is provided for the function publishing system.

**Prototype:**

**virtual void SetRepositionWhenClipped(bool repositionWhenClipped) = 0;**

**Remarks:**

This method allows you to set the flag which controls and determines whether the menu is repositioned when near the edge of the screen.

**Parameters:**

**bool repositionWhenClipped**

TRUE to turn repositioning ON, FALSE to turn it OFF.

**Prototype:**

**virtual bool GetRepositionWhenClipped() const = 0;**

**Remarks:**

This method returns the status of the flag which controls and determines whether the menu is repositioned when near the edge of the screen.

**Return Value:**

TRUE if the flag is ON, otherwise FALSE.

**Prototype:**

**virtual void SetRepositionWhenClippedBOOL(BOOL repositionWhenClipped) = 0;**

**Remarks:**

This method allows you to set the flag which controls and determines whether the menu is repositioned when near the edge of the screen. This version of **SetRepositionWhenClipped()** is provided for the function publishing

system.

**Parameters:**

**BOOL repositionWhenClipped**

TRUE to turn repositioning ON, FALSE to turn it OFF.

**Prototype:**

**virtual BOOL GetRepositionWhenClipped() const = 0;**

**Remarks:**

This method returns the status of the flag which controls and determines whether the menu is repositioned when near the edge of the screen. This version of **GetRepositionWhenClipped()** is provided for the function publishing system.

**Return Value:**

TRUE if the flag is ON, otherwise FALSE.

**Prototype:**

**virtual void SetRemoveRedundantSeparators(bool  
removeRedundantSeparators) = 0;**

**Remarks:**

This method allows you to set the flag which controls and determines whether the menu should remove redundant separators.

**Parameters:**

**bool removeRedundantSeparators**

TRUE to turn the flag ON, FALSE to turn it OFF.

**Prototype:**

**virtual bool GetRemoveRedundantSeparators() const = 0;**

**Remarks:**

This method returns the status of the flag which controls and determines whether the menu should remove redundant separators.

**Return Value:**

TRUE if the flag is ON, otherwise FALSE.

**Prototype:**

```
virtual void SetRemoveRedundantSeparators(BOOL removeRedundantSeparators) = 0;
```

**Remarks:**

This method allows you to set the flag which controls and determines whether the menu should remove redundant separators. This version of **SetRemoveRedundantSeparators()** is provided for the function publishing system.

**Parameters:**

**BOOL removeRedundantSeparators**  
TRUE to turn the flag ON, FALSE to turn it OFF.

**Prototype:**

```
virtual BOOL GetRemoveRedundantSeparators(BOOL) const = 0;
```

**Remarks:**

This method returns the status of the flag which controls and determines whether the menu should remove redundant separators. This version of **GetRemoveRedundantSeparators()** is provided for the function publishing system.

**Return Value:**

TRUE if the flag is ON, otherwise FALSE.

# Class IMenuTimer

See Also: [Class IMenu](#) , [Class IMenuGlobalContext](#)

class IMenuTimer

## Description:

This class is available in release 4.0 and later only.

This abstract class represents an interface for a timer and is used internally.

Methods that are marked as internal should not be used.

## Methods:

public:

## Prototype:

**virtual bool IsRunning() = 0;**

## Remarks:

This method is used internally.

This method indicates whether the timer is running or not by returning TRUE or FALSE.

## Prototype:

**virtual void Start(IMenu\* pIMenu, EventParam timingType) = 0;**

## Remarks:

This method is used internally.

This method starts or restarts a timer for a specified IMenu.

## Prototype:

**virtual void Stop() = 0;**

## Remarks:

This method is used internally.

This method will stop the timer.

## Prototype:

**virtual void CheckTime() = 0;**

**Remarks:**

This method is used internally.

This method instructs the timer to check the time. If the time has elapsed it will notify its IMenu client.

**Prototype:**

**virtual bool HasElapsed() = 0;**

**Remarks:**

This method is used internally.

This method indicates whether the timer has elapsed by returning TRUE or FALSE.

**Prototype:**

**virtual void SetElapsedTime(unsigned int elapsedTime) = 0;**

**Remarks:**

This method is used internally.

This method allows you to set the elapse time for the timer.

**Parameters:**

**unsigned int elapsedTime**

The time at which the timer should elapse

**Prototype:**

**virtual unsigned int GetElapsedTime() const = 0;**

**Remarks:**

This method is used internally.

