Microsoft DirectX 8.1

Common Framework

This source code is furnished to help you get up and running with DirectX more quickly. It is intended to illustrate how Direct3D works, and reduce the time it takes to build simple applications that allow you to experiment. It is used by the samples in the SDK. All of this source code lives in the Samples\Multimedia\Common\Src directory which can be installed during the SDK install.

This code is not intended to be cut and paste into a production application. This is not because it is poorly written, but rather that it is not designed for production. No attempt has been made to optimize it for performance. Very little error checking has been added as it obscures the functionality. Use to code to experiment with Direct3D to understand how it works.

Classes	Names	
	CD3DApplication	
	CD3DArcBall	
	CD3DCamera	
	CD3DFrame	
	CD3DFile	
	CD3DFont	
	CD3DMesh	
	CD3DScreenSaver	
Functions	Names	
	D3DUtil_CreateTexture	
	D3DUtil_CreateVertexShader	
	D3DUtil_GetCubeMapVertex	
	D3DUtil_GetRotationFromCursor	
	D3DUtil_InitMaterial	
	D3DUtil_InitLight	
	D3DUtil_SetColorKey	

D3DUtil_SetDeviceCursor

StructuresNamesD3DModeInfoD3DWindowedModeInfoD3DDeviceInfoD3DAdapterInfoMonitorInfoRenderUnit

Macros Names

Error Codes Names

Direct3DX Shader Assemblers Reference

This section contains reference information for the vertex and pixel shader assemblers provided by the Direct3DX utility library.

- <u>Vertex Shader Assembler Reference</u>
- <u>Pixel Shader Assembler Reference</u>

All of the example syntax used in this documentation is intended to demonstrate how to write shaders when using the Direct3DX vertex and pixel shader assemblers.

Introducing DirectX 8.1

Microsoft® DirectX® is a set of low-level application programming interfaces (APIs) for creating games and other high-performance multimedia applications. It includes support for two-dimensional (2-D) and three-dimensional (3-D) graphics, sound effects, music, input devices, and networked applications such as multiplayer games.

This document provides introductory information about DirectX 8.1. Information is divided into the following sections.

- What's New in DirectX 8.1
- DirectX 8.1 Components
- Using the DirectX 8.1 Documentation
- DirectX Tools
- <u>Programming DirectX with C/C++</u>
- Further Information

What's New in DirectX 8.1

Microsoft® DirectX® 8.1 is a major release primarily for graphics. It includes several new features for graphics, and bug fixes for Microsoft DirectInput® and Microsoft DirectPlay®.

New Tools in DirectX

- AppWizard. This tool provides an application wizard for creating a DirectX application with any combination of DirectX components. This tool is installed during the software development kit (SDK) install and can be accessed within Microsoft Visual Studio® when creating a new application. For more information see <u>AppWizard</u>.
- Error Lookup Tool. Use this tool to take the hexadecimal error codes and look up a text-based error message. This tool is installed as part of the SDK install and can be accessed from **Start, Programs, Microsoft DirectX 8.1 SDK, DirectX Utilities, DirectX Error Lookup**.

New Features in DirectX Graphics

- Expanded pixel shader functionality with new version 1.2, 1.3, and 1.4.
- Expanded the functionality of the Direct3DX (D3DX) utility library for meshes, textures, bump mapping, textures, and quaternions.
- MeshView. This tool provides an easy way to load, view, and modify meshes, and generally exercise D3DX functionality on progressive meshes. For more information about the tool, see <u>Mesh View Help</u>.
- A new <u>screen saver sample</u> is included that is built on the screen saver sample framework. The new framework includes support for multiple monitors.
- Added new samples.
- Reorganized the documentation into reference and programming guides. Expanded shaders and effects sections.

For more information see <u>What's New in Graphics</u>, or see each of the components.

New Features in DirectInput

Version 8.1 primarily includes performance improvements for DirectInput.

For more information, see <u>What's New in DirectInput</u>.

New Features in DirectPlay

Version 8.1 primarily includes performance improvements for DirectPlay.

For more information, see <u>What's New in DirectPlay</u>.

New Features in DirectX 8.0

Complete integration of DirectDraw and Direct3D

Microsoft DirectDraw[®] and Microsoft Direct3D[®] are merged into a single DirectX Graphics component. The application programming interface (API) has been extensively updated to make it even easier to use and to support the latest graphics hardware.

DirectMusic and DirectSound more integrated

Microsoft DirectMusic[®] and Microsoft DirectSound[®] are more tightly integrated than with DirectX 7.0. Wave files or resources can now be loaded by the DirectMusic loader and played through the DirectMusic performance, synchronized with MIDI notes.

DirectPlay updated

The DirectPlay component has been extensively updated to increase its capabilities and improve its ease-of-use. In particular, DirectPlay now supports voice communication between players.

DirectInput updated

DirectInput introduces one major new feature: action mapping. Action mapping enables you to establish a connection between input actions and input devices, which does not depend on the existence of particular device objects. It simplifies the input loop and reduces the need for custom game drivers, custom device profilers, and custom configuration of user interfaces in games.

DirectShow included in DirectX

Microsoft® DirectShow® is now part of DirectX and has been updated for this release.

Debug build available

You can use the <u>DirectX Control Panel Application</u> to switch between the debug and retail builds of DirectInput, Direct3D, and DirectMusic. To enable this feature, select Debug when you install the software development kit (SDK). This option installs both debug and retail dynamic-link libraries (DLLs) on your system. The Retail option installs only the retail DLLs.

DirectX 8.1 Components

Microsoft® DirectX® 8.1 is made up of the following components.

- DirectX Graphics combines the Microsoft DirectDraw® and Microsoft Direct3D® components of previous DirectX versions into a single application programming interface (API) that you can use for all graphics programming. The component includes the Direct3DX utility library that simplifies many graphics programming tasks.
- DirectX Audio combines the Microsoft DirectSound® and Microsoft DirectMusic® components of previous DirectX versions into a single API that you can use for all audio programming.
- Microsoft DirectInput[®] provides support for a variety of input devices, including full support for force-feedback technology.
- Microsoft DirectPlay® provides support for multiplayer networked games.
- Microsoft DirectShow® provides for high-quality capture and playback of multimedia streams.
- Microsoft DirectSetup is a simple API that provides one-call installation of the DirectX components.

Using the DirectX 8.1 Documentation

The following conventions are used in the syntax of methods, functions, and other API elements, as well as in explanatory material and sample code.

Convention	Meaning
Italic text	Denotes a placeholder or variable. You must provide the actual value. For example, the statement SetCursorPos(X , Y) requires you to substitute values for the X and Y parameters.
Bold text	Denotes a function, procedure, structure, macro, interface, method, data type, or other keyword in the programming interface or language.
[]	Encloses optional parameters.
•••	Specifies that the preceding item may be repeated.
FULL BOLD CAPITALS	Used for most type and structure names.
FULL CAPITALS	Used for enumeration values, flags, and constants.
monospace	Used for code examples and syntax spacing.
•	
•	Represents an omitted portion of a sample application.

Further Information

You can find further explanations of the graphics and multimedia concepts and terms discussed throughout the Microsoft® DirectX® documentation, as well as information on Microsoft® Windows® programming in general, in the following sources:

- Bargen, Bradley and Peter Donnelly, *Inside DirectX*, Microsoft® Press®, 1998.
- Begault, Durand R., *3-D Sound for Virtual Reality and Multimedia*, Academic Press, 1994.
- Blinn, James, *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kaufmann, 1996.
- Dodge, Charles and Thomas A. Jerse, *Computer Music: Synthesis, Composition, and Performance*, Schirmer Books, 1997 (2nd edition).
- Foley, James D., *Computer Graphics: Principles and Practice*, Addison-Wesley, 1991 (2nd edition).
- Hearn, Donald and M. Pauline Baker, *Computer Graphics*, Prentice-Hall, 1986.
- Kientzle, Tim, *A Programmer's Guide to Sound*, Addison-Wesley Developers Press, 1998.
- Kovach, Peter J., *Inside Direct3D*, Microsoft Press, 2000.
- Petzold, Charles, *Programming Windows 98*, Microsoft Press, 1998 (5th edition).
- Thompson, Nigel, *3D Graphics Programming for Windows*, Microsoft Press, 1996.
- Watt, Alan H., and Mark Watt, *Advanced Animation and Rendering Techniques*, Addison-Wesley, 1992.

Additional sources for the concepts and terms associated with COM can be found in the following sources:

- Brockschmidt, Kraig, *Inside OLE 2*, Microsoft Press, 1995 (2nd edition).
- Rogerson, Dale E., *Inside COM*, Microsoft Press, 1997.

Mesh View Help

Menu Descriptions

File

• Open Mesh File

Opens a dialog to select a file in the .x or .m file format to be loaded and viewed.

• Open PMesh File

Opens a dialog to select a file in the progressive mesh format to be loaded and viewed.

• Create Shape

Opens a sub-dialog to create some basic shapes that are defined programmatically (text, polygon, box, cylinder, torus, teapot, sphere, cone).

• Save Mesh As

Opens a dialog to save the selected mesh to a file. The file can be written as a text or binary file.

• Close Selected

Closes and deletes the currently selected mesh.

• Close Non Selected

Closes and deletes the meshes that are not currently selected.

View

• Wireframe

View all content in wireframe mode.

• Edges

View all content in solid-shaded mode with the edges drawn in black.

• Creases

Highlight the creases on the visible meshes. A crease is an edge with a vertex that has a different piece of data on it for multiple faces that refer to it, that is, a different normal for the vertex per face.

• Strips

Show the strips that are generated by this mesh in blue. The blue line goes from the center of each triangle to the next triangle in the triangle strip.

• Adjacency

Show the adjacency of the polygons in a mesh by drawing a line from the center of a polygon to the center of the adjacent polygon.

• Bounding Box

Draw the bounding boxes for the visible meshes.

• Normals

Draw the normals of the vertices on the visible meshes in yellow.

• Texture Coords

Show the texture coordinates for the viewed geometry as rays projecting from the vertices. Because a vertex in Microsoft® Direct3D® can have up to eight texture coordinates, users must specify which sets they would like to view. This viewing mode is especially useful when the texture coordinates are filled with tangents for use in pixel shaders for example.

• Textures

Display the textures on the visible geometry.

• Lighting

Show the geometry in the scene with lighting calculations still on.

• Culling

Perform back face culling on the visible geometry when disabled polygons facing away from the camera are not drawn.

• Hierarchy

Display the frame hierarchy of the meshes that are currently loaded. This is displayed in a separate floating window. To make the window disappear, on the **View** menu, clear the **Hierarchy** command.

• Play Animation

Play the current animation for the currently loaded geometry if one exists.

• Pause Animation

Stop the animation at the current frame when playing.

• Normal Speed

Interpret the time value in the animation from an X file as 4800 units per second. Otherwise, the interpreted value is 30 units per second.

MeshOps

• Optimize

Optimizes the currently selected mesh with the selected optimization method. See the Microsoft® Direct3DX reference pages to see the differences in methods.

• Weld Vertices

Removes duplicate vertices and makes polygons that use these vertices use the nondeleted vertex.

• Split Mesh

Splits the selected mesh into multiple meshes that are less than specified size in vertices and faces.

• Collapse Meshes

Collapses the currently selected meshes into a single mesh.

• Reset Matrices

Resets the matrices for the frames that are loaded to their initial position.

• Mesh Properties

Shows the selected mesh's FVF render states and whether or not it is a 32 bit mesh.

• Skinning Method

Allows the user to select the skinning method while animating a skinned mesh. The choices are nonindexed, indexed, and software skinning.

• Face Selection

Enters a mode for the user to select an individual face on the current mesh.

• Vertex Selection

Enters a mode for the user to select an individual vertex on the current mesh.

PMeshes

• Convert Selected to PM

Convert the selected mesh to a progressive mesh. The conversion uses the error parameters that are entered in a dialog box. For more information on these parameters refer to the Direct3DX documentation of progressive meshes.

• Snapshot to Mesh

Convert the current progressive mesh object to a static mesh object using the current settings of the progressive mesh.

• Set number of Faces

Set the current number of faces in the progressive mesh to a specific number.

• Set number of Vertices

Set the current number of vertices in the progressive mesh to a specific number.

• Trim

Set the minimum and maximum number of faces for a progressive mesh. Once the user has set the trim values to the desired minimum and maximum, the progressive mesh can be trimmed to the selected values, thereby reducing the dynamic range of the progressive mesh.

N-Patches

• N-Patch Selected

Draw the current object as an N-Patches object. The scroll bar in this mode selects the amount of N-Patch iterations for the current object.

• SnapShot to mesh

Convert the selected object to a static mesh based on the current N-Patch settings to create a high-resolution static mesh.

Icons and Usage

Icons

• Selection Modes

The first three icons are easy ways for the user to select the selection mode. They are Mesh Selection Mode (Arrow), Face Selection Mode (yellow outlined triangle), and Vertex Selection Mode (Red point highlighted triangle). These are the same modes that are available from the menus in MeshOps.

• Display Modes

The next icons are easy ways to select the most common display modes for geometry. They are Shaded mode (nonoutlined tri-color cube), Wire frame mode (wire frame cube), and Edge mode (outlined tri-color cube). These are the same modes that are available from the menus in View.

• Topology Display modes

The next icons display specific topological information about the geometry displayed. They are adjacency (A), Strips (S), Creases (C), and Normals (N). These are the same modes that are available from the menus in View.

• Info

The next icon is the info button that will display information about the currently selected element.

• Animation Controls

The last two icons are animation controls for playing and pausing the animation for the currently visible mesh.

Status Bar

The status bar in MView displays the current status of the visible geometry. The

order from left to right of the displayed information is currently selected element (face or vertex only), Mesh mode (polygon, Pmesh, or pMesh), display frames per second, display triangles per second, number of displayed triangles, and number of displayed vertices.

Scroll Bar

The scroll bar will appear in two of the three mesh modes, pMesh and nPatch mode. In pMesh mode, the scroll bar indicates the range of displayed triangles for the progressive mesh. You can slide the scroll bar up or down to change the number of triangles displayed. In nPatch mode, the scroll bar indicates how many nPatch levels are being used. As the scroll bar is moved up or down, the number of nPatch interactions performed are adjusted accordingly.

More Information

For more information on any of the functions used by MView, refer to the Direct3DX documentation included in the Microsoft® DirectX® software development kit (SDK).

Moire Sample

Description

The moire sample shows how to use the Microsoft® DirectX® software development kit (SDK) screen saver framework to write a screen saver that uses Microsoft® Direct3D®. The screen saver framework is similar to the sample application framework, using many methods and variables with the same names. After writing a program with the screen saver framework, you end up with a fully-functional Microsoft® Windows® screen saver, rather than with a regular Windows application.

The moire screen saver appears as a mesmerizing sequence of spinning lines and colors. It uses texture transformation and alpha blending to create a highly animated scene, even though the polygons that make up the scene do not move at all.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\ScreenSavers\Moire

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

Moire.scr can be started in five modes: configuration, preview, full, test, and password-change. You can choose some modes by clicking the right mouse button (right-click) on the moire.scr file and choosing Configure or Preview. Or you can start moire.scr from the command line with the following command-line parameters:

- -c Configuration mode
- -t Test mode
- -p Preview mode
- -a Password-change mode
- -s Full mode

When the screen saver is running in full mode, press any key or move the mouse to exit.

Programming Notes

Programs that use the screen saver framework are very similar to programs that use the Direct3D sample application framework. Each screen saver needs to create a class derived from the main application class, **CD3DScreensaver**. To provide functionality specific to each screen saver, the screen saver implements its own versions of the virtual functions **FrameMove**, **Render**, **InitDeviceObjects**, and so forth.

Screen savers can be written to be multimonitor-compatible, without much extra effort. If you do not want your screen saver to run on multiple monitors, you can just set the *m_bOneScreenOnly* variable to TRUE. This value is set to FALSE by default. The function **SetDevice** will be called each time the device changes. The way that moire deals with this is to create a structure called *DeviceObjects*, which contains all device-specific pointers and values. CMoireScreensaver holds an array of *DeviceObjects* structures, called *m DeviceObjectsArray*. When **SetDevice** is called, *m pDeviceObjects* is changed to point to the *DeviceObjects* structure for the specified device. When rendering, *m* rcRenderTotal refers to the rendering area that spans all monitors, and *m_rcRenderCurDevice* refers to the rendering area for the current device's monitor. The function **SetProjectionMatrix** shows one way to set up a projection matrix that makes proper use of these variables to either render a scene that spans all the monitors, or display a copy of the scene on each monitor. The projection matrix used depends on the value of *m_bAllScreensSame*, which you can enable the user to control in the configuration dialog.

The **ReadSettings** function is called by the screen saver framework at program startup time, to read various screen saver settings from the registry. **DoConfig** is called when the user wants to configure the screen saver settings. The program should respond to this by creating a dialog box with controls for the various screen saver settings. This dialog box should also have a button called **Display Settings** which, when pressed, should call **DoScreenSettingsDialog**. This common dialog box allows the user to configure what renderer and display mode should be used on each monitor. You should set the member variable *m_strRegPath* to a registry path that will hold the screen saver's settings. You can use this variable in your registry read/write functions. The screen saver

framework will also use this variable to store information about the default display mode in some cases.

This sample uses common DirectX code that consists of programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

What's New in DirectX Graphics

This section describes Microsoft® DirectX® graphics features that are new in DirectX 8.x.

To see what's new in all of DirectX, see <u>What's new in DirectX 8.1</u>.

New Features in DirectX 8.1

Pixel Shaders

• Added pixel shader versions 1.2, 1.3, and 1.4. These new versions expand existing functionality through a more powerful set of <u>instructions</u>, <u>registers</u>, and <u>modifiers</u> for programming pixel shaders.

New D3DX Functionality

Mesh API

- Added mesh functionality to improve performance using <u>D3DXConvertMeshSubsetToStrips</u> and <u>D3DXConvertMeshSubsetToSingleStrip</u>. Use the <u>OptimizedMesh</u> sample to understand improving mesh performance.
- Improved support for progressive meshes with <u>OptimizeBaseLOD</u>, <u>TrimByVertices</u> and <u>TrimByFaces</u>.
- Added <u>D3DXSplitMesh</u> to help split meshes into smaller meshes.

Bump Mapping

- Added <u>D3DXComputeTangent</u> to create a per-vertex coordinate system based on texture coordinate gradients.
- <u>D3DXComputeNormalMap</u> converts a height field to a normal map.