This method returns the elapse time that's been set for the timer.

**Prototype:**

**virtual IMenu\* GetIMenu() const = 0;**

**Remarks:**

This method is used internally.

This method returns a pointer to the IMenu client associated with the timer.

**Prototype:**

**virtual EventParam GetTimingType() const = 0;**

**Remarks:**

This method is used internally.

This method returns the timing type for the timer.

## Class ItemID

See Also: [Class IMenu](#), [Class IMenuItem](#) , [Class IMenuGlobalContext](#)

class ItemID

### Description:

This class is available in release 4.0 and later only.

This class is used internally. Methods that are marked as internal should not be used.

### Data Members:

public:

**IMenu\* mpMenu;**

**IMenuItem\* mpItem;**

### Methods:

public:

### Prototype:

**ItemID();**

### Remarks:

This method is used internally.

Constructor. Initialized **mpMenu(NULL)** and **mpItem(NULL)**.

### Default Implementation:

**{ }**

### Prototype:

**void Null();**

### Remarks:

This method is used internally.

Set **mpMenu = NULL** and **mpItem = NULL**.

### Prototype:

**friend bool operator==(ItemID& a, ItemID& b);**

**Remarks:**

This method is used internally.

This operator tests for equality of two ItemID's.

**Parameters:**

**ItemID& a, ItemID& b**

The two ItemID's you wish to test for equality.

**Prototype:**

**friend bool operator!=(ItemID& a, ItemID& b);**

**Remarks:**

This method is used internally.

This operator tests for inequality of two ItemID's.

**Parameters:**

**ItemID& a, ItemID& b**

The two ItemID's you wish to test for inequality.

**Default Implementation:**

```
{ return !(a == b); }
```

# Class IMenuColors

See Also: [Class Color](#)

class IMenuColors

## Description:

This class is available in release 4.0 and later only.

This class represents the container for a menu's color settings and is used internally. Methods that are marked as internal should not be used.

## Data Members:

public:

**Color mTitleBarBackgroundColor;**

The title bar background color.

**Color mTitleBarTextColor;**

The title bar text color.

**Color mItemBackgroundColor;**

The item background color.

**Color mItemTextColor;**

The item text color

**Color mLastExecutedItemTextColor;**

The last executed item text color.

**Color mHighlightedItemBackgroundColor;**

The highlighted item background color.

**Color mHighlightedItemTextColor;**

The highlighted item text color.

**Color mBorderColor;**

The border color.

**Color mDisabledShadowColor;**

The disabled shadow color.

**Color mDisabledHighlightColor;**

The disabled highlight color.

## Methods:

public:

**Prototype:**

**MenuColors();**

**Remarks:**

This method is used internally.

Constructor.

**Default Implementation:**

**{ ResetDefaults(); }**

**Prototype:**

**void ResetDefaults();**

**Remarks:**

This method is used internally.

This method resets the menu colors to their defaults.

**Default Implementation:**

```
{  
  mTitleBarBackgroundColor = Color(.0f, .0f, .0f);  
  mTitleBarTextColor = Color(.75f, .75f, .75f);  
  mItemBackgroundColor = Color(.75f, .75f, .75f);  
  mItemTextColor = Color(.0f, .0f, .0f);  
  mLastExecutedItemTextColor = Color(.95f, .85f, .0f);  
  mHighlightedItemBackgroundColor = Color(.95f, .85f, .0f);  
  mHighlightedItemTextColor = Color(.0f, .0f, .0f);  
  mBorderColor = Color(.0f, .0f, .0f);  
  mDisabledShadowColor = Color(.5f, .5f, .5f);  
  mDisabledHighlightColor = Color(1.0f, 1.0f, 1.0f);  
}
```

## Class DropClipFormat

See Also: [Class DropType](#), [Class VIZableClipFmt](#), [Class DropScriptClipFmt](#), [List of DropTypes](#)

```
class DropClipFormat : public InterfaceServer
```

### Description:

This class is available in release 4.0 and later only.

**DropClipFormat** is the base class for the various supported clipboard formats contained in a dropped **IDataObject**. Subclasses represent particular **IDataObject** clip format or package of related formats that can be accepted by various windows in 3ds max. The prime responsibility of each is to recognize its presence in a dropped **IDataObject** and to parse the data object into one of the supported **DropTypes**. Each subclass should have a singleton instance created. This is automatically registered with the DnD system for use in the clipform recognition routines.