MeshView Tool

• This tool provides an easy way to load, view, and modify meshes, and generally exercise Direct3DX (D3DX) functionality on progressive meshes. This tool is installed as part of the software development kit (SDK) install and can be accessed from Start/Programs/Microsoft DirectX 8.1 SDK/DirectX Utilities/DirectX MeshView. For more information about the tool, see <u>Mesh View Help</u>.

Effect Framework

• Effects. Added string support, added comments, and removed FourCC

constraints.

• Effect framework API. Support for state saving and restoring, support for handling OnLost and OnReset, support Set*() after Begin(). All of the ID3DXTechnique functionality has been moved into the ID3DXEffect interface to simplify the effect interface.

Texture Library

- Implemented a higher quality DXTn encoding algorithm.
- Use D3DXGetImageInfoFrom to get image information before loading it.
- Includes support for dynamic textures.
- <u>D3DXSaveSurfaceToFile</u> supports 8-bit paletted .bmp files and 24-bit RGB .dds files in all formats: mipmaps, cube maps, volumes.
- D3DPOOL_SCRATCH allows creation of resources that are not limited by device capabilities. They can be created and destroyed, locked and unlocked. These resources can be set to a device and used in rendering. Use with D3DX to convert to something useable such as loading a high-precision height field and converting to a normal map.
- Texture fill functions, <u>D3DXFillTexture</u>, <u>D3DXFillCubeTexture</u>, and <u>D3DXFillVolumeTexture</u>.

Math Library

- New math functions: Additional support for spherical quadratic quaternion interpolation using <u>D3DXQuaternionSquadSetup</u>. Use it with <u>D3DXQuaternionSquad</u>. <u>D3DXMatrixMultiplyTranspose</u> for matrices in vertex shaders and <u>D3DXFresnelTerm</u>.
- Math library. Added CPU specific optimizations for most important functions for 3DNow, SSE, and SSE2.
- Support for 16-byte aligned matrices using D3DXMATRIXA16.

Samples

- Several new <u>samples</u> have been included to demonstrate culling, lighting, volume fog, and self-shadowing using a shadow volume.
- A new <u>screen saver sample</u> is included that is built on the screen saver sample framework. The new framework includes support for multiple monitors.

Documentation Upgrades

- <u>DXGraphics SDK Docs</u>. The graphics SDK documentation has been reorganized into two sections: a <u>Reference</u> section and a <u>Programmers Guide</u>.
- New sections for creating programmable <u>vertex shaders</u>, <u>pixel shaders</u>, and <u>effects</u> have been added to the Programmers Guide.
- The global illumination equations and the <u>Mathematics of Lighting</u> section have been rewritten and examples included.

New Features in DirectX 8.0

This version maintains backward compatibility by exposing and supporting objects and interfaces offered by previous releases of DirectX. However, many new features and performance enhancements have also been added to the Microsoft Direct3D® API interfaces.

Pixel and Vertex Shaders

• The two programmable sections of the Direct3D architecture are vertex shaders and pixel shaders. Vertex shaders are invoked prior to vertex assembly and operate on vertices. Pixel shaders are invoked after any DrawPrimitive or DrawIndexedPrimitive calls and generate the pixels that are written to the render target. The addition of programmable shaders for vertex and pixel operations provides the framework for real-time programmable effects that rival movie quality. The innovative freedom that this programmability gives back to game developers—by allowing them to implement whatever effect they see fit with the programmable pipeline—has the potential to unlock a new round of incredible games. Pixel and vertex shaders can be written using ASCII files, thus the shader files can be updated at runtime without recompiling the source application.

Effects

• Allows you to change how an object is rendered, based on the hardware capabilities of the machine your application is running on. Effects are written using ASCII files, thus the effect file can be updated at runtime without recompiling the source application.

Complete Integration of DirectDraw and Direct3D

• Simplifies application initialization and improves data allocation and management performance, which reduces the memory footprint. Also, the integration of the graphics APIs enable parallel vertex input streams for more flexible rendering.

Multisampling Rendering Support

• Enables full-scene antialiasing and multisampling effects, such as motion blur and depth-of-field.

Point Sprites

• Enables high-performance rendering of particle systems for sparks, explosions, rain, snow, and so on.

3-D Volumetric Textures

• Enables range-attenuation in per-pixel lighting and volumetric atmospheric effects, and can be applied to more intricate geometry.

Higher-Order Primitive Support

• Enhances the appearance of three-dimensional (3-D) content and facilitates the mapping of content from major 3-D authoring tools.

Higher-Level Technologies

• Includes 3-D content-creation tool plug-ins (for export to Direct3D) for skinned meshes that use a variety of Direct3D techniques, multiresolution level-of-detail (LOD) geometry, and higher-order surface data.

Indexed Vertex Blending

• Extends geometry blending support to allow the matrices used for vertex blending to be referred to using a matrix index.

Expansion of the Direct3DX Utility Library

• Contains a wealth of new functionality. The Direct3DX utility library is a helper layer that sits on top of Direct3D to simplify common tasks encountered by 3-D graphics developers. Includes a skinning library, support for working with meshes, and functions to assemble vertex and pixel shaders. About 20 new functions have been added for DirectX 8.1. Note that the functionality supplied by D3D_OVERLOADS, first introduced with DirectX 5.0, has been moved to the Direct3DX utility library.

What's New in DirectInput

New Features for DirectInput 8.1

Microsoft® DirectX® 8.1 is a major release primarily for DirectX graphics. The improvements for Microsoft DirectInput® are primarily performance enhancements.

To find out more about the new features in DirectX, see <u>What's New in DirectX</u> <u>8.1</u>.

New Features for DirectInput 8.0

The following are some of the new features in DirectInput.

Action mapping

DirectInput for DirectX 8.0 introduces a major new feature: action mapping. Action mapping enables you to establish a connection between input actions and input devices that does not depend on the existence of particular device objects (such as specific buttons or axes). Action mapping simplifies the input loop and reduces the need for custom game drivers, custom device profilers, and custom configuration user interfaces in games.

For more information, see <u>Action Mapping</u>.

New DirectInput object features

The DirectInput object is now represented by the **IDirectInput8** interface. A new helper function, **DirectInput8Create**, creates the object and retrieves this interface. **IDirectInput8** has a new CLSID and cannot be obtained by calling **QueryInterface** on an interface to objects of the class **CLSID DirectInput** used in earlier DirectX versions.

New keyboard properties

Two keyboard properties have been added: DIPROP_KEYNAME, which retrieves a localized key name, and DIPROP_SCANCODE, which retrieves the scan code.

Joystick slider data in rglSlider array

Joystick slider data that was assigned to the z-axis of a <u>DIJOYSTATE</u> or <u>DIJOYSTATE2</u> structure under earlier DirectX versions will now be found in the **rglSlider** array of those same structures.

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

Instructions

Pixel shaders contain the following types of instructions.

- A *Version* instruction defines the shader version. There is only one version instruction in a shader and it is the first instruction in a shader.
- *Constant* instructions define constants. These instructions must be after the version instruction and before any arithmetic or texture address instructions.
- A *Phase* instruction splits a shader into two sections: phase 1 and phase 2. Each phase has separate arithmetic and texture address instruction limits. Version 1.4 is the only version that supports the phase instruction.
- *Arithmetic* instructions include common mathematical operations such as add and subtract, multiply, and taking a dot product.
- *Texture Address* instructions manipulate texture coordinate data that is associated with texture stages.

The instructions are listed below.

Version instructions	Version					
	1.() 1. 1	l 1.2	2 1.3	3 1.4	
<u>ps</u>	Х	Х	Х	Х	Х	
Constant instructions	Version					
	1.() 1. 1	l 1.2	2 1.3	3 1.4	
def	Х	Х	Х	Х	Х	
Dhase instructions	Version					
Phase instructions	1.() 1. 1	l 1.2	2 1.3	3 1.4	
<u>phase</u>					Х	
Arithmetic instructions	Version					
	1.0) 1. 1	l 1.2	2 1.3	3 1.4	
add	Х	Х	Х	Х	Х	
bem					Х	
<u>cmp</u>			Х	Х	Х	
<u>cnd</u>	Х	Х	Х	Х	Х	
<u>dp3</u>	Х	Х	Х	Х	Х	
<u>dp4</u>			Х	Х	Х	

lrp	Х	Х	Х	Х	х
mad	Х	Х	Х	Х	х
mov	Х	Х	Х	Х	Х
mul	Х	Х	Х	Х	Х
nop	Х	Х	Х	Х	Х
sub	Х	Х	Х	Х	X
T		V	ersi	on	
Texture address instructions	1.0	1.1	1.2	1.3	3 1.4
tex	Х	Х	Х	Х	
<u>texbem</u>	х	х	х	х	
texbeml	х	х	х	х	
texcoord	х	х	х	х	
texcrd					Х
<u>texdepth</u>					Х
<u>texdp3</u>			х	х	
<u>texdp3tex</u>			Х	Х	
<u>texkill</u>	Х	Х	Х	х	Х
texld					Х
<u>texm3x2depth</u>				Х	
<u>texm3x2pad</u>	Х	Х	Х	Х	
texm3x2tex	Х	Х	Х	Х	
texm3x3			Х	Х	
texm3x3pad	Х	Х	Х	Х	
<u>texm3x3tex</u>	Х	Х	Х	Х	
<u>texm3x3spec</u>	Х	Х	Х	Х	
<u>texm3x3vspec</u>	Х	Х	Х	Х	
<u>texreg2ar</u>	Х	Х	Х	Х	
texreg2gb	Х	Х	Х	Х	
<u>texreg2rgb</u>			Х	Х	

Microsoft Directx 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

Registers

Registers hold data for use by the pixel shader. Registers are fully described in the following sections.

- <u>Register Types</u>. Describes the four types of registers available and their purposes.
- <u>Read Port Count</u>. Details the restrictions on using multiple registers in a single instruction.
- <u>Read-only, Read/Write</u>. Describes which registers can be used for reading, writing, or both.
- <u>Range</u>. Details the range of the component data.

Register Types

Registers transfer data to the shader ALU and store temporary results. The table below identifies the four types of registers and the number available in each shader version.

Register name	Туре	Versions						
Register name		1.0	1.1	1.2	1.3	1.4		
c _n	Constant register	8	8	8	8	8		
r _n	Temporary register	2	2	2	2	6		
t _n	Texture register	4	4	4	4	6		
v _n	Color register	2	2	2	2	2 in <u>phase</u> 2		

- Constant registers contain constant data organized in four fixed-point values. Data can be loaded into a constant register using
 SetPixelShaderConstant or it can be defined using the def shader instruction. Constant registers are not usable by texture address instructions. The only exception is the texm3x3spec instruction, which uses a constant register to supply an eye-ray vector.
- **Temporary registers** are used to store intermediate results, as four fixed-point values. r0 additionally serves as the pixel shader output. The value in r0 at the end of the shader is the pixel color for the shader.

Shader pre-processing will fail <u>CreatePixelShader</u> on any shader that attempts to read from a temporary register that has not been written by a previous instruction. <u>D3DXAssembleShader</u> will fail similarly, assuming validation is enabled (do not use *D3DXASM_SKIPVALIDATION*).

• Texture registers

• Pixel shader version 1.1 to 1.3

For pixel shader version 1.1 to 1.3, texture registers contain texture data, organized in four fixed-point values. Texture data is loaded into a texture register when a texture is sampled. Texture sampling uses texture coordinates to look up, or *sample*, a color value at the specified (u,v,w,q) coordinates while taking into account the texture stage state attributes. The texture coordinate data is interpolated from the vertex texture coordinate data and is associated with a specific texture stage. There is a default one-to-one association between texture stage number and texture coordinate declaration order. By default, the first set of texture coordinates defined in the vertex format is associated with texture stage 0.

For these pixel shader versions, texture registers behave just like temporary registers when used by arithmetic instructions. For pixel shader version 1.0, texture registers are read-only to arithmetic instructions.

• Pixel shader version 1.4

For pixel shader version 1.4, texture registers (t#) contain read-only texture coordinate data. This means that the texture coordinate set and the texture stage number are independent from each other. The texture stage number (from which to sample a texture) is determined by the destination register number (r0 to r5). For the texture instruction, the texture coordinate set is determined by the source register (t0 to t5), so the texture coordinate set can be mapped to any texture stage. In addition, the source register (specifying texture coordinates) for texture contents of the temporary register are used as texture coordinates.

For this pixel shader version, texture registers contain texture

coordinate data and are also available to texture addressing instructions as source parameters.

• **Color registers** contain per-pixel color values. The values are obtained by per-pixel iteration of the diffuse and specular color values in the vertex data. Color registers store data in four fixed-point values. For pixel shader version 1.4 shaders, color registers are available only during the second phase.

If the shade mode is set to D3DSHADE_FLAT, the application iteration of both vertex colors (diffuse and specular) is disabled. Regardless of the shade mode, fog will still be iterated by the pipeline if pixel fog is enabled. Keep in mind that fog is applied later in the pipeline than the pixelshader.

It is common to load the v0 register with the vertex diffuse color data. It is also common to load the v1 register with the vertex specular color data.

Input color data values are clamped (saturated) to the range 0 through 1 because this is the valid input range for color registers in the pixel shader.

Pixel shaders have read only access to color registers. The contents of these registers are iterated values, but iteration is performed at much lower precision than texture coordinates.

Read Port Limit

The read port limit specifies the number of different registers of each register type that can be used as a source register in a single instruction.

Dogistor nomo	Type	Versions					
Register name	Туре	1.0	1.1	1.2	1.3	1.4	
c _n	Constant register	2	2	2	2	2	
r _n	Temporary register	2	2	2	2	3	
t _n	Texture register	1	2	3	3	1	
v _n	Color register	1	2	2	2	2	

For example, the color registers for almost all versions have a read port limit of

two. This means that a single instruction can use a maximum of two different color registers (v0 and v1 for instance) as source registers. This example shows two color registers being used in the same instruction. As shown in the table, two color registers can be used in every version except 1.0.

mad r0, v0, v1, v1 // This is valid for versions 1.1, 1.2, 1.3, 1.4

Any valid destination register can be used in the same instruction because read port count restrictions do not affect destination registers.

Destination registers are independent of the read port count restrictions.

For co-issued instructions, the maximum number of different registers (of the same type) that can be used across two co-issued instructions is three. This is true for all shader versions.

Read-only, Read/Write

The register types are identified according to read-only (RO) capability or read/write (RW) capability in the following table. Read-only registers can be used only as source registers in an instruction; they can never be used as a destination register.

Register name	e Type	Versions				
Register name		1.0	1.1	1.2	1.3	1.4
c _n	Constant register	RO	RO	RO	RO	RO
r _n	Temporary register	RW	RW	RW	RW	RW
t _n	Texture register	See following note	RW	RW	RW	RO
v _n	Color register	RO	RO	RO	RO	RO

Registers that are RW capable can be used to store intermediate results. This includes the temporary registers and texture registers for some of the shader versions.

Note

• For pixel shader version 1.0, texture registers are RW for texture addressing

instructions, but RO for arithmetic instructions.

• For pixel shader version 1.4, texture registers are RO for texture addressing instructions, and texture registers can be neither read from nor written to by arithmetic instructions. Also, because texture registers have become texture coordinate registers, having RO access is not a regression of previous functionality.

Range

The range is the maximum and minimum register data value. The ranges vary based on the type of register. The ranges for some of the registers can be queried from the <u>device caps</u> using <u>GetDeviceCaps</u>.

Register name	Туре	Range	Versions
c _n	Constant register	-1 to +1	All versions
r _n	Temporary register	- <u>MaxPixelShaderValue</u> to + <u>MaxPixelShaderValue</u>	All versions
t _n	Texture	 <u>MaxPixelShaderValue</u> to + <u>MaxPixelShaderValue</u> 	1.0 to 1.3
'n	register	- <u>MaxTextureRepeat</u> to + <u>MaxTextureRepeat</u>	1.4
v _n	Color register	0 to +1	All versions

Early pixel shader hardware represents data in registers using a fixed-point number. This limits precision to a maximum of approximately eight bits for the fractional part of a number. Keep this in mind when designing a shader.

For pixel shader version 1.0 to 1.3, <u>MaxTextureRepeat</u> must be a minimum of one.

For 1.4, <u>MaxTextureRepeat</u> must be a minimum of eight.

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

Modifiers

Modifiers are used to modify instructions, source registers, and destination registers.

- Instruction modifiers
- <u>Source register modifiers</u>
- <u>Source register selectors</u>
- <u>Destination register write masks</u>

Texture Register Modifiers

Pixel shader version 1.4 includes two new instructions, <u>texld</u> and <u>texcrd</u>, which contain custom register modifier functionality. These instructions support different register modifiers, register selectors, and register write masks. For more information, see <u>Texture Register Modifiers</u>.

D3DXConvertMeshSubsetToStrips

Convert the specified mesh subset into a series of strips.

HRESULT D3DXConvertMeshSubsetToStrips(

LPD3DXBASEMESH MeshIn, DWORD AttribId, DWORD IBOptions, LPDIRECT3DINDEXBUFFER8* ppIndexBuffer, DWORD* pNumIndices, LPD3DXBUFFER* ppStripLengths, DWORD* pNumStrips,

);

Parameters

MeshIn

[in] Pointer to a **ID3DXBaseMesh** interface, representing the mesh to convert to strips.

AttribId

[in] Attirbute ID of the mesh subset to convert to strips.

IBOptions

[in] A combination of one or more flags from the **D3DXMESH** enumeration, specifying options for the create index buffer.

ppIndexBuffer

[out] Pointer to an **ID3DXBuffer** object, representing the index buffer containing the strips.

pNumIndices

[out] Number of indices in the buffer returned in the *ppIndexBuffer* parameter.

ppStripLengths

[out] Buffer containing an array of one **DWORD** per strip, in the index buffer that specifies the length of that strip.

pNumStrips

[out] Number of individual strips in the index buffer and corresponding strip length array.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h. **Import Library:** Use D3dx8.lib.

D3DXConvertMeshSubsetToSingleStr

Converts the specified mesh subset into a single triangle strip.

HRESULT D3DXConvertMeshSubsetToSingleStrip(
 LPD3DXBASEMESH MeshIn,
 DWORD AttribId,
 DWORD IBOptions,
 LPDIRECT3DINDEXBUFFER8* ppIndexBuffer,
 DWORD* pNumIndices
);

Parameters

MeshIn

[in] Pointer to a **ID3DXBaseMesh** interface, representing the mesh to convert to a strip.

AttribId

[in] Attirbute ID of the mesh subset to convert to strips.

IBOptions

[in] A combination of one or more flags from the **D3DXMESH**

enumeration, specifying options for the create index buffer.

ppIndexBuffer

[out] Pointer to an **ID3DXBuffer** object, representing the index buffer containing the strip.

pNumIndices

[out] Number of indices in the buffer returned in the *ppIndexBuffer* parameter.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h. **Import Library:** Use D3dx8.lib.

OptimizedMesh Sample

Description

The OptimizedMesh sample illustrates how to load and optimize a file-based mesh using the Microsoft® Direct3DX mesh utility functions.

For more information on Direct3DX, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK root)\Samples\Multimedia\Direct3D\OptimizedMesh

Executable: (SDK root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
CTRL-O	Opens mesh file.
CTRL-M	Toggles optimized mesh.

Programming Notes

Many Microsoft Direct3D® samples in the DirectX SDK use file-based meshes. However, the OptimizedMesh sample is a good example of the basic code necessary for loading a mesh. The D3DX mesh loading functionality collapses the frame hierarchy of an .x file into one mesh.

For other samples, the bare bones D3DX mesh functionality is wrapped in a common class CD3DMesh. If you want to keep the frame hierarchy, you can use the common class CD3DFile.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK root)\Samples\Multimedia\Common.

D3DXSplitMesh

Splits a mesh into meshes smaller than the specified size.

```
HRESULT D3DXSplitMesh(
  CONST LPD3DXMESH pMeshIn,
                  pAdjacencyIn,
 CONST DWORD*
 CONST DWORD
                   MaxSize,
  CONST DWORD
                   Options,
  DWORD*
                   pMeshesOut,
 LPD3DXBUFFER*
                  ppMeshArrayOut,
  LPD3DXBUFFER*
                  ppAdjacencyArrayOut,
  LPD3DXBUFFER*
                  ppFaceRemapArrayOut,
  LPD3DXBUFFER*
                  ppVertRemapArrayOut,
```

```
);
```

Parameters

pMeshIn

[in] Pointer to an **ID3DXMesh** interface, representing the source mesh. *pAdjacencyIn*

[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh to be simplified.

MaxSize

[in] Maximum number of vertices or faces in the new mesh.

Options

[in] Option flags for the new meshes.

pMeshesOut

[out, retval] Number of meshes returned.

ppMeshArrayOut

[out, retval] Buffer containing an array of <u>ID3DXMesh</u> interfaces for the new meshes.

ppAdjacencyArrayOut

[out, retval] Buffer containing an array of adjacency arrays for the new meshes. This parameter is optional. Set it to NULL if it is unused.

ppFaceRemapArrayOut

[out, retval] Buffer containing an array of face remap arrays for the new meshes. This parameter is optional. Set it to NULL if it is unused.

ppVertRemapArrayOut

[out, retval] Buffer containing an array of vertex remap arrays for the new meshes. This parameter is optional. Set it to NULL if it is unused.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL D3DXERR_INVALIDDATA E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h. **Import Library:** Use D3dx8.lib.

D3DXComputeTangent

Computes a per vertex coordinate system based on texture coordinate gradients.

HRESULT D3DXComputeTangent(LPD3DXMESH InMesh, DWORD TexStage, LPD3DXMESH OutMesh, DWORD TexStageUVec, DWORD TexStageVVec, DWORD Wrap, DWORD* pAdjacency);

Parameters

InMesh

[in] Pointer to an **ID3DXMesh** interface, representing the input mesh. *TexStage*

[in] Texture coordinate set in input mesh to use for gradients.

OutMesh

[out] Pointer to an **ID3DXMesh** interface, representing the returned mesh. *TexStageUVec*

[in] Texture coordinate set in output mesh to receive U tangent vector. Set this value to D3DX_COMP_TANGENT_NONE if you do not want to generate a U tangent vector.

TexStageVVec

[in] Texture coordinate set in output mesh to receive V tangent vector. Set this value to D3DX_COMP_TANGENT_NONE if you do not want to generate a V tangent vector.

Wrap

[in] Set this value to 0 for no wrapping or 1 to wrap in the U and V directions.

pAdjacency

[out] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the created mesh.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL E_OUTOFMEMORY

Remarks

Setting both *TexStageVVec* and *TexStageWVec* to D3DX_COMP_TANGENT_NONE will cause this method to fail, since it has nothing to do.

Requirements

Header: Declared in D3dx8mesh.h. **Import Library:** Use D3dx8.lib.

D3DXComputeNormalMap

Converts a height map into a normal map. The (x,y,z) components of each normal are mapped to the (r,g,b) channels of the output texture.

```
HRESULT D3DXComputeNormalMap(
  LPDIRECT3DTEXTURE8 pTexture,
  LPDIRECT3DTEXTURE8 pSrcTexture,
  CONST PALETTEENTRY* pSrcPalette,
  DWORD Flags,
  DWORD Channel,
  FLOAT Amplitude
);
```

Parameters

pTexture

[out, retval] Pointer to an **IDirect3DTexture8** interface, representing the destination texture.

pSrcTexture

[in, retval] Pointer to an **IDirect3DTexture8** interface, representing the source height-map texture.

pSrcPalette

[in] Pointer to a **PALETTEENTRY** type that contains the source palette of 256 colors or NULL.

Flags

[in] One or more **D3DX_NORMALMAP** flags that control generation of normal maps.

Channel

[in, out] One **D3DX_CHANNEL** flag specifying the source of height information.

Amplitude

[in] Constant value by which the height information is multiplied.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be the following value.

D3DERR_INVALIDCALL

Remarks

This method computes the normal by using the central difference with a kernel size of 3×3 . RGB channels in the destination contain biased (x,y,z) components of the normal.

Requirements

Header: Declared in D3dx8tex.h. **Import Library:** Use D3dx8.lib.

ID3DXEffect

The methods from the **ID3DXTechnique** interface have been moved into the **ID3DXEffect** interface for Microsoft® DirectX® 8.1.

The **ID3DXEffect** interface is used to set and query effects, and to choose techniques. An effect object can contain multiple techniques to render the same effect.

The **ID3DXEffect** interface is obtained by calling **D3DXCreateEffect** or **D3DXCreateEffectFromFile**. The methods of the **ID3DXEffect** interface can be organized into the following groups.

Copying	<u>CloneEffect</u> <u>GetCompiledEffect</u>
Effect Parameters	<u>SetDword</u>
	<u>SetFloat</u>
	<u>SetMatrix</u>
	<u>SetPixelShader</u>
	SetString
	<u>SetTexture</u>
	<u>SetVector</u>
	<u>SetVertexShader</u>
	GetDword
	<u>GetFloat</u>
	<u>GetMatrix</u>
	GetParameterDesc
	<u>GetPixelShader</u>
	GetString
	<u>GetTexture</u>
	<u>GetVector</u>
	GetVertexShader
	<u>GetDesc</u>

Information	<u>GetDevice</u>
	<u>GetPassDesc</u>
Techniques	<u>FindNextValidTechnique</u>
	<u>GetTechnique</u>
	<u>GetTechniqueDesc</u>
	<u>SetTechnique</u>
	<u>Validate</u>
	<u>Begin</u>
Technique Application	End
	Pass
Miscellaneous	<u>OnLostDevice</u>
	<u>OnResetDevice</u>

The **ID3DXEffect** interface, like all COM interfaces, inherits from the **IUnknown Interface**.

The **LPD3DXEFFECT** type is defined as a pointer to the **ID3DXEffect** interface.

typedef struct ID3DXEffect *LPD3DXEFFECT;

Requirements

Header: Declared in D3dx8effect.h. **Import Library:** Use D3dx8.lib.

See Also

<u>D3DXCreateEffect</u>, <u>D3DXCreateEffectFromFile</u>, <u>D3DXCreateEffectFromResource</u>

D3DXSaveSurfaceToFile

Saves a surface to a file.

Parameters

pDestFile

[in] File name to save the surface to.

DestFormat

[in] **<u>D3DXIMAGE_FILEFORMAT</u>** specifying file format to use when saving.

pSrcSurface

[in] Pointer to **IDirect3DSurface8** interface, containing the image to be saved.

pSrcPalette

[in] Pointer to a **PALETTEENTRY** structure containing a palette of 256 colors. This parameter can be NULL.

pSrcRect

[in] Pointer to a **RECT** structure. Specifies the source rectangle. Set this parameter to NULL to specify the entire image.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be the following.

D3DERR_INVALIDCALL

Remarks

This function supports the following file formats: .bmp and .dds.

This function handles conversion to and from compressed texture formats.

This function supports both Unicode and ANSI strings.

Requirements

Header: Declared in D3dx8tex.h. **Import Library:** Use D3dx8.lib.

See Also

D3DXSaveTextureToFile, D3DXSaveVolumeToFile

D3DXFillTexture

Uses a user-provided function to fill each texel of each mip level of a given texture.

```
HRESULT D3DXFillTexture(
   LPDIRECT3DTEXTURE8 pTexture,
   LPD3DXFILL2D pFunction,
   LPVOID pData
);
```

Parameters

pTexture

[out, retval] Pointer to an **IDirect3DTexture8** interface, representing the filled texture.

pFunction

[in] Pointer to a **LPD3DXFILL2D** user-provided evaluator function, which will be used to compute the value of each texel.

pData

[in] Pointer to an arbitrary block of user-defined data. This pointer will be passed to the function provided in *pFunction*.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be the following values.

D3DERR_INVALIDCALL

Requirements

Header: Declared in D3dx8tex.h. **Import Library:** Use D3dx8.lib.

D3DXFillCubeTexture

Uses a user-provided function to fill each texel of each mip level of a given cube texture.

```
HRESULT D3DXFillCubeTexture(
   LPDIRECT3DCUBETEXTURE8 pTexture,
   LPD3DXFILL3D pFunction,
   LPVOID pData
);
```

Parameters

pTexture

[out, retval] Pointer to an **IDirect3DCubeTexture8** interface, representing the filled texture.

pFunction

[in] Pointer to a **LPD3DXFILL3D** user-provided evaluator function, which will be used to compute the value of each texel.

pData

[in] Pointer to an arbitrary block of user-defined data. This pointer will be passed to the function provided in *pFunction*.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

Requirements

Header: Declared in D3dx8tex.h. **Import Library:** Use D3dx8.lib.

D3DXFillVolumeTexture

Uses a user-provided function to fill each texel of each mip level of a given texture.

```
HRESULT D3DXFillVolumeTexture(
  LPDIRECT3DVOLUMETEXTURE8 pVolumeTexture,
  LPD3DXFILL3D
                           pFunction,
  LPVOID
                            pData
);
```

Parameters

pVolumeTexture

[out, retval] Pointer to an **IDirect3DVolumeTexture8** interface,

representing the filled texture.

pFunction

[in] Pointer to an LPD3DXFILL3D user-provided evaluator function, which will be used to compute the value of each texel.

pData

[in] Pointer to an arbitrary block of user-defined data. This pointer will be passed to the function provided in *pFunction*.

Return Values

If the function succeeds, the return value is D3D OK.

If the function fails, the return value can be the following value.

D3DERR INVALIDCALL

Remarks

If the volume is nondynamic (because of a usage parameter set to 0 at the creation) and located in video memory (the memory pool set to D3DPOOL DEFAULT), **D3DXFillVolumeTexture** will fail because Direct3DX cannot lock nondynamic volumes located in video memory.

Requirements

Header: Declared in D3dx8tex.h. **Import Library:** Use D3dx8.lib.

D3DXQuaternionSquadSetup

Setup control points for spherical quadrangle interpolation.

```
void D3DXQuaternionSquadSetup(
   D3DXQUATERNION* pAOut,
   D3DXQUATERNION* pBOut,
   D3DXQUATERNION* pCOut,
   CONST D3DXQUATERNION* pQ0,
   CONST D3DXQUATERNION* pQ1,
   CONST D3DXQUATERNION* pQ2,
   CONST D3DXQUATERNION* pQ3
);
```

Parameters

```
pAOut

[out] Pointer to AOut.

pBOut

[out] Pointer to BOut.

pCOut

[out] Pointer to COut.

pQ0

[in] Pointer to the input control point, Q0.

pQ1

[in] Pointer to the input control point, Q1.

pQ2

[in] Pointer to the input control point, Q2.

pQ3

[in] Pointer to the input control point, Q3.
```

Remarks

This function operates as shown below. It takes four control points (Q0, Q1, Q2, Q3), which are supplied to the inputs pQ0, pQ1, pQ2, and pQ3. The function then alters these values to find a curve that flows along the shortest path. The values of q0, q2, and q3 are calculated as shown below.

Having calculated the new Q values, the values for AOut, BOut, and COut are calculated as shown below.

```
AOut = q1 * e^{[-0.25 *(Ln[Exp(q1)*q2] + Ln[Exp(q1)*q0])}
BOut = q2 * e^{[-0.25 *(Ln[Exp(q2)*q3] + Ln[Exp(q2)*q1])}
Cout = q2
Note:
Ln is the API method <u>D3DXQuaternionLn</u>
Exp is the API method <u>D3DXQuaternionExp</u>
```

Example

The following example shows how to use a set of quaternion keys (Q0, Q1, Q2, Q3) to compute the inner quadrangle points (A, B, C). This ensures that the tangents are continuous across adjacent segments.

A B Q0 Q1 Q2 Q3

This is how you can interpolate between Q1 and Q2.

```
// rotation about the z axis
                                       0, 0.707f, -.707f);
D3DXQUATERNION Q0 = D3DXQUATERNION(0,
D3DXQUATERNION Q1 = D3DXQUATERNION(0,
                                       0, 0.000f, 1.000f);
D3DXQUATERNION Q2 = D3DXQUATERNION(0,
                                       0, 0.707f, 0.707f);
D3DXQUATERNION Q3 = D3DXQUATERNION(0,
                                       0, 1.000f, 0.000f);
D3DXQUATERNION A, B, C, Qt;
FLOAT time = 0.5f;
D3DXQuaternionSquadSetup(&A;, &B;, &C;, &Q0;, &Q1;, &Q2;, &Q3;);
D3DXQuaternionSquad(&Qt;, &Q1;, &A;, &B;, &C;, time);
Note:
C is +/- Q2 depending on the result of the function
Qt is the result of the function
The result is a rotation of 45 degrees around the z axis for time =
```

Requirements

Header: Declared in D3dx8math.h. **Import Library:** Use D3dx8.lib.

See Also

D3DXQuaternionSquad

D3DXQuaternionSquad

Interpolates between quaternions, using spherical quadrangle interpolation.

```
D3DXQUATERNION* D3DXQuaternionSquad(
   D3DXQUATERNION* pOut,
   CONST D3DXQUATERNION* pQ1,
   CONST D3DXQUATERNION* pA,
   CONST D3DXQUATERNION* pB,
   CONST D3DXQUATERNION* pC,
   FLOAT t
);
```

Parameters

pOut [in, out] Pointer to the D3DXQUATERNION structure that is the result of the operation.

 pQ1

 [in] Pointer to a source D3DXQUATERNION structure.

 pA

 [in] Pointer to a source D3DXQUATERNION structure.

 pB

 [in] Pointer to a source D3DXQUATERNION structure.

 pC

 [in] Pointer to a source D3DXQUATERNION structure.

 t

 [in] Pointer to a source D3DXQUATERNION structure.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the result of the spherical quadrangle interpolation.

Remarks

This function uses the following sequence of spherical linear interpolation

operations: Slerp(Slerp(pQ1, pC, t), Slerp(pA, pB, t), 2t(1 - t))

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionSquad** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h. **Import Library:** Use D3dx8.lib.

See Also

D3DXQuaternionExp, D3DXQuaternionLn, D3DXQuaternionSquadSetup

D3DXMatrixMultiplyTranspose

Determines the product of two matrices, followed by a transpose.

```
D3DXMATRIX* D3DXMatrixMultiplyTranspose(
   D3DXMATRIX* pOut,
   CONST D3DXMATRIX* pM1,
   CONST D3DXMATRIX* pM2
);
```

Parameters

pOut

[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

рМ1

[in] Pointer to a source **D3DXMATRIX** structure.

рМ2

[in] Pointer to a source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXMATRIX** structure that is the product of two matrices.

Remarks

The result represents the transformation M2, followed by the transformation M1, transposed by T (Out = T(M1 * M2)).

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixMultiplyTranspose** function can be used as a parameter for another function.

This function is useful to set matrices as constants for vertex and pixel shaders.

Requirements

Header: Declared in D3dx8math.h. **Import Library:** Use D3dx8.lib.

See Also

D3DXMatrixMultiply, D3DXQuaternionMultiply

D3DXFresnelTerm

Calculate the Fresnel term.

```
FLOAT D3DXFresnelTerm(
   FLOAT CosTheta,
   FLOAT RefractionIndex,
);
```

Parameters

CosTheta [in] The value must be between 0 and 1. RefractionIndex [in] the refraction index of a material. The value must be greater than 1.

Return Values

This function returns the Fresnel term for unpolarized light. CosTheta is the cosine of the incident angle.

Requirements

Header: Declared in D3dx8math.h. **Import Library:** Use D3dx8.lib.

Samples

DirectX Graphics C/C++ Samples

The following samples are built on a base class that includes Microsoft® Windows® and Microsoft® Direct3D® functionality. This base class provides many of the basic features in a Windows application, such as creating windows and handling messages. The samples include a derived class that overrides the methods necessary to add Direct3D features, such as bump maps, vertex blending, and volume textures. For more information about the sample architecture, see <u>Sample Framework</u>.

- Billboard Sample
- <u>BumpEarth Sample</u>
- BumpLens Sample
- <u>BumpUnderwater Sample</u>
- BumpWaves Sample
- Bump Self-Shadow Sample
- <u>ClipMirror Sample</u>
- <u>CubeMap Sample</u>
- <u>Cull Sample</u>
- DolphinVS Sample
- DotProduct3 Sample
- DXTex Tool
- Emboss Sample
- EnhancedMesh Sample
- FishEye Sample
- Lighting Sample
- <u>MFCFog Sample</u>
- <u>MFCPixelShader Sample</u>
- MFCTex Sample
- Moire Sample
- <u>OptimizedMesh Sample</u>
- <u>Pick Sample</u>
- <u>PointSprites Sample</u>
- <u>ProgressiveMesh Sample</u>

- <u>RTPatch Sample</u>
- <u>ShadowVolume Sample</u>
- <u>SkinnedMesh Sample</u>
- <u>SphereMap Sample</u>
- <u>StencilDepth Sample</u>
- <u>StencilMirror Sample</u>
- Text3D Sample
- <u>VertexBlend Sample</u>
- <u>VertexShader Sample</u>
- <u>VolumeFog Sample</u>
- <u>VolumeTexture Sample</u>
- <u>Water Sample</u>

The <u>Application Wizard</u> is available to help generate Microsoft® DirectX® applications.