### Data Members:

protected:

```
static Tab<DropClipFormat*> clipFmts;
```

The table of supported clip formats.

### Methods:

public:

### Prototype:

```
DropClipFormat();
```

### Remarks:

Constructor.

Each **DropClimFormat** instance created is kept in the **clipFmts** table

### Prototype:

```
static DropClipFormat* FindClipFormat(IDataObject*  
pDataObject);
```

### Remarks:

This method returns a pointer to the **DropClipFormat** (singleton) corresponding to the clip format in the given **IDataObject**, or NULL if the **IDataObject** contains no recognizable formats. This is primarily used by the low-level default **DragEnter()** function in DnD manager.

**Parameters:**

**IDataObject\* pDataObject**

The data object you wish to return the clip format for.

**Prototype:**

**virtual bool CheckClipFormat(IDataObject\* pDataObject)**

**Remarks:**

This method should be implemented by each subclass to detect the presence of its clipformat(s) in the given **IDataObject**. See **ParseDataObject()** below for a detailed example.

**Parameters:**

**IDataObject\* pDataObject**

The data object.

**Return Value:**

TRUE if the data was queried successfully, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

**virtual DropType\* ParseDataObject(IDataObject\* pDataObject);**

**Remarks:**

This method should be implemented by each subclass to parse its clipformat(s) in the given **IDataObject** into the corresponding **DropType** subclass instance. For example, the **DropClipFormats** that accept dropped files will typically return one of the **FileDropType** subclasses depending on the filename suffix. A list of built-in clipformats:

**IDropPackageClipFmt** iDrop XML package

**VIZableClipFmt** VIZable file URL

**DropScriptClipFmt** internal dropScript

Here's an example (simplified) **VIZableClipFmt** implementation, which accepts a custom **CF\_MAXURL** clip format containing the URL of a file. **CheckClipFormat()** returns true if it finds the **CF\_MAXURL** clipboard format present in the given **IDataObject**. Because this is a dropping file, **ParseDataObject()** clears the current droptype data (the **FileDropType::Init()**), extracts the file name from the **IDataObject** and installs it into the **FileDropType current\_package** variable. It then asks the **FileDropType** class to recognize the actual file type being dropped and return the corresponding **FileDropType** subclass instance (using **FileDropType::FindDropType()**).

```
bool VIZableClipFmt::CheckClipFormat(IDataObject*
pDataObject)
{
    // accept CF_MAXURL clip formats
    FORMATETC fmt = { NULL, NULL, DVASPECT_CONTENT,
-1, NULL };
    fmt.cfFormat = RegisterClipboardFormat(_T("CF_MAXURL"));
    fmt.tymed = TYMED_HGLOBAL;
    return SUCCEEDED(pDataObject->QueryGetData(&fmt)) ==
TRUE;
}
DropType* VIZableClipFmt::ParseDataObject(IDataObject*
pDataObject)
{
    // parse a CF_MAXURL clipformat into one of the FileDropTypes
&
    // fill in the FileDropType::current_package URLTab
HRESULT hr;
```

```

FORMATETC fmt = { NULL, NULL, DVASPECT_CONTENT,
-1, NULL };
STGMEDIUM stg = { TYMED_NULL, NULL, NULL };

fmt.tymed = TYMED_HGLOBAL;
fmt.cfFormat = RegisterClipboardFormat(_T("CF_MAXURL"));

// clear out the file drop current data
FileDropType::Init();

// look for CF_MAXURL formats
hr = pDataObject->GetData(&fmt, &stg);
if(SUCCEEDED(hr))
{
    // found, get the max file name
    TCHAR szUrl[MAX_PATH];
ZeroMemory(szUrl, sizeof(szUrl));
wcstombs(szUrl,
    reinterpret_cast<wchar_t*>(GlobalLock(stg.hGlobal)),
    MAX_PATH-1);
GlobalUnlock(stg.hGlobal);
ReleaseStgMedium(&stg);
    // add it to the current_package URLTab
FileDropType::current_package.Add(szUrl);
}

// if we have a non-NULL package, get the appropriate
// FileDropType (usually based on file extension), by asking the
// utility DropType finder in FileDropType
if (FileDropType::current_package.Count() > 0)
    return FileDropType::FindDropType(

```

```
        FileDropType::current_package[0], pDataObject);  
else  
    return NULL;  
}
```

**Parameters:**

**IDataObject\* pDataObject**

The data object.

**Return Value:**

A pointer to the drop-type.

**Default Implementation:**

```
{ return NULL; }
```

## Class FileDropType

See Also: [Class DropType](#), , [Class URLTab](#), [Class SceneFileDropType](#), [Class ImageFileDropType](#), [Class ImportFileDropType](#), [Class ScriptFileDropType](#), [List of DropTypes](#)

```
class FileDropType : public DropType
```

### Description:

This class is available in release 4.0 and later only.

This is an intermediate base class for drop content that comes in the form of a package of filenames or URLs. This class maintains a list of all its subclass singleton instances and provides utility methods for finding an appropriate subclass instance, based on the dropped filename, and for download URL packages.

### Data Members:

protected:

```
static Tab<FileDropType*> fileDropTypes;
```

The table of FileDropTypes.

```
static TSTR download_directory;
```

Cache for current default URL package download directory .

public

```
static URLTab current_package;
```

Currently dropping URL package. Filled in by the active **DropClipFormat** in its **ParseDataObject()** method .

### Methods:

public:

### Prototype:

```
FileDropType();
```

### Remarks:

Constructor.

### Prototype:

**static void Init();**

**Remarks:**

This method clears the currently-parsed drop data.

**Default Implementation:**

```
{ current_package.Clear(); DropType::Init(); }
```

**Prototype:**

```
virtual int TypeCode()=0;
```

**Remarks:**

This method returns the typecode of the DropType.

**Default Implementation:**

```
{ return FILE_DROPTYPE; }
```

**Prototype:**

```
virtual bool IsDropType(int code);
```

**Remarks:**

This method returns TRUE if the DropType is of the specified DropType code, otherwise FALSE.

**Parameters:**

**int code**

The DropType code.

**Default Implementation:**

```
{ return code == TypeCode() || code == FILE_DROPTYPE; }
```

**Prototype:**

```
virtual bool Load(bool showProgress = true);
```

**Remarks:**

This method will load the URLTab in current\_package (filled in by the current DropClipFormat).

**Parameters:**

**bool showProgress = true**

The download progress dialog can be displayed by passing true.

**Prototype:**

```
static FileDropType* FindDropType(TCHAR* filename,  
IDataObject* pDataObject = NULL);
```

**Remarks:**

This method finds and returns the **FileDropType** subclass corresponding to the given filename suffix. See the [List of DropTypes](#) for more details.

**Parameters:**

**TCHAR\* filename**

The filename suffix.

**IDataObject\* pDataObject = NULL**

A pointer to the IDataObject.

**Return Value:**

The FileDropType that corresponds to the filename suffix. This could be one of the following; **sceneFileDropType**, **imageFileDropType**, **importFileDropType**, **dropScriptFileDropType**.

**Prototype:**

```
virtual bool CheckDropType(TCHAR* filename);
```

**Remarks:**

Subclasses should implement this method to recognize the file types associated with this drop type. This is used by **FindDropType()**.

**Parameters:**

**TCHAR\* filename**

The filename suffix.

**Return Value:**

TRUE if the filename suffix checks out, otherwise FALSE.

**Default Implementation:**

```
{ return false; }
```

**Prototype:**

**static bool DownloadPackage(URLTab& package, TCHAR\* szDirectory, HWND hwnd = NULL, bool showProgress = true);**

**Remarks:**

This method serves as a utility function that can be used to download a package of URLs to the specified directory. If the **hwnd** argument is supplied, any progress or other messages are centered over that window.

**Parameters:**

**URLTab& package**

A reference to the local copies of the URL strings.

**TCHAR\* directory**

The directory path string to download to.

**HWND hwnd = NULL**

A handle to the window. If this is set to NULL, the default 3ds max window is used.

**bool showProgress = false**

The download progress dialog can be displayed by passing true.

**Return Value:**

TRUE if the download was successful, otherwise FALSE.

**Prototype:**

**static TCHAR\* GetDownloadDirectory();**

**Remarks:**

This method returns the fully-specified path to the directory in which package drops are downloaded.