Although DirectX samples include Microsoft® Visual C++® project workspace files, you might need to verify other settings in your development environment to ensure that the samples compile properly. For more information, see <u>Compiling DirectX Samples and Other DirectX Applications</u>.

See Also

DirectX Graphics C/C++ Tutorials

DirectX Graphics

This section provides information about using the Microsoft® DirectX® Graphics application programming interfaces (APIs).

As with other components of DirectX, DirectX Graphics can be used with C, C++, and Microsoft® Visual Basic®.

Roadmap

Discover the features of Microsoft® DirectX 8.1® in three ways.

• What's New in DirectX Graphics

This section highlights new features and functionality of this component in DirectX 8.1. If you have used Microsoft® Direct3D® or Microsoft® DirectDraw® before, read this section first because much has changed since DirectX 7.0.

<u>Programmers Guide</u>

This section contains architecture descriptions, functional block diagrams, descriptions of the building blocks in the pipeline, code snippets, and sample applications.

• <u>Reference</u>

This section contains the reference pages for the Direct3D application programming interface (API). This includes the syntax for the API methods, functions, instructions, and data structures. It includes an explanation of how the API method works, and often includes code snippets.

Reference

This section contains the reference pages for the Microsoft® Direct3D® application programming interface (API). Information is contained in the following sections.

- <u>Direct3D C/C++ Reference</u>
- <u>Direct3DX C/C++ Reference</u>
- <u>Vertex Shader Reference</u>
- <u>Pixel Shader Reference</u>
- Effect File Reference
- <u>X File C/C++ Reference</u>

Programmers Guide

This guide contains a description of the graphics pipeline implemented by Microsoft® Direct3D®. It is a guide for developers who are implementing threedimensional (3-D) graphics functionality into their applications. The guide contains architecture descriptions, functional block diagrams, and descriptions of the building blocks in the pipeline, as well as code snippets and sample applications. The information is divided into the following sections:

• <u>Getting Started with Direct3D</u>

This section contains both an overview of the pipeline and tutorials that can help you get a simple graphics application running in a few minutes.

• <u>Using Direct3D</u>

This section explains how to use the fixed function pipeline. Included here are the basic functional steps in the graphics pipeline: converting geometry, adding lighting, and rendering output.

• Programmable Pipeline

This section covers the new programmable extensions to the pipeline. Included here are details about using vertex shaders for manipulating object geometry, pixel shaders for controlling pixel shading, and effects and effect files for building applications that can run on a variety of hardware platforms.

<u>Advanced Topics</u>

This section contains examples of different types of special effects you can implement. Topics such as environment and bump mapping, antialiasing, vertex blending, and tweening show how to apply leading-edge special effects to your application.

• <u>Samples</u>

This section contains sample applications.

• <u>Direct3D Appendix</u>

This section contains details on additional topics, such as X Files and graphics state.

For more information about specific API methods, see the <u>Reference</u> pages.

Microsoft DirectX 8.1 (shader versions 1.0, 1.1)

Vertex Shaders

Previous to Microsoft® DirectX® 8.*n*, Microsoft® Direct3D® operated a fixed function pipeline for converting three-dimensional (3-D) geometry to rendered screen pixels. The user sets attributes of the pipeline that control how Direct3D transforms, lights, and renders pixels. The fixed function vertex format is declared at compile time and determines the input vertex format. Once defined, the user has little control over pipeline changes during runtime.

Programmable shaders add a new dimension to the graphics pipeline by allowing the transform, lighting, and rendering functionality to be modified at runtime. A shader is declared at runtime but, once done, the user is free to change which shader is active as well as to control the shader data dynamically using streaming data. This gives the user a new level of dynamic flexibility over the way that pixels are rendered.

A vertex shader file contains vertex shader instructions. Vertex shaders can control vertex color and how textures are applied to vertices. Lighting can be added through the use of vertex shader instructions. The shader instruction file contains ASCII text so it is readable and in some ways looks similar to assembly language. A vertex shader is invoked after any DrawPrimitive or DrawIndexedPrimitive call. Shaders can be dynamically switched using **SetVertexShader** to specify a new shader file, or by changing the instructions in the ASCII text shader file using the streaming data inputs. The <u>Vertex Shader</u> Assembler Reference has a complete listing of shader instructions.

For additional information, see the following sections.

• Create a Vertex Shader

This section contains a code sample that uses a vertex shader to apply a constant color to object vertices. This example contains a detailed explanation of the methods used.

• Shader2 - Apply vertex colors

Additional examples shows more code samples that add textures and blend vertex colors and textures.

• <u>Shader3 - Apply a texture map</u>

Additional examples shows more code samples that add textures and blend vertex colors and textures.

• <u>Shader4 - Apply a texture map with lighting</u>

Additional examples shows more code samples that add textures and blend vertex colors and textures.

• <u>Debugging</u>

Pixel Shaders

Before Microsoft® DirectX® 8.x, Microsoft® Direct3D® used a fixed function pipeline for converting three-dimensional (3-D) geometry to rendered screen pixels. The user sets attributes of the pipeline that control how Direct3D transforms, lights, and renders pixels. The fixed function vertex format is declared at compile time and determines the input vertex format. Once defined, the user has little control over pipeline changes during run time.

Shaders add a new dimension to the graphics pipeline by allowing the vertex transform, lighting, and individual pixel coloring functionality to be programmed. Pixel shaders are short programs that execute for each pixel when triangles are rasterized. This gives the user a new level of dynamic flexibility over the way that pixels are rendered.

A pixel shader contains pixel shader instructions made up of ASCII text. Arithmetic instructions can be used to apply diffuse and/or specular color. Texture addressing instructions provide a variety of operations for reading and applying texture data. Functionality is available for masking and swapping color components. The shader ASCII text looks similar to assembly language and is assembled using Direct3DX assembler functions from either a text string or a file. The assembler output is a series of opcodes that an application may provide to Direct3D by means of <u>IDirect3DDevice8::CreatePixelShader</u>. The <u>Pixel</u> <u>Shader Reference</u> has a complete listing of shader instructions.

To understand more about pixel shaders, see the following sections.

- <u>Create a Pixel Shader</u> contains a code sample that uses a pixel shader to apply Gouraud interpolated diffuse colors to an object. This example contains a detailed explanation of the methods used.
- <u>Texture Considerations</u> details the texture stage states that are ignored during pixel shaders.
- <u>Confirming Pixel Shader Support</u> gives a more detailed explanation of the structures for enumerating pixel shader support.
- <u>Pixel Shader Examples</u> shows more code samples that add textures and blend vertex colors and textures.
- <u>Converting Texture Operations</u> gives examples of converting texture operations to pixel shader instructions.

• <u>Debugging</u> provides debugging information.

Effects

Microsoft® Direct3D® provides a rich feature set for creating complex and visually realistic three-dimensional (3-D) scenes. Effect files help you write an application that uses all the rendering capabilities for the hardware on which it runs. Effects are a collection of different rendering techniques that can fit onto a variety of hardware devices.

For example, to create a realistic rippled pond of water that reflects light as shown in the following image, you begin with the first technique that renders the water, adds specular highlights, adds caustic textures, and applies light to the water in a single pass. If your hardware cannot render this technique in a single pass, a second technique might render the water, add specular highlights or caustic textures, but not apply light to the water.

Before you use a technique, you can validate it using Direct3D to see if it is supported by the current hardware configuration.

Effects are defined in an *effect file*. An effect file consists of one or more techniques. Each technique consists of one or more passes. These files are text based and can be changed without recompiling the source application. This enables you to program games that make optimum use of video card functionality. Effect files also make it easy to upgrade an existing game to run on newer video cards as additional features are developed.

The following topics discuss effects and how you can use them in your application.

- <u>Create an Effect</u>
- <u>Multiple Techniques</u>

Exercises*

*From the DirectX Meltdown 2001 conference, Programmable Shader workshop.

- <u>Exercise 1</u> Fixed Function Diffuse Lighting and Vertex Shader Diffuse Lighting.
- <u>Exercise 2</u> Vertex Shader Diffuse Lighting. Light the model, taking both diffuse material and a diffuse light source into consideration.
- <u>Exercise 3</u> Transforms. Transform the vertex normal into world space to take light source movement into consideration.
- Exercise 4 Texturing. Set up texture to pass onto FF PS Modulate between the texture and diffuse color arguments.
- <u>Exercise 5</u> Vertex Shader Specular Lighting.
- <u>Exercise 6</u> Standard Texture Effect.
- <u>Exercise 7</u> Multi-Texturing with Shaders.
- <u>Exercise 8</u> Texturing with Lights.
- <u>Exercise 9</u> Dot 3 Bump Mapping, Dot 3 Specular Bump Mapping, and Table Lookup Specular Bump Mapping.
- <u>Exercise 10</u> Anisotropic Bump Mapping.
- <u>Exercise 11</u> Area Lighting, Area and Diffuse Lighting.

For more information about effect files, see Effect File Format.

Mathematics of Lighting

The Microsoft® Direct3D® Light Model covers ambient, diffuse, specular, and emissive lighting. This is enough flexibility to solve a wide range of lighting situations. You refer to the total amount of light in a scene as the *global illumination* and compute it using the following equation.

Global Illumination = Ambient Light + Diffuse Light + Specular Light

Ambient lighting is constant lighting. It is constant in all directions and it colors all pixels of an object the same. It is fast to calculate but leaves objects looking flat and unrealistic. To see how ambient lighting is calculated by Direct3D, see <u>Ambient Lighting</u>.

Diffuse lighting depends on both the light direction and the object surface normal. It varies across the surface of an object as a result of the changing light direction and the changing surface numeral vector. It takes longer to calculate diffuse lighting because it changes for each object vertex, however the benefit of using it is that it shades objects and gives them three-dimensional (3-D) depth. To see how diffuse lighting is calculated in Direct3D, see <u>Diffuse Lighting</u>.

Specular lighting identifies the bright specular highlights that occur when light hits an object surface and reflects back toward the camera. It is more intense than diffuse light and falls off more rapidly across the object surface. It takes longer to calculate specular lighting than diffuse lighting, however the benefit of using it is that it adds significant detail to a surface. To see how specular lighting is calculated in Direct3D, see <u>Specular Lighting</u>.

Emissive lighting is light that is emitted by an object, for example, a glow. To see how emissive lighting is calculated in Direct3D, see <u>Emissive Lighting</u>.

Realistic lighting can be accomplished by applying each of these types of lighting to a 3-D scene. To achieve a more realistic lighting effect, you add more lights; however, the scene takes longer to render. To achieve all the effects a designer wants, some games use more CPU power than is commonly available. In this case, it is typical to reduce the number of lighting calculations to a minimum by using lighting maps and environment maps to add lighting to a scene while using texture maps.

All lighting computations are made in model space by transforming the light source's position and direction, along with the camera position, to model space using the inverse of the world matrix. As a result, if the world or view matrices introduce non-uniform scaling, the resultant lighting might be inaccurate. To see how lighting transformations are calculated, see <u>Camera Space Transformations</u>.

Diffuse and specular light values can be affected by a given light's attenuation and spotlight characteristics. Terms for both of these are included in the diffuse and specular equations. For more information, see <u>Attenuation and Spotlight</u> <u>Terms</u>.

Action Mapping

Traditionally, applications have done their own mapping of events to particular buttons and axes. A car-racing game, for example, might assume that the x-axis on the user's joystick or mouse was the most suitable control for steering the car. The only way to accommodate new or unusual devices was to provide configuration options so that the user could specify some other axis, such as a rotational axis, to use for steering. Moreover, the application had no way of knowing which installed device was the best fit for the game, so the user typically had to choose a device from a menu or make sure only the preferred device was attached.

Using action mapping, you no longer need to make assumptions about the best use of devices and device objects. Instead, your application binds actions to virtual controls wherever possible. Rather than getting data from the x-axis and steering the car to the left or the right accordingly, the application might get data from a virtual control called DIAXIS_DRIVINGR_STEER. Microsoft® DirectInput® assigns the virtual control to a physical control—that is, a device object. It does so by taking into account the application genre, user preferences, information from the device manufacturer, and the user's configuration of the device.

Action mapping also simplifies the input loop by returning data for all devices in a form independent of the particular device. A single action can be mapped to more than one device, and the input loop can respond to the action the same way regardless of which device is being read.

The following topics contain more information on the steps required to implement action mapping.

- Preparing the Action Map
- Finding Matching Devices
- Configuring the Action Map
- <u>User Configuration of the Device</u>
- Retrieving Action Data
- <u>Maintaining Files During Development</u>

IDirectInput8

Applications use the methods of the **IDirectInput8** interface to enumerate, create, and retrieve the status of Microsoft® DirectInput® devices, initialize the DirectInput object, and invoke an instance of the Microsoft Windows® Control Panel.

IDirectInput8 supersedes the **IDirectInput**, **IDirectInput2**, and **IDirectInput7** interfaces used in earlier versions of Microsoft DirectX[®].

IDirectInput8 is an interface to a new class of object, represented by the class identifier CLSID_DirectInput8, and cannot be obtained by calling QueryInterface on an interface to objects of class CLSID_DirectInput. Instead, obtain the **IDirectInput8** interface by using the **DirectInput8Create** function.

The methods of the **IDirectInput8** interface can be organized into the following groups.

Device Management ConfigureDevices

	<u>CreateDevice</u>
	EnumDevices
	EnumDevicesBySemantics
	<u>FindDevice</u>
	<u>GetDeviceStatus</u>
Miscellaneous	<u>Initialize</u>
	<u>RunControlPanel</u>

The **IDirectInput** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown <u>AddRef</u> <u>QueryInterface</u> <u>Release</u> The **LPDIRECTINPUT8** type is defined as a pointer to the **IDirectInput8** interface:

typedef struct IDirectInput8 *LPDIRECTINPUT8;

Requirements

Windows NT/2000/XP: Requires Windows® 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

DirectInput8Create

Creates a DirectInput® object and returns an **<u>IDirectInput8</u>** or later interface.

```
HRESULT WINAPI DirectInput8Create(
  HINSTANCE hinst,
  DWORD dwVersion,
  REFIID riidltf,
  LPVOID* ppvOut,
  LPUNKNOWN punkOuter
);
```

Parameters

hinst

Instance handle to the application or DLL that is creating the DirectInput object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that might be necessary for backward compatibility.

It is an error for a DLL to pass the handle to the parent application. For example, an ActiveX® control embedded in a Web page that uses DirectInput must pass its own instance handle, and not the handle to the Web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that might be necessary.

dwVersion

Version number of DirectInput for which the application is designed. This value is normally DIRECTINPUT_VERSION. If the application defines DIRECTINPUT_VERSION before including Dinput.h, the value must be greater than 0x0700. For earlier versions, use **DirectInputCreateEx**, which is in Dinput.lib.

riidltf

Unique identifier of the desired interface. For DirectX 8.0, this value is IID_IDirectInput8A or IID_IDirectInput8W. Passing the IID_IDirectInput8 define selects the ANSI or Unicode version of the interface, depending on whether UNICODE is defined during compilation.

ppvOut

Address of a pointer to a variable to receive the interface pointer if the call succeeds.

punkOuter

Pointer to the address of the controlling object's **IUnknown** interface for COM aggregation, or NULL if the interface is not aggregated. Most callers pass NULL. If aggregation is requested, the object returned in **ppvOut* is a pointer to **IUnknown**, as required by COM aggregation.

Return Values

If the function succeeds, the return value is DI_OK.

If the function fails, the return value can be one of the following error values:

DIERR BETADIRECTINPUTVERSION DIERR INVALIDPARAM DIERR OLDDIRECTINPUTVERSION DIERR OUTOFMEMORY

Remarks

The DirectInput object created by this function is implemented in Dinput8d.dll. Versions of interfaces earlier than DirectX 8.0 cannot be obtained in this implementation. To use earlier versions, create the DirectInput object by using **DirectInputCreate** or **DirectInputCreateEx**, which are in Dinput.lib.

Calling the function with *punkOuter* = NULL is equivalent to creating the object through **CoCreateInstance**(&*CLSID_DirectInput8, punkOuter, CLSCTX_INPROC_SERVER, &IID_IDirectInput8W, lplpDirectInput),* then initializing it with <u>IDirectInput8::Initialize</u>.

Calling the function with *punkOuter* != NULL is equivalent to creating the object through **CoCreateInstance**(&*CLSID_DirectInput8, punkOuter, CLSCTX_INPROC_SERVER, &IID_IUnknown, lplpDirectInput*). The aggregated object must be initialized manually.

Requirements

Windows NT/2000/XP: Requires Windows® 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in dinput.h.

Import Library: Use dinput8.lib.

DIJOYSTATE

Describes the state of a joystick device. This structure is used with the **IDirectInputDevice8::GetDeviceState** method.

```
typedef struct DIJOYSTATE {
    LONG
            1X;
    LONG
            1Y;
            1Z;
    LONG
    LONG
            lRx;
    LONG
            lRy;
    LONG
            lRz;
    LONG
            rglSlider[2];
    DWORD
            rgdwPOV[4];
    BYTE
            rgbButtons[32];
} DIJOYSTATE, *LPDIJOYSTATE;
```

Members

lX

X-axis, usually the left-right movement of a stick.

lY

Y-axis, usually the forward-backward movement of a stick.

lΖ

Z-axis, often the throttle control. If the joystick does not have this axis, the value is 0.

lRx

X-axis rotation. If the joystick does not have this axis, the value is 0.

lRy

Y-axis rotation. If the joystick does not have this axis, the value is 0.

lRz

Z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is 0.

rglSlider[2]

Two additional axes, formerly called the u-axis and v-axis, whose semantics depend on the joystick. Use the **IDirectInputDevice8::GetObjectInfo** method to obtain semantic information about these values.

rgdwPOV[4]

Direction controllers, such as point-of-view hats. The position is indicated in hundredths of a degree clockwise from north (away from the user). The center position is normally reported as -1; but see Remarks. For indicators that have only five positions, the value for a controller is -1, 0, 9,000, 18,000, or 27,000.

rgbButtons[32]

Array of buttons. The high-order bit of the byte is set if the corresponding button is down, and clear if the button is up or does not exist.

Remarks

You must prepare the device for joystick-style access by calling the **IDirectInputDevice8::SetDataFormat** method, passing the *c_dfDIJoystick* global data format variable.

If an axis is in relative mode, the appropriate member contains the change in position. If it is in absolute mode, the member contains the absolute axis position.