**Prototype:**

**static bool DownloadUrlToDisk(HWND hwnd, TCHAR\* szUrl, TCHAR\* szPathname, DWORD flags=0);**

**Remarks:**

This method allows you to download the file referenced by the URL to disk.

**Parameters:**

**HWND hwnd = NULL**

A handle to the window.

**TCHAR\* url**

The URL string of the file to download.

**TCHAR\* fileName**

The filename string of the URL to store on disk.

**DWORD flags=0**

Additional controls to the download behavior. Currently only one flag is supported, **DOWNLOADDLG\_NOPLACE**, which hides an option in the progress dialog that allows the user to place (move) a dropped object immediately after being dropped.

**Return Value:**

TRUE if the download was successful, otherwise FALSE.

## Class DropScriptDropType

See Also: [Class DropType](#), [Class DragAndDropHandler](#), [Class FPParams](#), [Class MacroEntry](#), [List of DropTypes](#)

```
class DropScriptDropType : public DropType
```

### Description:

This class is available in release 4.0 and later only.

This class is an intermediate base class for drop content that comes in the form of a dropScript. This is a special kind of macroScript that implements dropScript event handlers (see the DropScript documentation for details.) The prime subclass is **DropScriptFileDropType** which recognizes files of type .ds. The parsed data for this type is a single parsed macroScript, represented as a **MacroEntry** pointer. The **DropScriptDropType** class provides utility methods for compiling a .ds file into the **current\_dropscript** slot and for running the DnD-associated handlers in the current dropScript.

The methods **RunDropScriptDragEnter(FPParams\* params)**, **RunDropScriptDragOver(FPParams\* params)** and **RunDropScriptDrop(FPParams\* params)** take care of the 'on droppable' handler in the **current\_dropscript**, if supplied. The **DragAndDropHandler::DragEnter** call is usually made once on initial entry to a registered DnD target window and **DragAndDropHandler::DragOver** is usually called as the mouse moves over this window. In both cases, the handler returns true or false to indicate whether the dropping dropScript will be accepted. If a handler is not supplied, the dropScript is always deemed droppable. If the handler returns false, the not-droppable cursor is shown.

The handler is called with a set of arguments, supplied by the **DragAndDropHandler**, that usually depends on the window currently under the mouse pointer. For example, over a viewport, the current mouse coordinates, scene node under the mouse, slot number in a list window, etc. By convention, the first argument is positional and always a window type name, such as "Viewport" or "MaterialEditor", and all the others are keyword arguments, since they will vary from window to window. They are delivered to the RunDropScriptXXX methods in a Function Publishing **FPParam** object, so

that handler code needs to deal as little as possible with the MAXScript SDK. Here's an example code fragment from the default drop handler:

```
FPParams params (6,  
    TYPE_NAME, (vpwin ? _T("viewport") : _T("max")),  
    TYPE_KEYARG_MARKER,  
    TYPE_NAME, _T("node"),  
    TYPE_INODE, m_nodectx,  
    TYPE_NAME, _T("point"),  
    TYPE_POINT, &pt);  
// run the dragEnter handler & set dropeffect based on result  
if (dropScriptFileDropType.RunDropScriptDragEnter(&params))  
    *pdwEffect = DROPEFFECT_COPY;  
else  
    *pdwEffect = DROPEFFECT_NONE;
```

In the above code, the handler is called with 3 actual arguments, one position and two keyword. They are loaded into the 'params' instance with the **FPParams varargs** constructor. The first is the positional window name, in this case either #viewport or #max, then comes a special **TYPE\_KEYARG\_MARKER** signalling that the following arguments are keyword. The keyword args are given in pairs, name then value, in this case node: and point:. See the Function Publishing system documentation for full details on using the **FPParams** class for passing parameter blocks. An example droppable handler might be as follows:

**on droppable window node: do**

**return window == #viewport and superclassOf node == Shape**

This handler effectively makes the dropScript droppable if the mouse is over a Shape object in a viewport window. Notice that the function only looks at the node: keyword argument in this definition; arguments delivered as keyword arguments can vary from call to call and the called function can choose to look at only subset of them.

## Data Members:

public

**static MacroEntry\* current\_dropscript;**

Cache for current macroScript. See

`\MAXSDK\INCLUDE\iMacroScript.h` for the MacroScript manager

public API. There are also utility methods in DropScriptDropType that do all the necessary DnD compiling & running of macroScripts, so you only have to deal with the MacroScript manager for special processing.

## Methods:

public:

### Prototype:

**virtual int TypeCode()=0;**

### Remarks:

This method returns the typecode of the DropType.

### Default Implementation:

```
{ return DROPSCRIPT_DROPTYPE; }
```

### Prototype:

**virtual bool IsDropType(int code);**

### Remarks:

This method returns TRUE if the DropType is of the specified DropType code, otherwise FALSE.

### Parameters:

**int code**

The DropType code.

### Default Implementation:

```
{ return code == TypeCode() || code ==  
DROPSCRIPT_DROPTYPE; }
```

### Prototype:

**virtual DWORD DropEffect();**

**Remarks:**

This method returns the dropeffect currently supported by the accepted dropping type.

**Default Implementation:**

```
{ return DROPEFFECT_MOVE; }
```

The following methods provide assistance for developing custom drag-and-drop handlers that want to accept dropScripts. They work on the shared `current_dropscript` static data member in **DropScriptDropType**.

**Prototype:**

**BOOL CompileDropScript(TCHAR\* filename);**

**Remarks:**

This method parses the given file, looking for a single macroScript definition. If successful, interns the macroScript and places its corresponding **MacroEntry\*** in the **current\_dropscript** static data member. Note that if there is more code than just a single macroScript in the file, only the last macroScript definition is taken; the other code is NOT executed, so you cannot include auxiliary global functions and other prep code in the file. These should be inside the body of the macroScript, as local data and functions.

**Parameters:**

**TCHAR\* filename**

The filename of the script.

**Return Value:**

TRUE if successfully compiled, otherwise FALSE.

**Prototype:**

**BOOL RunDropScriptDragEnter(FPParams\* params);**

**Remarks:**

This methods takes care of the 'on droppable' handler in the **current\_dropscript**, if supplied. If the handler returns false, the not-droppable cursor is shown.

**Parameters:**

**FPParams\* params**

The set of arguments for the handler.

**Return Value:**

TRUE if droppable script will be accepted, otherwise FALSE.

**Prototype:**

**BOOL RunDropScriptDragOver(FPParams\* params);**

**Remarks:**

This methods takes care of the 'on droppable' handler in the **current\_dropscript**, if supplied, during the process of dragging contents over the drop target. If the handler returns false, the not-droppable cursor is shown.

**Parameters:**

**FPParams\* params**

The set of arguments for the handler.

**Return Value:**

TRUE if droppable script will be accepted, otherwise FALSE.

**Prototype:**

**BOOL RunDropScriptDrop(FPParams\* params);**

**Remarks:**

This methods takes care of the 'on droppable' handler in the **current\_dropscript**, if supplied and handles the parsing of the dropped script. If the handler returns false, the not-droppable cursor is shown.

**Parameters:**

**FPParams\* params**

The set of arguments for the handler.

**Return Value:**

TRUE if droppable script will be accepted, otherwise FALSE.

**Prototype:**

```
void InitDragDropCheck(MacroEntry* dropscript, LPARAM  
mousePt, WPARAM keyState, HWND hwnd);
```

**Remarks:**

This method will initialize a drag and drop check.

**Parameters:**

**MacroEntry\* dropscript**

The drop script macro entry.

**LPARAM mousePt**

The initial mouse cursor position.

**WPARAM keyState**

The initial state of the keyboard.

**HWND hwnd**

The handle to the initial start window.

**Prototype:**

```
bool ReadyToDrag();
```

**Remarks:**

This method returns TRUE if the system is ready to drag, otherwise FALSE.

**Default Implementation:**

```
{ return current_dropscript != NULL; }
```

## List of DropTypes

See Also: [Class DropType](#) , [Class DragAndDropHandler](#)

The following is the list of built-in DropTypes provided with 3ds max. The DropTypes are listed by their class name and DropType code. This type code can be obtained by using the **DropType::TypeCode()** method. Third-party DropType subclasses should use random codes above **0x1000000**.

### **FileDropType (FILE\_DROPTYPE)**

Intermediate base class for drop content in the form of a package of file names or URL's.

### **SceneFileDropType (SCENEFILE\_DROPTYPE)**

The \*.max scene file.