Some drivers report the centered position of the POV indicator as 65,535. Determine whether the indicator is centered as follows:

BOOL POVCentered = (LOWORD(dwPOV) == 0xFFFF);

Note Under Microsoft® DirectX® 7, sliders on some joysticks could be assigned to the Z axis, with subsequent code retrieving data from that member. Using DirectX 8, those same sliders will be assigned to the **rglSlider** array. This should be taken into account when porting applications to DirectX 8. Make any necessary alterations to ensure that slider data is retrieved from the rglSlider array.

Requirements

Windows NT/2000/XP: Requires Windows 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

See Also

DIJOYSTATE2

DIJOYSTATE2

Describes the state of a joystick device with extended capabilities. This structure is used with the **IDirectInputDevice8::GetDeviceState** method.

typedef struct DIJOYSTATE2 { LONG 1X; LONG 1Y; 1Z; LONG LONG lRx; LONG lRy; lRz; LONG rglSlider[2]; LONG DWORD rgdwPOV[4]; BYTE rgbButtons[128]; LONG 1VX;lVY; LONG lvz; LONG LONG lVRx; lVRy; LONG lVRz; LONG rglVSlider[2]; LONG LONG lAX; LONG lAY; LONG lAZ; LONG lARx; LONG lARy; LONG lARz; rglASlider[2]; LONG LONG lFX; lFY; LONG lFZ; LONG lFRx; LONG LONG lFRy; LONG lFRz; rglFSlider[2]; LONG } DIJOYSTATE2, *LPDIJOYSTATE2;

Members

lX

X-axis, usually the left-right movement of a stick.

lY

Y-axis, usually the forward-backward movement of a stick.

lΖ

Z-axis, often the throttle control. If the joystick does not have this axis, the value is 0.

lRx

X-axis rotation. If the joystick does not have this axis, the value is 0.

lRy

Y-axis rotation. If the joystick does not have this axis, the value is 0.

lRz

Z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is 0.

rglSlider[2]

Two additional axis values (formerly called the u-axis and v-axis) whose semantics depend on the joystick. Use the

IDirectInputDevice8::GetObjectInfo method to obtain semantic information about these values.

rgdwPOV[4]

Direction controllers, such as point-of-view hats. The position is indicated in hundredths of a degree clockwise from north (away from the user). The center position is normally reported as -1; but see Remarks. For indicators that have only five positions, the value for a controller is -1, 0, 9,000, 18,000, or 27,000.

rgbButtons[128]

Array of buttons. The high-order bit of the byte is set if the corresponding button is down, and clear if the button is up or does not exist.

lVX

X-axis velocity.

lVY

Y-axis velocity.

lVZ

Z-axis velocity.

lVRx

X-axis angular velocity.

lVRy

Y-axis angular velocity.

lVRz

Z-axis angular velocity.

rglVSlider[2]

Extra axis velocities.

lAX

X-axis acceleration.

lAY

Y-axis acceleration.

lAZ

Z-axis acceleration.

lARx

X-axis angular acceleration.

lARy

Y-axis angular acceleration.

lARz

Z-axis angular acceleration.

rglASlider[2]

Extra axis accelerations.

lFX

X-axis force.

lFY

Y-axis force.

lFZ

Z-axis force.

lFRx

X-axis torque.

lFRy

Y-axis torque.

lFRz

Z-axis torque.

rglFSlider[2]

Extra axis forces.

Remarks

You must prepare the device for access to a joystick with extended capabilities by calling the **IDirectInputDevice8::SetDataFormat** method, passing the *c_dfDIJoystick2* global data format variable.

If an axis is in relative mode, the appropriate member contains the change in position. If it is in absolute mode, the member contains the absolute axis position.

Some drivers report the centered position of the POV indicator as 65,535. Determine whether the indicator is centered as follows:

```
BOOL POVCentered = (LOWORD(dwPOV) == 0xFFFF);
```

Note Under Microsoft® DirectX® 7, sliders on some joysticks could be assigned to the Z axis, with subsequent code retrieving data from that member. Using DirectX 8, those same sliders will be assigned to the **rglSlider** array. This should be taken into account when porting applications to DirectX 8. Make any necessary alterations to ensure that slider data is retrieved from the rglSlider array.

Requirements

Windows NT/2000/XP: Requires Windows 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

See Also

DIJOYSTATE

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

ps

Provides the method to specify the version of the shader code.

ps.mainVer.subVer

Registers

Argument	Description	Values
mainVer	main version number	r 1
subVer	sub version number	0, 1, 2, 3, 4

Remarks

This instruction must be the first instruction in a shader.

Example

```
// This example declares a version 1.0 shader. \ensuremath{\text{ps.1.0}}
```

```
// This example declares a version 1.4 shader.
ps.1.4
```

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

def

Provides a method to define constants to be used within the pixel shader.

def dest, fVal0, fVal1, fVal2, fVal3

Registers

Argument	Description	Registers	Version
Aigument	Description	v_n c_n t_n r_n	VEI SIOII
dest	Destination register	х	1.0, 1.1, 1.2, 1.3, 1.4
fVal0, fVal1, fVal2, fVal3	Source floating point value	N/A N/A N/A N/A	1.0, 1.1, 1.2, 1.3, 1.4

N/A Not applicable. The float values do not use registers.

Each of the float values in fVal0, fVal1, fVal2, fVal3 is between -1.0 and 1.0. This is *not* necessarily the number specified in **MAXPIXELSHADERVALUE**.

To learn more about registers, see <u>Registers</u>.

Remarks

def instructions must be placed after the version instruction and before any arithmetic or texture address instructions.

The **def** instruction is an alternative to setting pixel shader constants by calling <u>SetPixelShaderConstant</u>. When SetPixelShader is called, the def instruction is effectively translated into a *SetPixelShaderConstant call*. Constant registers that are initialized by the **def** instruction during <u>SetPixelShader</u> can be overwritten by calling SetPixelShaderConstant manually.

This instruction does not count against the instruction limit. It is stripped from the instruction stream prior to being sent to the driver.

For more information about **MAXPIXELSHADERVALUE**, see **D3DCAPS8**.

Example

// This example outputs a constant color.

// The shader is shown below.
ps.1.0 // version instruction
def c0, 1.0f, 0.0f, 0.0f, 1.0f // set c0 register
mov r0, c0 // output constant color

Microsoft DirectX 8.1 (pixel shader version 1.4)

phase

The phase instruction marks the transition between phase 1 and phase 2. If no phase instruction is present, the entire shader if executed as if it is a phase 2 shader.

phase

Remarks

This instruction applies to version 1.4 only.

Shader instructions that occur before the phase instruction are phase 1 instructions. All other instructions are phase 2 instructions. By having two phases for instructions, the maximum number of instructions per shader is increased.

The unfortunate side-effect of the phase transition is that the alpha component of **temporary registers** are unset or uninitialized during the transition.

Example

This example shows how to group instructions as phase 1 or phase 2 instructions within a shader.

The phase instruction is also commonly called the phase marker because it marks the transition between phase 1 and 2 instructions. In a version 1.4 pixel shader, if the phase marker is not present, the shader is executed as if it is running in phase 2. This is important because there are differences between phase 1 and 2 instructions and register availability. The differences are noted throughout the reference section.

```
ps.1.4
   // Add phase 1 instructions here.
phase
   // Add phase 2 instructions here.
```

add

Performs a component-wise add of two registers.

add dest, src0, src1

Registers

Argumont	Description	R	egi	ste	ers	Version
Argument	Description	vn	c _n	t _n	r _n	VEISIOII
					Х	1.0
dest	Destination register			х	Х	1.1, 1.2, 1.3
					Х	1.4
		х	Х	х	Х	1.0, 1.1, 1.2, 1.3
src0, src1	Source register		X		Х	1.4 <u>phase</u> 1
		Х	Х		Х	1.4 phase 2

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction performs a component-wise addition of two registers as shown below.

dest.r = src0.r + src1.r dest.g = src0.g + src1.g dest.b = src0.b + src1.b dest.a = src0.a + src1.a

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample</u> <u>Framework</u>. The sample framework is the foundation on which many of the samples are built.

// This example adds the vertex color to the texture color.

// The input texture is shown on the left. The rendered output from
// pixel shader is shown on the right. In this example, it is bright
// because the texture color values have been doubled.

// Additional code loads the texture in texture stage 0.
LPDIRECT3DDEVICE8 m_pd3dDevice; // Init this device pointer in t
LPDIRECT3DTEXTURE8 m_pTexture0; // Use this variable to hold a p
TCHAR strPath[512] = "textureFile.jpg";

// Use a helper function from the SDK to load the texture. D3DUtil_CreateTexture(m_pd3dDevice, strPath, &m;_pTexture0, D3DFMT_ m_pd3dDevice->SetTexture(0, m_pTexture0); Microsoft DirectX 8.1 (pixel shader version 1.4)

bem

Apply a fake bump environment-map transform.

bem dest.rg, src0, src1

Registers

Argument	Description	Registers	Version
Aiguinein	Description	$\mathbf{v}_n \mathbf{c}_n \mathbf{t}_n \mathbf{r}_n$	
dest	Destination register	Х	1.4 <u>phase</u> 1
src0	Source register	X X	1.4 phase 1
src1	Source register	Х	1.4 phase 1

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction performs the following calculation.

Rules for using bem:

- 1. bem must appear in the first phase of a shader (that is, before a phase marker).
- 2. bem consumes two arithmetic instruction slots.
- 3. Only one use of this instruction is allowed per shader.
- 4. Destination writemask must be .rg /.xy.
- 5. This instruction cannot be co-issued.

6. Aside from the restriction that destination write mask be .rg, modifiers on source src0, src1, and instruction modifiers are unconstrained.

cmp

Conditionally chooses between src1 and src2, based on the comparison src0 $\geq = 0$.

cmp dest, src0, src1, src2

Registers

Argument	Description	R v _n	egi , C _n	ister t _n 1	1S In	Version
dest	Destination register			хх	ζ	1.2, 1.3
uesi	Destillation register			Х	Κ	1.4
		х	Х	ХУ	ζ	1.2, 1.3
src0, src1, src2	Source register		Х	Х	Κ	1.4 <u>phase</u> 1
		х	Х	У	Κ	1.4 phase 2

To learn more about registers, see <u>Registers</u>.

Remarks

The comparison is done per channel.

For pixel shader version 1.2 and 1.3, **cmp** counts as **two** arithmetic instructions. Unfortunately, this was discovered too late in the development cycle, and therefore is not validated properly when calling CreatePixelShader. It is incorrectly being counted as consuming only one arithmetic instruction. Be sure to manually count this instruction as two arithmetic instructions toward the maximum instruction count. For more information about instruction counts, see <u>Counting Instructions</u>.

In addition, for pixel shader version 1.2 and 1.3, the destination register for cmp cannot be the same as any of the source registers. Validation does not catch this, so be sure to keep this in mind.

Example

This example does a four-channel comparison.

```
// Compares all four components.
ps.1.4
def c0, -0.6, 0.6, 0, 0.6
def c1 0,0,0,0
def c2 1,1,1,1
cmp r0, c0, c1, c2 // r0 is assigned 1,0,0,0 based on the followin
// r0.x = c2.x because c0.x < 0
// r0.y = c1.y because c0.y >= 0
// r0.z = c1.z because c0.z >= 0
// r0.w = c1.w because c0.w >= 0
```

cnd

Conditionally chooses between src1 and src2, based on the comparison src0 > 0.5.

cnd dest, src0, src1, src2

Registers

Argument	Description	F	Reg	ist	ers	Version
¹ in Sument	Description	$v_n c_n t_n r_n$				V CI SION
					Х	1.0
dest	Destination register			х	Х	1.1, 1.2, 1.3
					Х	1.4
src0					r0.a	1.0, 1.1, 1.2, 1.3
src1, src2	Source register	Х	Х	Х	Х	1.0, 1.1, 1.2, 1.3
src0, src1, src2	8		Х		Х	1.4 <u>phase</u> 1
5100, 5101, 5102		х	х		Х	1.4 phase 2

To learn more about registers, see <u>Registers</u>.

Remarks

For versions 1.0 to 1.3, src0 must be r0.a. Version 1.4 has no such restriction.

```
// Version 1.1 to 1.3
if (r0.a > 0.5)
  dest = src1
else
  dest = src2
// Version 1.4 compares each channel separately.
for each component in src0
{
    if (src0.component > 0.5)
        dest.component = src1.component
    else
```

```
dest.component = src2.component
}
```

Example

These examples show a four-channel comparison done in a version 1.4 shader, as well as a single-channel comparison possible in a version 1.1 shader.

```
// Version 1.4 compares all four components.
ps.1.4
def c0, -0.5, 0.5, 0, 0.6
def c1 0,0,0,0
def c2 1,1,1,1
cnd r1, c0, c1, c2 // r0 contains 1,1,1,0 because,
// r1.x = c2.x because c0.x \le 0.5
// r1.y = c2.y because c0.y \le 0.5
// r1.z = c2.z because c0.z \leq 0.5
// r1.w = c1.w because c0.w > 0.5
// Version 1.1 to 1.3 compares against the replicated alpha channel
// of r0 only.
ps.1.1
def c0, -0.5, 0.5, 0, 0.6
def c1 0,0,0,0
def c2 1,1,1,1
mov r0, c0
cnd r1, r0.a, c1, c2 // r1 gets assigned 0,0,0,0 because
// r0.a > 0.5, therefore r1.xyzw = c1.xyzw
// This example compares two values, A and B, to each other.
// This example assumes A is loaded into v0 and B is loaded into v1.
// Both A and B must be in the range of -1 to +1, and since the
// color registers (v_n) are defined to be between 0 and 1,
// the restriction happens to be satisfied in this example.
// The shader is shown below.
                      // version instruction
ps.1.0
sub r0, v0, v1_bias // r0 = A - (B - 0.5)
cnd r0, r0.a, c0, c1 // r0 = ( A > B ? c0 : c1 )
// The result in r0 is c0 if A > B. Otherwise, the result in r0 is c
```

dp3

Calculates a three-component dot product. The scalar result is replicated to all four channels.

dp3 dest, src0, src1

Registers

Argumont	Description	R	egi	iste	ers	Version
Argument		vn	C _n	t _n	r _n	version
					Х	1.0
dest	Destination register			Х	Х	1.1, 1.2, 1.3
					Х	1.4
		Х	Х	Х	Х	1.0, 1.1, 1.2, 1.3
src0, src1	Source register		Х		Х	1.4 <u>phase</u> 1
		Х	Х		Х	1.4 phase 2

dp3 does not automatically clamp the output result between zero and one. If clamping is necessary, use the <u>saturate modifiers</u>.

dp3 can be co-issued as long as dp3 is writing the color channels and the other instruction is writing the alpha channel.

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction executes in the vector pipeline, always writing out to the color channels. For version 1.4, this instruction still uses the vector pipeline but may write to any channel.

dp3 r0.rgb, t0, v0 // Copy scalar result to RGB component

An instruction with a destination register RGB write mask may be co-issued

with dp3 as shown below.

dp3 r0.rgb, t0, v0 +mov r2.a, t0 // Copy scalar result to color compone
// Copy alpha component from t0 in par

The **dp3** instruction can be modified using the <u>Signed Scaling</u> input argument modifier (**_bx2**) applied to its input arguments if they are not already expanded to signed dynamic range. For a lighting shader, the saturate instruction modifier (**_sat**) is often used to clamp the negative values to black, as shown in the following example.

dp3_sat r0, t0_bx2, v0_bx2 // Here t0 is a bump map, v0 contains

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample</u> <u>Framework</u>. The sample framework is the foundation on which many of the samples are built.

This example uses a dot product to square the vertex diffuse color c

// The shader is s	hown below.
ps.1.0 //	Version instruction.
tex t0 //	Declare texture.
dp3 r0, v0, v0 //	Dot product squares the vertex color values,
11	color(v0) * color(v0).
11	Bright colors max out at white.
11	Dimmer colors yield gray.

The results of this example are shown below. The input vertex colors are shown on the left. The rendered output from the pixel shader is shown on the right. The left, bottom, and right edges are white because the input color components reach the maximum color value when squared. The center color is gray because the squared color values are lower where the input colors blend together.

Additional code loa	ds a texture in	texture stage 0
LPDIRECT3DDEVICE8	m_pd3dDevice;	<pre>// Initialize the pointer befo</pre>
LPDIRECT3DTEXTURE8	<pre>m_pTexture0;</pre>	<pre>// Use this variable to hold a</pre>
TCHAR	strPath[512] =	"textureFile.jpg";

D3DUtil_CreateTexture(m_pd3dDevice, strPath, &m;_pTexture0, D3DFMT_ m_pd3dDevice->SetTexture(0, m_pTexture0);

dp4

Calculates a four-component dot product. The scalar result is replicated to all channels.

dp4 dest, src0, src1

Registers

Argument	Description			isters t _n r _n	Version
1.					1.2, 1.3
dest	Destination register			х	1.4
		Х	х	X X	1.2, 1.3
src0, src1	Source register		Х	Х	1.4 <u>phase</u> 1
		Х	Х	Х	1.4 phase 2

This instruction cannot be co-issued.

This instruction does not automatically clamp the output result between zero and one. If clamping is necessary, use the <u>saturate modifier</u>.

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction executes in both the vector and alpha pipeline.

dp4 r0, t0, v0 // Copy scalar result to RGBA components.

For pixel shader version 1.2 and 1.3, **dp4** counts as **two** arithmetic instructions. Unfortunately, this was discovered too late in the development cycle and therefore is not validated properly when calling CreatePixelShader. It is being incorrectly counted as consuming only one arithmetic instruction. Be sure to manually count this instruction as two arithmetic instructions toward the

maximum instruction count. For more information about instruction counts, see <u>Counting Instructions</u>.

In addition, for pixel shader version 1.2 and 1.3, the destination register for dp4 cannot be the same as any of the source registers. Validation does not catch this, so be sure to keep this in mind.

lrp

Interpolates linearly between the second and third source registers by a proportion specified in the first source register.

lrp dest, src0, src1, src2

Registers

Argumont	Description	Registers				Version
Argument		vn	c _n	t _n	r _n	VEISIOII
					х	1.0
dest	Destination register			Х	Х	1.1, 1.2, 1.3
					Х	1.4
		Х	Х	Х	Х	1.0, 1.1, 1.2, 1.3
src0, src1, src2	Source register		Х		Х	1.4 <u>phase</u> 1
		х	Х		Х	1.4 phase 2

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction performs the linear interpolation based on the following formula.

dest = src0 * src1 + (1-src0) * src2
// which is the same as
dest = src2 + src0 * (src1 - src2)

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample Framework</u>. The sample framework is the foundation on which many of the samples are built.

// This example combines a texture color with a diffuse color value.

// The shader is shown below.
ps.1.0 // Version instruction.
tex t0 // Declare texture.
lrp r0, t0, v0, t0 // Blend from v0 to t0 by t0 amount.

// The input colors and the output colors are shown below. The first // (src0) determines the amount of the second image (src1) and the t // (src2) that are blended to make the final image (dest). Where the // is white, the second image appears in the output. Where the first // black, the third images appears in the output. Where the first is // final image contains color values from the second and third image



// Additional code loads the texture in texture stage 0.
LPDIRECT3DDEVICE8 m_pd3dDevice; // initialize the pointer
LPDIRECT3DTEXTURE8 m_pTexture0; // a pointer to the texture.
TCHAR strPath[512] = "textureFile.jpg";

D3DUtil_CreateTexture(m_pd3dDevice, strPath, &m;_pTexture0, D3DFMT_ m_pd3dDevice->SetTexture(0, m_pTexture0);

mad

Multiply and add instruction. Sets the destination register to (src0 * src1) + src2. mad dest, src0, src1, src2

Registers

Argumont	Registe		0		Version	
Argument	Description	vn	c _n	t _n	r _n	VEISIOII
					Х	1.0
dest	Destination register			Х	Х	1.1, 1.2, 1.3
					Х	1.4
		Х	Х	Х	Х	1.0, 1.1, 1.2, 1.3
src0, src1, src2	Source register		Х		Х	1.4 <u>phase</u> 1
		Х	Х		Х	1.4 phase 2

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction performs a multiply accumulate operation based on the following formula.

dest = src0 * src1 + src2

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample Framework</u>. The sample framework is the foundation on which many of the samples are built.

// This example blends a diffuse color, a texture color and a consta

// The shader is shown below.
ps.1.0 // Version instruction.
tex t0 // Declare texture.
mad r0, v0, t0, v0 // Mix diffuse color and texture color.

// The following four images show the contents of the three source r // and the resulting output register. The output register shows the // the gradient in the center of the destination image, where the mi // appear. This is a result of adding the pixels from the center of // the pixel color created from the center of the product of src0*sr



// Additional code loads a texture in texture stage 0.
LPDIRECT3DDEVICE8 m_pd3dDevice; // initialize this pointer
LPDIRECT3DTEXTURE8 m_pTexture0; // a pointer to the texture.
TCHAR strPath[512] = "textureFile.jpg";

D3DUtil_CreateTexture(m_pd3dDevice, strPath, &m;_pTexture0, D3DFMT_ m_pd3dDevice->SetTexture(0, m_pTexture0);

mov

Copies the contents of the source to the destination.

mov dest, src

Registers

Argumont	Description	R	egi	ste	ers	Version
Argument	Description	vn	c _n	t _n	r _n	VEISIOII
					Х	1.0
dest	Destination register			х	х	1.1, 1.2, 1.3
					х	1.4
		Х	Х	х	х	1.0, 1.1, 1.2, 1.3
SrC	Source register		X		Х	1.4 <u>phase</u> 1
		Х	Х		Х	1.4 phase 2

To learn more about registers, see <u>Registers</u>.

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample</u> <u>Framework</u>. The sample framework is the foundation on which many of the samples are built.

// This example copies the texture color to the output. // The shader is shown below. ps.1.0 // Version instruction tex t0 // Declare texture. mov r0, t0 // Move texture to output. // The following images show the contents of the source register and

// The following images show the contents of the source register and // the resulting destination register. The images are identical beca // the mov instruction was used.

- I I	
	_

// Additional code loads the texture in texture stage 0.
LPDIRECT3DDEVICE8 m_pd3dDevice; // Init this device pointer in t
LPDIRECT3DTEXTURE8 m_pTexture0; // Use this variable to hold a p
TCHAR strPath[512] = "textureFile.jpg";

D3DUtil_CreateTexture(m_pd3dDevice, strPath, &m;_pTexture0, D3DFMT_ m_pd3dDevice->SetTexture(0, m_pTexture0);

mul

Multiplies the components of two source registers. The result is dest = src0 * src1.

mul dest, src0, src1

Registers

Argument	t Description	Registers			ers	Version
Aiguillein		vn	c n	t _n	r _n	V CI SIOII
dest	Destination register				Х	1.0
				Х	Х	1.1, 1.2, 1.3
					Х	1.4
src0, src1	Source register	Х	Х	Х	Х	1.0, 1.1, 1.2, 1.3
			Х		Х	1.4 <u>phase</u> 1
		Х	Х		Х	1.4 phase 2

To learn more about registers, see <u>Registers</u>.

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample</u> <u>Framework</u>. The sample framework is the foundation on which many of the samples are built.

// This example combines the texture color and the diffuse color.
// The shader is shown below.
ps.1.0 // Version instruction
tex t0 // Declare texture.
mul r0, v0, t0 // Multiply diffuse color with gradient texture.

// The following images show the contents of the source registers th
// resulting output register.

// Where src1 is white, the destination pixel color is the same as t
// source pixel color since dest = src * 1.0.
// Where src1 is black, the destination is also black.
// The pixel colors in the middle of the destination image are a ble
// of src0 and src1.



// Additional code loads a texture in texture stage 0. LPDIRECT3DDEVICE8 m_pd3dDevice; // Init this device pointer in t LPDIRECT3DTEXTURE8 m_pTexture0; // Use this variable to hold a p TCHAR strPath[512] = "textureFile.jpg";

D3DUtil_CreateTexture(m_pd3dDevice, strPath, &m;_pTexture0, D3DFMT_ m_pd3dDevice->SetTexture(0, m_pTexture0);

nop

No operation is performed.

nop

Registers

None

Remarks

This instruction performs a no-op, or no operation. The syntax for calling it is as follows:

nop

sub

Performs subtraction. Subtracts the second source register from the first source register.

sub dest, src0, src1

Registers

Argumont	Description	Registers				Version
Argument		vn	c _n	t _n	r _n	version
dest	Destination register				Х	1.0
				Х	Х	1.1, 1.2, 1.3
					Х	1.4
src0, src1	Source register	Х	Х	Х	Х	1.0, 1.1, 1.2, 1.3
			Х		Х	1.4 <u>phase</u> 1
		Х	Х		Х	1.4 phase 2

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction performs the subtraction based on the following formula.

dest = src0 - src1

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample Framework</u>. The sample framework is the foundation on which many of the samples are built.

// This example subtracts texture color from the diffuse color.

// The shader is shown below. ps.1.0 // Version instruction tex t0 // Declare texture. sub r0, v0, t0 // Subtract texture color from diffuse color.

// The following images show the contents of the source registers an // the resulting destination register. The colors in the second imag // are subtracted from the color in the first image (src0) to make t // resulting image (dest).

// Additional code loads the texture in texture stage 0.
LPDIRECT3DDEVICE8 m_pd3dDevice; // Init this device pointer in t
LPDIRECT3DTEXTURE8 m_pTexture0; // Use this variable to hold a p
TCHAR strPath[512] = "textureFile.jpg";

D3DUtil_CreateTexture(m_pd3dDevice, strPath, &m;_pTexture0, D3DFMT_ m_pd3dDevice->SetTexture(0, m_pTexture0); Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3**)

tex

Loads the destination register with color data (RGBA) sampled from a texture.

The texture must be bound to a particular texture stage (n) using <u>SetTexture</u>. Texture sampling is controlled by the texture stage state attributes, set with <u>SetTextureStageState</u>.

tex dest

Registers

Argument	Description	Registers v _n c _n t _n r _n	Version
dest	Destination register	Х	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

The destination register number specifies the texture stage number.

Texture sampling uses texture coordinates to look up, or sample, a color value at the specified (u,v,w,q) coordinates while taking into account the texture stage state attributes.

The texture coordinate data is interpolated from the vertex texture coordinate data and is associated with a specific texture stage. The default association is a one-to-one mapping between texture stage number and texture coordinate declaration order. This means that the first set of texture coordinates defined in the vertex format are by default associated with texture stage 0.

Texture coordinates may be associated with any stage using two techniques. When using a fixed function vertex shader or the fixed function pipeline, the texture stage state flag TSS_TEXCOORDINDEX can be used in <u>SetTextureStageState</u> to associate coordinates to a stage. Otherwise, the texture coordinates are output by the vertex shader oT_n registers when using a programmable vertex shader.

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample Framework</u>. The sample framework is the foundation on which many of the samples are built.

// This example applies a texture to a quad. // The shader is shown below. // version instruction ps.1.0 tex t0 // samples the texture at stage 0 using texture coordi mov r0, t0 // copies the color in t0 to output register r0 // The rendered output from the pixel shader is shown below. It is // simply a texture map applied to a quad object. // Additional code is required to use this shader and an example // scenario is shown below. // Load the texture in texture stage 0. m_pd3dDevice; // Initialize the pointer be m_pTexture0: // a pointer for the texture LPDIRECT3DDEVICE8 LPDIRECT3DTEXTURE8 m_pTexture0; // a pointer for the texture strPath[512] = "DX5_Logo.bmp"; TCHAR // Helper function from the SDK D3DUtil CreateTexture(m pd3dDevice, strPath, &m; pTexture0, D3DFMT m_pd3dDevice->SetTexture(0, m_pTexture0); // This code creates the shader from a file. The contents of the sha // file can also be supplied as a text string. strPShaderPath[512]; TCHAR pCode; LPD3DXBUFFER // Helper function from the SDK DXUtil_FindMediaFile(strPShaderPath, _T("shaderFile.txt")); // Assemble the vertex shader from the file. D3DXAssembleShaderFromFile(strPShaderPath, 0, NULL, &pCode;, NULL) m_pd3dDevice->CreatePixelShader((DWORD*)pCode->GetBufferPointer(), &m;_hPixelShader); pCode->Release();

```
// Define the object vertex data.
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    FLOAT tu1, tv1;
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_TEX1|TEXCOORD2(0))
static CUSTOMVERTEX g_Vertices[]=
{
    // X
                    Ζ
                           u1
                                 v1
              У
    { -1.0f, -1.0f, 0.0f, 0.0f, 1.0f, },
    { +1.0f, -1.0f, 0.0f, 1.0f, 1.0f, },
    { +1.0f, +1.0f, 0.0f, 1.0f, 0.0f, },
    { -1.0f, +1.0f, 0.0f, 0.0f, 0.0f, },
    // v1 is flipped to meet the top down convention in Windows
    // the upper left texture coordinate is (0,0)
    // the lower right texture coordinate is (1,1).
};
// Create and fill the guad vertex buffer.
m_pd3dDevice->CreateVertexBuffer( 4*sizeof(CUSTOMVERTEX),
                D3DUSAGE_WRITEONLY, D3DFVF_CUSTOMVERTEX,
                D3DPOOL_MANAGED, &m; pQuadVB );
CUSTOMVERTEX* pVertices = NULL;
m_pQuadVB->Lock( 0, 4*sizeof(CUSTOMVERTEX), (BYTE**)&pVertices;, 0 )
for( DWORD i=0; i<4; i++ )</pre>
    pVertices[i] = g_Vertices[i];
m_pQuadVB->Unlock();
// Check to see if the hardware supports pixel shaders.
if( D3DSHADER VERSION MAJOR( pCaps->PixelShaderVersion ) < 1 )
        return E_FAIL;
// Set up the transforms.
D3DXVECTOR3 from( 0, 0, -5.0f );
D3DXVECTOR3 at( 0.0f, 0.0f, 0.0f );
D3DXVECTOR3 up( 0.0f, 1.0f, 0.0f );
D3DXMATRIX matWorld;
D3DXMatrixIdentity( &matWorld; );
m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld; );
D3DXMATRIX matView;
D3DXMatrixLookAtLH( &matView;, &from;, &at;, &up; );
m pd3dDevice->SetTransform( D3DTS VIEW, &matView; );
```

D3DXMATRIX matProj; D3DXMatrixPerspectiveFovLH(&matProj;, D3DX_PI/4, 1.0f, 0.5f, 1000.0 m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj;); // Render the output. // Clear the back buffer to black. m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_TARGET, 0x00000000, 1.0f, 0L // Set device state. m_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE); m_pd3dDevice->SetRenderState(D3DRS_CLIPPING, FALSE); m_pd3dDevice->SetRenderState(D3DRS_LIGHTING, FALSE); m_pd3dDevice->SetRenderState(D3DRS_ZENABLE, FALSE); m_pd3dDevice->SetTexture(0, m_pTexture0); m_pd3dDevice->SetStreamSource(0, m_pQuadVB, sizeof(CUSTOMVERTEX)); m_pd3dDevice->SetVertexShader(D3DFVF_CUSTOMVERTEX); m_pd3dDevice->SetPixelShader(m_hPixelShader); m pd3dDevice->DrawPrimitive(D3DPT TRIANGLEFAN, 0, 2);

m_pd3dDevice->SetTexture(0, NULL);

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3**)

texbem

Apply a fake bump environment-map transform. This is accomplished by modifying the texture address data of the destination register, using address perturbation data (du,dv), and a two-dimensional (2-D) bump environment matrix.

texbem dest, src

Registers

Argument	Description	Registers	Version
Aiguinein	i Description	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	V CI SIOII
dest	Destination register	X	1.0, 1.1, 1.2, 1.3
STC	Source register	Х	1.0, 1.1, 1.2, 1.3

The red and green color data in the src register is interpreted as the perturbation data (du,dv).

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction transforms red and green components in the source register using the 2-D bump environment-mapping matrix. The result is added to the texture coordinate set corresponding to the destination register number, and is used to sample the current texture stage.

This operation always interprets du and dv as signed quantities. For versions 1.0 and 1.1, the <u>Signed Scaling</u> input modifier (**_bx2**) is not permitted on the input argument.

This instruction produces defined results when input textures contain signed format data. Mixed format data works only if the first two channels contain signed data. For more information about surface formats, see <u>D3DFORMAT</u>.

This can be used for a variety of techniques based on address perturbation, including fake per-pixel environment mapping and diffuse lighting (bump mapping).

```
// When using this instruction, texture registers must follow the fc
// The texture assigned to stage t(n) contains the (du,dv) data.
// The texture assigned to stage t(m) is sampled.
tex t(n)
texbem t(m), t(n) where m > n
// The calculations done within the instruction are shown below.
// 1. New values for texture addresses (u',v') are calculated.
// 2. Sample the texture using (u',v')
u' = TextureCoordinates(stage m)<sub>u</sub> + D3DTSS_BUMPENVMAT00(stage m)*t(n
D3DTSS_BUMPENVMAT10(stage m)*t(n)<sub>G</sub>
v' = TextureCoordinates(stage m)<sub>v</sub> + D3DTSS_BUMPENVMAT01(stage m)*t(n
D3DTSS_BUMPENVMAT11(stage m)*t(n)<sub>G</sub>
```

 $t(m)_{RGBA}$ = TextureSample(stage m) using (u', v') as coordinates.

Note When using **texbem** or **texbeml**, do not re-read the source register later in the shader because the data within the register might be corrupted. The shader validation allows this even though the result will be undefined.

Example

```
// Here is an example shader with the texture maps identified and
// the texture stages identified.
ps.1.0
tex t0 ; define t0 to get a 2-tuple DuDv
texbem t1, t0 ; compute (u',v')
; sample t1 using (u',v')
mov r0, t1 ; output result
// texbem requires the following textures in the following texture s
//
// Stage 0 is assigned a bump map with (du, dv) perturbation data.
//
// Stage 1 uses a texture map with color data.
//
// This instruction sets the matrix data on the texture stage that i
```

// This is different from the functionality of the fixed function pi
// the perturbation data and the matrices occupy the same texture st

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3**)

texbeml

Apply a fake bump environment-map transform with luminance correction. This is accomplished by modifying the texture address data of the destination register, using address perturbation data (du,dv), a two-dimensional (2-D) bump environment matrix, and luminance.

texbeml dest, src

Registers

Argumont	Description	Registers	Version
Argument	Description	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	VEISIOII
dest	Destination register	X	1.0, 1.1, 1.2, 1.3
STC	Source register	Х	1.0, 1.1, 1.2, 1.3

The red and green color data in the src register is interpreted as the perturbation data (du,dv).

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction transforms the red and green components in the source register using the 2-D bump environment mapping matrix. The result is added to the texture coordinate set corresponding to the destination register number. A luminance correction is applied using the luminance value and the bias texture stage values. The result is used to sample the current texture stage.

This can be used for a variety of techniques based on address perturbation such as fake per-pixel environment mapping.

This operation always interprets du and dv as signed quantities. For versions 1.0 and 1.1, the <u>Signed Scaling</u> input modifier (**_bx2**) is not permitted on the input argument.

This instruction produces defined results when input textures contain mixed format data. For more information about surface formats, see <u>D3DFORMAT</u>.

```
// When using this instruction, texture registers must follow the fo
// The texture assigned to stage tn contains the (du,dv) data.
// The texture assigned to stage t(m) is sampled.
tex
        t(n)
texbeml t(m), t(n) where m > n
// This example shows the calculations done within the instruction.
// 1. New values for texture addresses (u', v') are calculated.
// 2. Sample the texture using (u', v')
// 3. Luminance correction is applied.
u' = TextureCoordinates(stage m)<sub>u</sub> + D3DTSS_BUMPENVMAT00(stage m)*t(n
           D3DTSS_BUMPENVMAT10(stage m)*t(n)<sub>G</sub>
v' = TextureCoordinates(stage m)_v + D3DTSS_BUMPENVMAT01(stage m)*t(n)
          D3DTSS_BUMPENVMAT11(stage m)*t(n)<sub>G</sub>
t(m)_{RGBA} = TextureSample(stage m) using (u', v') as coordinates.
t(m)_{RGBA} = t(m)_{RGBA}*[t(n)_{B}*(D3DTSS_BUMPENVLSCALE(stage m) + D3DTSS_BUM
```

Note When using **texbem** or **texbeml**, do not re-read the source register later in the shader because the data within the register might be corrupted. The shader validation allows this even though the result will be undefined.

Example

```
// Here is an example shader with the texture maps identified and
// the texture stages identified.
ps.1.0
tex t0 ; define t0 to get a 2-tuple DuDv
texbeml t1, t0 ; compute (u',v')
; apply luminance correction
; sample t1 using (u',v')
mov r0, t1 ; output result
```

```
// This example requires the following textures in the following tex
//
// Stage 0 is assigned a bump map with (du, dv) perturbation data.
```

//
// Stage 1 is assigned a texture map with color data.
//
// texbeml sets the matrix data on the texture stage that is sampled

- // This is different from the functionality of the fixed function pi
- // the perturbation data and the matrices occupy the same texture st

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3**)

texcoord

Interprets texture coordinate data (UVW1) as color data (RGBA).

texcoord dest

Registers

Argument	t Description	Registers v _n c _n t _n r _n	Version
dest	Destination register	X	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction interprets the texture coordinate set (UVW1) corresponding to the destination register number as color data (RGBA). If the texture coordinate set contains fewer than three components, the missing components are set to 0. The fourth component is always set to 1. All values are clamped between 0 and 1.