### **ImageFileDropType (IMAGEFILE\_DROPTYPE)**

The image files (.bmp, .tga, etc.).

### **ImportFileDropType (IMPORTFILE\_DROPTYPE)**

The importable files (.3ds, .dxf, etc.).

### **ScriptFileDropType (SCRIPTFILE\_DROPTYPE)**

The script files (.ms, .mse, .mcr).

### **DropScriptDropType (DROPSCRIPT\_DROPTYPE)**

Intermediate base class for drop content in the form of a dropScript.

### **DropScriptFileDropType (DROPSCRIPTFILE\_DROPTYPE)**

The drop script files (.ds, .dse).

### **MSZipPackageFileDropType (MSZIPFILE\_DROPTYPE)**

The script zip package files (.mzp).

### **BitmapDropType (BITMAP\_DROPTYPE)**

The bitmap type.

## Structure Color48

See Also: [Class Structure Color24](#), [Structure Color64](#).

Note: **typedef unsigned short USHORT;**

```
struct Color48 {
```

```
    UWORD r,g,b;
```

16 bits for each of the Red, Green and Blue components.

```
};
```

## Structure SceneInfo

See Also: [Class MaxNetManager](#), [Structure MaxJob](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to store information about the scene.

### typedef struct {

#### **int objects;**

The number of objects in the scene.

#### **int faces;**

The total number of faces in the scene.

#### **int lights;**

The total number of lights in the scene.

#### **int start;**

The scene start time.

#### **int end;**

The scene end time.

#### **DWORD flags;**

This variable contains all the flags relating to the scene's rendering options, which are following values:

#### **SCENE\_SHADOWMAPPED**

The shadowmapped flag.

#### **SCENE\_RAYTRACED**

The raytraced flag.

#### **SCENE\_VIDEOCOLORCHECK**

The video color check flag.

#### **SCENE\_TWOSIDED**

The render two sided flag..

#### **SCENE\_RENDERHIDEN**

The render hidden objects flag.

#### **SCENE\_RENDERATMOSPHER**

The render atmospheric effects flag.

**SCENE\_SUPERBLACK**

The render super black flag.

**SCENE\_RENDERALPHA**

The render alpha data flag.

**SCENE\_SERIALNUMBERING**

The serial numbering flag.

**SCENE\_DITHER256**

The dither 256 color flag.

**SCENE\_DITHERTRUE**

The dither true color flag.

**SCENE\_RENDERFIELDS**

The render fields flag.

**SCENE\_DISPLACEMENT**

The render displacement flag.

**SCENE\_EFFECTS**

The render effects flag.

**SCENE\_FIELDORDER**

The field ordering, 0 for even and 1 for odd.

**} SceneInfo;**

## Structure MaxJobRenderElements

See Also: [Class MaxNetManager](#), [Structure MaxJob](#)

### Description:

This structure is available in release 4.0 and later only.

This structure is used by the Network Rendering API to enable or disable Render Elements.

**typedef struct {**

**bool enabled;**

    This flag determines if render elements are enabled or not.

**} MaxJobRenderElements;**

## Structure TextBufferOutput

See Also: [Class MaxNetManager](#), [Structure JobText](#)

### Description:

This structure is available in release 4.0 and later only.

This structure contains the actual information for the output text buffer.

**typedef struct {**

**bool device;**

The device flag will be set to TRUE if the output is sent to a device instead of a file.

**float gamma;**

The output gamma for the device, which is optional. If the gamma value is set to 0.0, this value will be ignored.

**TCHAR data[256];**

The text output buffer.

**} TextBufferOutput;**

## Structure JobRenderElement

See Also: [Class MaxNetManager](#), [Structure JobText](#)

### Description:

This structure is available in release 4.0 and later only.

This structure contains the details on a specific Render Element for a job.

**typedef struct {**

**bool enabled;**

This flag indicates if the Render Element is enabled or disabled.

**bool filterenabled;**

This flag indicates if filters are enabled or disabled for the Render Element.

**bool atmosphere\_applied;**

This flag indicates if atmospheric effects are enabled or disabled for the Render Effect.

**bool shadows\_applied;**

This flag indicates if shadows are to be applied for the Render Effect.

**TCHAR name[128];**

The name of the Render Effect.

**TCHAR output[MAX\_PATH];**

The output path and file name of the resulting Render Element.

**} JobRenderElement;**