The advantage of **texcoord** is that it provides a way to pass vertex data interpolated at high precision directly into the pixel shader. However, once the data is written into the destination register, some precision will be lost, depending on the number of bits used by the hardware for registers.

No texture is sampled by this instruction. Only texture coordinates set on this texture stage are relevant.

Any texture data (such as position, normal, and light source direction) can be mapped by a vertex shader into a texture coordinate. This is done by associating a texture with a texture register using <u>SetTexture</u> and by specifying how the texture sampling is done using <u>SetTextureStageState</u>. If the fixed function pipeline is used, be sure to supply the TSS_TEXCOORDINDEX flag.

// This instruction is used as follows:
texcoord tn

// A texture register (tn) contains four color values (RGBA). The da
// thought of as vector data (xyzw). Texcoord will retrieve 3 of the
// texture coordinate set x, and the fourth component (w) is set to
// The texture address is copied from the texture coordinate set n.
// The result is clamped between 0 and 1.

Example

This example is for illustration only. The C code accompanying the shader has not been optimized for performance. It can use helper functions from the <u>Sample</u> <u>Framework</u>. The sample framework is the foundation on which many of the samples are built.

// Here is an	example shader using texcoord.
ps.1.0	; version instruction
texcoord t0	; declare t0 hold texture coordinates,
	; which represent rgba values in this example
mov r0, t0	; move the color in tO to output register rO

The rendered output from the pixel shader is shown below. The (u,v,w,1) coordinate values map to the (rgb) channels. The alpha channel is set to 1. At the corners of the image, coordinate (0,0,0,1) is interpreted as black, (1,0,0,1) is red, (0,1,0,1) is green, and (1,1,0,1) contains green and red, producing yellow.

// Additional code is required to use this shader and an example // scenario is shown below. // This code creates the shader from a file. The contents of the sha // file can also be supplied as a text string. TCHAR strPShaderPath[512]; LPD3DXBUFFER pCode; DXUtil_FindMediaFile(strPShaderPath, _T("shaderFile.txt")); // Assemble the vertex shader from the file. D3DXAssembleShaderFromFile(strPShaderPath, 0, NULL, &pCode;, NULL) m_pd3dDevice->CreatePixelShader((DWORD*)pCode->GetBufferPointer(),

```
&m;_hPixelShader );
```

Microsoft DirectX 8.1 (pixel shader version 1.4)

texcrd

Copies texture coordinate data from the source texture coordinate iterator register as color data in the destination register.

texcrd dest, src

Registers

Argumont	Description	Registers	Version	
Argument	Description	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	VEISIUII	
dest	Destination register	X	1.4	
src	Source register	Х	1.4 <u>phase</u> 1	
510	Source register	Х	1.4 phase 2	

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction interprets coordinate data as color data (RGBA).

No texture is sampled by this instruction. Only texture coordinates set on this texture stage are relevant.

When using texcrd, keep in mind the following detail about how data is copied from the source register to the destination register. The source texture coordinate register (t#) holds data in the range [-D3DCAPS8.MaxTextureRepeat, D3DCAPS8.MaxTextureRepeat], while the destination register (r#) can hold data only in the (likely smaller) range [-D3DCAPS8.MaxPixelShaderValue, D3DCAPS8.MaxPixelShaderValue]. Note that for pixel shader version 1.4, D3DCAPS8.MaxPixelShaderValue]. Note that for pixel shader version 1.4, D3DCAPS8.MaxPixelShaderValue must be a minimum of eight. The texcrd instruction, in the process of clamping source data that is out of range of the destination register, is likely to behave differently on different hardware. The first pixel shader version 1.4 hardware on the market will perform a special clamp for values outside of range. This clamp is designed to produce a number that can fit into the destination register, but also to preserve texture addressing behavior for out-of-range data (see **D3DTEXTUREADDRESS**) if the data were to be subsequently used for texture sampling. However, new hardware from different manufacturers might not exhibit this behavior and might simply chop data to fit the destination register range. Therefore, the safest course of action when using pixel shader version 1.4 texcrd is to supply texture coordinate data only into the pixel shader that is already within the range [-8,8] so that you do not rely on the way hardware clamps.

Unlike <u>texcoord</u>, **texcrd** does not clamp values between 0 and 1.

Rules for using **texcrd**:

- 1. The same .xyz or .xyw modifier must be applied to every read of an individual t(n) register within a texcrd or texld instruction.
- 2. The fourth channel result of texcrd is unset/undefined in all cases.
- 3. The third channel is unset/undefined for the xyw_dw case.

Example

The complete set of allowed syntax for texcrd, taking into account all valid source modifier/selector and destination write mask combinations, is shown below. Note that the .rgba and .xyzw notation can be used interchangeably.

```
texcrd r(m).rgb, t(n).xyz
// Copies first three channels of texture coordinate iterator regist
// t(n), into r(m). The fourth channel of
// r(m) is uninitialized.
texcrd r(m).rgb, t(n)
// Produces the same result as the previous instruction.
texcrd r(m).rgb, t(n).xyw
// Puts first, second, and fourth components of t(n) into first thre
// of r(m). The fourth channel of r(m) is uninitialized.
// Here is a projective divide example using the _dw modifier.
texcrd r(m).rg, t(n)_dw.xyw
// This example copies x/w and y/w from t(n) into the
// first two channels of r(m). The third and fourth
// channels of r(m) are uninitialized. Any data previously
// written to the third channel of r(m) will be lost. Data
```

// in the fourth channel of r(m) is lost due to the phase
// marker. For version 1.4, the D3DTTFF_PROJECTED flag is ignored.

Microsoft DirectX 8.1 (pixel shader version 1.4)

texdepth

Calculate depth values to be used in the depth buffer comparison test for this pixel.

texdepth dest

Registers

Argument	Description	Registers v _n c _n t _n r _n	Version
dest	Destination register	r5 1	.4 <u>phase</u> 2 only

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction uses r5.r / r5.g in the depth buffer comparison test for this pixel. The data in the blue and alpha channels is ignored. If r5.g = 0, the result of r5.r / r5.g = 1.0.

Temporary register r5 is the only register that this instruction can use.

After executing this instruction, temporary register r5 is unavailable for additional use in the shader.

When multisampling, using this instruction eliminates most of the benefit of the higher resolution depth buffer. Because the pixel shader executes once per pixel, the single depth value output by <u>texm3x2depth</u> or **texdepth** will be used for each of the sub-pixel depth comparison tests.

Example

Here is an example using texdepth.

ps.1.4

texld r0, t0 //	Sample texture from texture stage 0 (dest
	register number) into r0.
//	Use texture coordinate data from t0.
texcrd r1.rgb, t1 //	Load a second set of texture coordinate data
add r5.rg, r0, r1 //	Add the two sets of texture coordinate data.
phase //	Phase marker, required when using texdepth i
texdepth r5 //	Calculate pixel depth as r5.r / r5.g.
//	Do other color calculations with shader outp

Microsoft DirectX 8.1 (pixel shader versions 1.2 and 1.3)

texdp3

Performs a three-component dot product between data in the texture register number and the texture coordinate set corresponding to the destination register number.

texdp3 dest, src

Registers

Argumen	t Description	Registers v _n c _n t _n r _n	Version
dest	Destination register		1.2, 1.3
SrC	Source register	Х	1.2, 1.3

Remarks

Texture registers must use the following sequence.

tex	t(n)	// Define tn as a standard 3-vector (tn must be
		<pre>// defined in some way before texdp3 uses it).</pre>
texdp3	t(m), t(n)	// where m > n
		<pre>// Perform a three-component dot product between</pre>
		<pre>// the texture coordinate set m. The scalar resul</pre>
		<pre>// replicated to all components of t(m).</pre>

Here is more detail about how the dot product is accomplished.

```
// The texdp3 instruction performs a three-component dot product and // replicates it to all four color channels. t(m)_{RGBA} = TextureCoordinates(stage m)_{UVW} \cdot t(n)_{RGB}
```

Microsoft DirectX 8.1 (pixel shader versions 1.2 and 1.3)

texdp3tex

Performs a three-component dot product and uses the result to do a 1-D texture lookup.

texdp3tex dest, src

Registers

Argumen	t Description	Registers v _n c _n t _n r _n	Version
dest	Destination register	Х	1.2, 1.3
SrC	Source register	Х	1.2, 1.3

Remarks

Texture registers must use the following sequence.

Here is more detail about how the dot product and texture lookup are done.

// The texdp3tex instruction performs a three-component dot product. u' = TextureCoordinates(stage m)_{UVW} • $t(n)_{RGB}$

// The result is used to sample the texture at texture stage m by pe // a 1-D lookup. t(m)_{RGBA} = TextureSample(stage m)_{RGBA} using (u',0,0) as coordinates. Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

texkill

texkill src

Cancels rendering of the current pixel if any of the first three components (UVW) of the texture coordinates is less than zero.

Registers

Argumont	t Description	Registers	Version
Argument Descriptio		$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	Version
STC	Source register	X	1.0, 1.1, 1.2, 1.3
SrC	Source register	X X	1.4 phase 2 only

To learn more about registers, see <u>Registers</u>.

Remarks

texkill does not sample any texture. It operates on the first three components of the texture coordinates given by the source register number. For ps 1.4, texkill operates on the data in the first three components of the source register.

You can use this instruction to implement arbitrary clip planes in the rasterizer.

When using vertex shaders, the application is responsible for applying the perspective transform. This can cause problems for the arbitrary clipping planes because if it contains anisomorphic scale factors, the clip planes need to be transformed as well. Therefore, it is best to provide an unprojected vertex position to use in the arbitrary clipper, which is the texture coordinate set identified by the **texkill** operator.

For ps 1.0, 1.1, 1.2, and 1.3, texkill operates on the texture coordinate set given by the source register number. In version 1.4, however, texkill operates on the data contained in the texture coordinate iterator register (tn) or in the temporary register (rn) that has been specified as the source.

When multisampling is enabled, any antialiasing effect achieved on polygon edges due to multisampling will not be achieved along any edge that has been generated by **texkill**. The pixel shader runs once per pixel.

Example

This example is for illustration only.

```
// This example masks out pixels that have negative texture coordina
// colors are interpolated from vertex colors provided in the vertex
// The shader is shown below.
             // version instruction
ps.1.0
texkill t0 // Mask out pixel using texture coordinates from stage
mov r0, v0
             // Move the diffuse color in v0 to r0.
// The rendered output from the pixel shader is shown below. It show
// vertex color data applied to a plane. The texture coordinate data
// is declared in the vertex data declaration in this example.
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color;
    FLOAT tu1, tv1;
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1|D
static CUSTOMVERTEX g_Vertices[]=
{
    // x y z color u1,
{ -1.0f, -1.0f, 0.0f, 0xffff0000, -0.5f,
                                                v1
                                               1.0f, },
    { 1.0f, -1.0f, 0.0f, 0xff00ff00,
                                        0.5f,
                                               1.0f, },
      1.0f, 1.0f, 0.0f, 0xff0000ff,
                                               0.0f, },
                                       0.5f,
    {
    { -1.0f, 1.0f, 0.0f, 0xffffffff, -0.5f,
                                               0.0f, },
};
```

// The texture coordinates range from -0.5 to 0.5 in u, and 0.0 to 1
// This instruction causes the negative u values get masked out.
// The first image shows the vertex colored applied to the quad with

// texkill instruction applied.
// The second image shows the result of the texkill instruction. The
// from the texture coordinates below 0 (where x goes from -0.5 to 0
// out. The background color (white) is used where the pixel color i

Microsoft Directx 8.1 (**pixel shader version 1.4**)

texld

Loads the destination register with color data (RGBA) sampled using the contents of the source register as texture coordinates. The sampled texture is the texture associated with the destination register number.

texld dest, src

Registers

Argument	Description	Registers	Version
¹ in Sument	Description	$v_n c_n t_n r_n$	v CI SIOII
dest	Destination register	Х	1.4
STC	Source register	Х	1.4 <u>phase</u> 1
		XX	1.4 phase 2

When using r(n) as a source register, the first three components (XYZ) must have been initialized in the previous **phase** of the shader.

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction samples the texture in the texture stage associated with the destination register number. The texture is sampled using texture coordinate data from the source register.

The syntax for the texld and texcrd instructions expose support for a projective divide with a <u>Texture Register Modifier</u>. For pixel shader version 1.4, the D3DTTFF_PROJECTED <u>texture transform flags</u> is always ignored.

Rules for using **texld**:

1. The same .xyz or .xyw modifier must be applied to every read of an individual t(n) register within both texcrd or texld instructions. If .xyw is

being used on t(n) register read(s), this can be mixed with other read(s) of the same t(n) register using .xyw_dw.

- 2. The _dz source modifier is only valid on texld with r(n) source register (thus phase 2 only).
- 3. The _dz source modifier may be used no more than two times per shader.

Examples

The texld instruction offers some control over which components of the source texture coordinate data are used. The complete set of allowed syntax for texld follows, and includes all valid source register modifiers, selectors, and write mask combinations.

```
texld r(m), t(n).xyz
// Uses xyz from t(n) to sample 1-D, 2-D, or 3-D texture.
texld r(m), t(n)
// Same as previous.
texld r(m), t(n).xyw
// Uses xyw (skipping z) from t(n) to sample 1-D, 2-D or 3-D texture
texld r(m), t(n)_dw.xyw
// Samples 1-D or 2-D texture at x/w, y/w from t(n). The result
// is undefined for a cube-map lookup.
texld r(m), r(n).xyz
// Samples 1-D, 2-D, or 3-D texture at xyz from r(m).
// This is possible in the second phase of the shader.
texld r(m), r(n)
// Same as previous.
texld r(m), r(n)_dz.xyz
// Samples 1-D or 2-D texture at x/z, y/z from r(m).
// Possible only in second phase.
// The result is undefined for a cube-map lookup.
texld r(n), r(n)_dz
// Same as previous.
```

Microsoft DirectX 8.1 (pixel shader version 1.3)

texm3x2depth

Calculate the depth value to be used in depth testing for this pixel.

texm3x2depth dest, src

Registers

Argumon	t Description	Registers v _n c _n t _n r _n	Varsian	
Arguinen	Description	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	VEISIUII	
dest	Destination register	X	1.3	
SrC	Source register	Х	1.3	

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction must be used with the **<u>texm3x2pad</u>** instruction.

When using these two instructions, texture registers must use the following sequence.

tex t(n)			<pre>// Define tn as a standard 3-vector.(tn // defined in some way before it is use</pre>
texm3x2pad	t(m),	t(n)	<pre>// Where m > n // Calculate z value.</pre>
texm3x2depth	t(m+1),	t(n)	<pre>// Calculate w value; use both z and w // find depth.</pre>

The depth calculation is done after using a dot product operation to find z and w. Here is more detail about how the depth calculation is accomplished.

```
// The texm3x2pad instruction calculates z.
z = TextureCoordinates(stage m)<sub>UVW</sub> • t(n)<sub>RGB</sub>
// The texm3x2depth instruction calculates w.
w = TextureCoordinates(stage m+1)<sub>UVW</sub> • t(n)<sub>RGB</sub>
```

```
// Calculate depth and store the result in t(m+1).
if (w == 0)
  t(m+1) = 1.0
else
  t(m+1) = z/w
```

The calculated depth is tagged to be used in the depth test for the pixel, replacing the existing depth test value for the pixel.

Be sure to clamp z/w to be in the range of (0-1). If z/w is outside this range, the result stored in the depth buffer will be undefined.

After executing tex3x2depth, register t(m+1) is no longer available for use in the shader.

When multisampling, using this instruction eliminates most of the benefit of the higher resolution depth buffer. Because the pixel shader executes once per pixel, the single depth value output by <u>texm3x2depth/texdepth</u> will be used for each of the sub-pixel depth comparison tests.

texm3x2pad

Performs the first row multiplication of a two-row matrix multiply. This instruction must be combined with either <u>texm3x2tex</u> or <u>texm3x2depth</u>. Refer to either of these instructions for details on using texm3x2pad.

texm3x2pad dest, src

Registers

Argumont	Description	Registers	Version
Argument	Description	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	VEISIOII
dest	Destination register	X	1.0, 1.1, 1.2, 1.3
STC	Source register	Х	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction cannot be used by itself.

texm3x2tex

Performs the final row of a 3×2 matrix multiply and uses the result to do a texture lookup. texm3x2tex must be used in conjunction with the texm3x2pad instruction.

texm3x2tex dest, src

Registers

Argument	t Description	Registers	Version
-	_	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	
dest	Destination register	Х	1.0, 1.1, 1.2, 1.3
STC	Source register	Х	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

The instruction is used as one of two instructions representing a 3×2 matrix multiply operation. This instruction must be used with the texm3x2pad.

When using these two instructions, texture registers must use the following sequence.

tex t(n)			<pre>// Define tn as a standard 3-vector (t // be defined in some way before it is</pre>
texm3x2pad	t(m),	t(n)	// where $m > n$
			<pre>// Perform first row of matrix multipl</pre>
texm3x2tex	t(m+1),	t(n)	// Perform second row of matrix multip
			<pre>// to get (u,v) to sample texture</pre>
			<pre>// associated with stage m+1.</pre>

Here is more detail about how the 3×2 multiply is accomplished.

// The texm3x2pad instruction performs the first row of the multiply $u' = t(n)_{RGB}$ • TextureCoordinates(stage m)_{UVW}

// The texm3x2tex instruction performs the second row of the multipl $v' = t(n)_{RGB}$ • TextureCoordinates(stage m+1)_{UVW}

```
// The texm3x2tex instruction samples the texture on stage (m+1) wit // stores the result in t(m+1). t(m+1)_{RGB} = TextureSample(stage m+1)_{RGB} using (u', v') as coordinates.
```

Example

// Here is an example shader with the texture maps and // the texture stages identified. ps.1.0 // Bind texture in stage 0 to register t0. tex t0 texm3x2pad t1, t0 // First row of matrix multiply. // Second row of matrix multiply to get (u,v) texm3x2tex t2, t0 // with which to sample texture in stage 2. mov r0, t2 // Output result. // This example requires the following textures in the following tex 11 // Stage 0 takes a map with (x, y, z) perturbation data. 11 // Stage 1 holds texture coordinates. No texture is required in the // // Stage 2 holds both texture coordinates as well as a 2-D texture s // that texture stage.

Microsoft DirectX 8.1 (pixel shader versions 1.2 and 1.3)

texm3x3

Performs a 3×3 matrix multiply when used in conjunction with two <u>texm3x3pad</u> instructions.

texm3x3 dest, src

Registers

Argument	t Description	$\frac{\text{Registers}}{v_n c_n t_n r_n} \text{Version}$	
dest	Destination register	Х	1.2, 1.3
STC	Source register	Х	1.2, 1.3

Remarks

This instruction is the same as the <u>texm3x3tex</u> instruction, without the texture lookup.

This instruction is used as the final of three instructions representing a 3×3 matrix multiply operation. The 3×3 matrix is comprised of the texture coordinates of the third texture stage, and by the two preceding texture stages. Any texture assigned to any of the three texture stages is ignored.

This instruction must be used with two texm3x3pad instructions. Texture registers must follow the following sequence.

tex t(n)			<pre>// Define tn as a standard 3-vector (tn mus // be defined in some way before it is used</pre>
texm3x3pad	t(m),	t(n)	// where m > n
			<pre>// Perform first row of matrix multiply.</pre>
			<pre>// Perform second row of matrix multiply.</pre>
texm3x3	t(m+2),	t(n)	<pre>// Perform third row of matrix multiply to</pre>
			// 3-vector result.

Here is more detail about how the 3×3 multiply is accomplished.

// The first texm3x3pad instruction performs the first row of the mu // to find u'. u' = TextureCoordinates(stage m)_{UVW} • $t(n)_{RGB}$

// The second texm3x3pad instruction performs the second row of the // to find v'.

v' = TextureCoordinates(stage m+1)_{UVW} • t(n)_{RGB}

// The texm3x3tex instruction performs the third row of the multiply
// to find w'.

 $w' = TextureCoordinates(stage m+2)_{UVW} \cdot t(n)_{RGB}$

// Place the result of the matrix multiply in the destination regist $t(m+2)_{RGBA} = (u', v', w', 1)$

texm3x3pad

Performs the first or second row multiply of a three-row matrix multiply. This instruction must be used in combination with <u>texm3x3</u>, <u>texm3x3spec</u>, <u>texm3x3vspec</u>, or <u>texm3x3tex</u>.

texm3x3pad dest, src

Registers

Argumont	Description	Registers	Version
Argument	Description	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	VEISIOII
dest	Destination register	Х	1.0, 1.1, 1.2, 1.3
STC	Source register	Х	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction cannot be used by itself.

texm3x3tex

Performs a 3×3 matrix multiply and uses the result to do a texture lookup. texm3x3tex must be used with two texm3x3pad instructions.

texm3x3tex dest, src

Argumont	Description	Registers	Version
Argument	Description	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	VEISIOII
dest	Destination register	X	1.0, 1.1, 1.2, 1.3
STC	Source register	Х	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction is used as the final of three instructions representing a 3×3 matrix multiply operation, followed by a texture lookup. The 3×3 matrix is comprised of the texture coordinates of the third texture stage and the two preceding texture stages. The resulting three-component vector (u,v,w) is used to sample the texture in stage 3. Any texture assigned to the preceding two texture stages is ignored. The 3×3 matrix multiply is typically useful for orienting a normal vector to the correct tangent space for the surface being rendered.

This instruction must be used with two texm3x3pad instructions. Texture registers must use the following sequence.

Here is more detail about how the 3×3 multiply is accomplished.

// The first texm3x3pad instruction performs the first row of the mu
// to find u'.
u' = TextureCoordinates(stage m)_{UVW} • t(n)_{RGB}
// The second texm3x3pad instruction performs the second row of the
// to find v'.
v' = TextureCoordinates(stage m+1)_{UVW} • t(n)_{RGB}
// The texm3x3spec instruction performs the third row of the multipl
// to find w'.
w' = TextureCoordinates(stage m+2)_{UVW} • t(n)_{RGB}
// Lastly, the texm3x3tex instruction samples t(m+2) with (u',v',w')
// and stores the result in t(m+2).
t(m+2)_{RGBA} = TextureSample(stage m+2)_{RGBA} using (u', v', w') as coordir

Example

// Here is an example shader with the texture maps identified and // the texture stages identified. ps.1.0 tex t0 // Bind texture in stage 0 to register t0. texm3x3pad t1, t0 // First row of matrix multiply. // Second row of matrix multiply. texm3x3pad t2, t0 // Third row of matrix multiply to get a texm3x3tex t3, t0 // 3-vector with which to sample texture at st // output result. mov r0, t3 // This example requires the following texture stage setup. 11 // Stage 0 is assigned a texture map with normal data. This is often // referred to as a bump map. The data is (XYZ) normals for // each texel. Texture coordinate set 0 defines how to sample this // normal map. 11 // Texture coordinate set 1 is assigned to row 1 of the 3×3 matrix. // Any texture assigned to stage 1 is ignored. 11 // Texture coordinate set 2 is assigned to row 2 of the 3×3 matrix. // Any texture assigned to stage 2 is ignored. 11 // Texture coordinate set 3 is assigned to row 3 of the 3×3 matrix. // A volume or cube texture should be set to stage 3 for lookup by // transformed 3-D vector. 11

texm3x3spec

Performs a 3×3 matrix multiply and uses the result to perform a texture lookup. This can be used for specular reflection and environment mapping. texm3x3spec must be used in conjunction with two **texm3x3pad** instructions.

texm3x3spec dest, src0, src1, src2

Registers

Argument	Description	Registers	Version
Aiguinein		$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	V CI SIOII
dest	Destination register	Х	1.0, 1.1, 1.2, 1.3
src0, src1	Source register	Х	1.0, 1.1, 1.2, 1.3
src2	Source register	Х	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction performs the final row of a 3×3 matrix multiply, uses the resulting vector as a normal vector to reflect an eye-ray vector, and then uses the reflected vector to perform a texture lookup. The shader reads the eye-ray vector from a constant register. The 3×3 matrix multiply is typically useful for orienting a normal vector to the correct tangent space for the surface being rendered.

The 3×3 matrix is comprised of the texture coordinates of the third texture stage and the two preceding texture stages. The resulting post reflection vector (u,v,w) is used to sample the texture on the final texture stage. Any texture assigned to the preceding two texture stages is ignored.

This instruction must be used with two texm3x3pad instructions. Texture registers must use the following sequence.

tex t(n)	// Define tn as a standard 3-vector (t
	<pre>// be defined in some way before it is</pre>

// where m > ntexm3x3pad t(m), t(n) // Perform first row of matrix multipl texm3x3pad t(m+1), t(n) // Perform second row of matrix multip texm3x3spec t(m+2), t(n), c0 // Perform third row of matrix multipl // Then do a texture lookup on the tex // associated with texture stage m+2. // The first texm3x3pad instruction performs the first row of the mu // to find u'. u' = TextureCoordinates(stage m)_{UVW} • t(n)_{RGB} // The second texm3x3pad instruction performs the second row of the // to find v'. $v' = TextureCoordinates(stage m+1)_{UVW} \cdot t(n)_{RGB}$ // The texm3x3spec instruction performs the third row of the multipl // to find w'. $w' = TextureCoordinates(stage m+2)_{UVW} \cdot t(n)_{RGB}$ // The texm3x3spec instruction then does a reflection calculation. $(u'', v'', w'') = 2*[(N \cdot E)/(N \cdot N)]*N - E$ // where the normal N is given by // N = (u', v', w') // and the eve-ray vector E is given by the constant register // E = c# (Any constant register--c0, c1, c2, etc.--can be used.) // Lastly, the texm3x3spec instruction samples t(m+2) with (u'',v'',w)// and stores the result in t(m+2). $t(m+2)_{RGBA}$ = TextureSample(stage m+2)_{RGBA} using (u', v', w') as coore

Example

// Here is an example shader with the texture maps and // the texture stages identified. ps.1.0 // Bind texture in stage 0 to register t0 tex t0 // be defined in some way before it is use // First row of matrix multiply. texm3x3pad t1, t0 texm3x3pad t2, t0 // Second row of matrix multiply. texm3x3spec t3, t0, c# // Third row of matrix multiply to get a 3 // Reflect 3-vector by the eye-ray vector // Use reflected vector to lookup texture // stage 3 mov r0, t3 // output result

// This example requires the following texture stage setup.

11 // Stage 0 is assigned a texture map with normal data. This is often // referred to as a bump map. The data is (XYZ) normals for // each texel. Texture coordinates at stage n defines where to sampl // normal map. 11 // Texture coordinate set m is assigned to row 1 of the 3×3 matrix. // Any texture assigned to stage m is ignored. // // Texture coordinate set m+1 is assigned to row 2 of the 3×3 matrix // Any texture assigned to stage m+1 is ignored. // // Texture coordinate set m+2 is assigned to row 3 of the 3×3 matrix // Stage m+2 is assigned a volume or cube texture map. The texture p // color data (RGBA). 11 // The eye-ray vector E is given by a // constant register E = c#.

texm3x3vspec

Performs a 3×3 matrix multiply and uses the result to perform a texture lookup. This can be used for specular reflection and environment mapping where the eye-ray vector is not constant. texm3x3vspec must be used in conjunction with two texm3x3pad instructions.

If the eye-ray vector is constant, the <u>texm3x3spec</u> instruction will perform the same matrix multiply and texture lookup.

texm3x3vspec dest, src

Registers

Argumont	Description	Registers	Version
Argument	d Description	$\mathbf{v}_{\mathbf{n}} \mathbf{c}_{\mathbf{n}} \mathbf{t}_{\mathbf{n}} \mathbf{r}_{\mathbf{n}}$	VEISIOII
dest	Destination register	X	1.0, 1.1, 1.2, 1.3
SrC	Source register	X	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction performs the final row of a 3×3 matrix multiply operation, interprets the resulting vector as a normal vector to reflect an eye-ray vector, and then uses the reflected vector as a texture address for a texture lookup. It works just like <u>texm3x3spec</u>, except that the eye-ray vector is taken from the fourth component of the texture coordinates. The 3×3 matrix multiply is typically useful for orienting a normal vector to the correct tangent space for the surface being rendered.

The 3×3 matrix is comprised of the texture coordinates of the third texture stage and the two preceding texture stages. The resulting post-reflection vector (UVW) is used to sample the texture in stage 3. Any texture assigned to the preceding two texture stages is ignored. This instruction must be used with the **texm3x3pad** instruction. Texture registers must use the following sequence.

tex t(n) // Define tn as a standard 3-vector (tn // be defined in some way before it is u texm3x3pad t(m), t(n) // where m > n// Perform first row of matrix multiply. // Perform second row of matrix multiply texm3x3pad t(m+1), t(n)texm3x3vspec t(m+2), t(n)// Perform third row of matrix multiply. // Then do a texture lookup on the textu // associated with texture stage m+2. // The first texm3x3pad instruction performs the first row of the mu // to find u'. u' = TextureCoordinates(stage m)_{UVW} • t(n)_{RGB} // The second texm3x3pad instruction performs the second row of the // to find v'. v' = TextureCoordinates(stage m+1)_{UVW} • t(n)_{RGB} // The texm3x3spec instruction performs the third row of the multipl // to find w'. $w' = TextureCoordinates(stage m+2)_{UVW} \cdot t(n)_{RGB}$ // The texm3x3vspec instruction also does a reflection calculation. $(u'', v'', w'') = 2*[(N \cdot E)/(N \cdot N)]*N - E$ // where the normal N is given by // N = (u', v', w') // and the eye-ray vector E is given by // E = $(TextureCoordinates(stage m)_0, TextureCoordinates(stage m+1)_0)$ // Lastly, the texm3x3vspec instruction samples t(m+2) with (u'',v'', // and stores the result in t(m+2). $t(m+2)_{RGBA}$ = TextureSample(stage m+2)_{RGBA} using (u'', v'', w'') as coord

Example

// Here is an example shader with the texture maps identified and // the texture stages identified. ps.1.0 tex t0 // Bind texture in stage 0 to register t0. texm3x3pad t1, t0 // First row of matrix multiply. texm3x3pad t2, t0 // Second row of matrix multiply. texm3x3vspec t3, t0 // Third row of matrix multiply to get a 3-vec // Reflect 3-vector by the eye-ray vector.

// Use reflected vector to do a texture lookup // at stage 3. mov r0, t3 // Output result. // This example requires the following texture stage setup. 11 // Stage 0 is assigned a texture map with normal data. This is often // referred to as a bump map. The data is (XYZ) normals for // each texel. Texture coordinates at stage n defines how to sample // normal map. 11 // Texture coordinate set m is assigned to row 1 of the 3×3 matrix. // Any texture assigned to stage m is ignored. 11 // Texture coordinate set m+1 is assigned to row 2 of the 3×3 matrix // Any texture assigned to stage m+1 is ignored. 11 // Texture coordinate set m+2 is assigned to row 3 of the 3×3 matrix // Stage m+2 is assigned a volume or cube texture map. The texture p // color data (RGBA). 11 // The eye-ray vector E is passed into the instruction in the fourth // component (q) of the texture coordinate data at stages m, m+1, an

texreg2ar

Interprets the alpha and red color components of the source register as texture address data (u,v) to sample the texture at the stage corresponding to the destination register number. The result is stored in the destination register.

texreg2ar dest, src

Registers

Argumont	Description	Registers	Version
Argument		$\mathbf{v}_n \mathbf{c}_n \mathbf{t}_n \mathbf{r}_n$	V CI SIUII
dest	Destination register	Х	1.0, 1.1, 1.2, 1.3
SrC	Source register	Х	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction is useful for color-space remapping operations.

```
// Here is an example of the sequence the instruction follows.
tex t(n)
texreg2ar t(m), t(n) where m > n
// Here is more detail about how the remapping is accomplished.
// The first instruction loads the texture color (RGBA) into registe
tex tn
// The second instruction remaps the color.
t(m)<sub>RGBA</sub> = TextureSample(stage m)<sub>RGBA</sub> using t(n)<sub>AR</sub> as coordinates.
```

For this instruction, the source register must use unsigned data. Use of signed or mixed data in the source register will produce undefined results. For more information, see <u>D3DFORMAT</u>.

texreg2gb

Interprets the green and blue color components of the source register as texture address data to sample the texture at the stage corresponding to the destination register number.

texreg2gb dest, src

Registers

Argumen	t Description	Registers v _n c _n t _n r _n	Version
dest	Destination register	Х	1.0, 1.1, 1.2, 1.3
SrC	Source register	Х	1.0, 1.1, 1.2, 1.3

To learn more about registers, see <u>Registers</u>.

Remarks

This instruction is useful for color-space remapping operations.

```
// Here is an example of the sequence the instruction follows.
tex t(n)
texreg2gb t(m), t(n) where m > n
// Here is more detail about how the remapping is accomplished.
// The first instruction loads the texture color (RGBA) into registe
tex tn
// The second instruction remaps the color.
t(m)<sub>RGBA</sub> = TextureSample(stage m)<sub>RGBA</sub> using t(n)<sub>GB</sub> as coordinates.
```

For this instruction, the source register must use unsigned data. Use of signed or mixed data in the source register will produce undefined results. For more information, see <u>D3DFORMAT</u>.

Microsoft DirectX 8.1 (pixel shader versions 1.2 and 1.3)

texreg2rgb

Interprets the red, green, and blue (RGB) color components of the source register as texture address data in order to sample the texture at the stage corresponding to the destination register number. The result is stored in the destination register.

texreg2rgb dest, src

Registers

Argument	t Description	Registers v _n c _n t _n r _n	Version
dest	Destination register	X	1.2, 1.3
SrC	Source register	Х	1.2, 1.3

Remarks

This instruction is useful for color-space remapping operations. It supports twodimensional (2-D) and three-dimensional (3-D) coordinates. It can be used just like the <u>texreg2ar</u> or <u>texreg2gb</u> to remap 2-D data. However, this instruction also supports 3-D data so it can be used with cube maps and 3-D volume textures.

```
// Here is an example of the sequence the instruction follows.
tex t(n)
texreg2rgb t(m), t(n) where m > n
```

Here is more detail about how the remapping is accomplished.

```
// The first instruction loads the texture color (RGBA) into registe
tex tn
// The second instruction remaps the color.
t(m)<sub>RGB</sub> = TextureSample(stage m)<sub>RGB</sub> using t(n)<sub>RGB</sub> as coordinates.
```

Microsoft DirectX 8.1 (C++)

IDirect3DDevice8::SetPixelShaderCor

Sets the values in the pixel constant array.

```
HRESULT SetPixelShaderConstant(
   DWORD Register,
   CONST void* pConstantData,
   DWORD ConstantCount
);
```

Parameters

Register

[in] Register address at which to start loading data into the pixel constant array.

pConstantData

[in] Pointer to the data block holding the values to load into the pixel constant array. The size of the data block is (*ConstantCount* * 4 * sizeof(float)).

ConstantCount

[in] Number of constants to load into the pixel constant array. Each constant is comprised of four floating-point values.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be <u>D3DERR_INVALIDCALL</u>.

Remarks

This is the method used to load the constant registers of the pixel shader assembler.

Requirements

Header: Declared in D3d8.h. **Import Library:** Use D3d8.lib.

See Also

IDirect3DDevice8::GetPixelShaderConstant

Microsoft DirectX 8.1 (C++)

IDirect3DDevice8::CreatePixelShader

Creates a pixel shader.

```
HRESULT CreatePixelShader(
    CONST DWORD* pFunction,
    DWORD* pHandle
);
```

Parameters

pFunction

[in] Pointer to the pixel shader function token array, specifying the blending operations. This value cannot be NULL.

pHandle

[out, retval] Pointer to the returned pixel shader handle.

Return Values

If the method succeeds, the return value is <u>D3D_OK</u>.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL D3DERR_OUTOFVIDEOMEMORY E_OUTOFMEMORY

Requirements

Header: Declared in D3d8.h. **Import Library:** Use D3d8.lib.

See Also

IDirect3DDevice8::DeletePixelShader, D3DXAssembleShader, D3DXAssembleShaderFromFile

Microsoft DirectX 8.1 (C++)

D3DXAssembleShader

Assembles an ASCII description of a shader into binary form, where the shader source is in memory.

```
HRESULT D3DXAssembleShader(
  LPCVOID pSrcData,
  UINT SrcDataLen,
  DWORD Flags,
  LPD3DXBUFFER* ppConstants,
  LPD3DXBUFFER* ppCompiledShader,
  LPD3DXBUFFER* ppCompilationErrors
);
```

Parameters

pSrcData

[in] Pointer to the source code.

SrcDataLen

[in] Size of the source code, in bytes.

Flags

[in] A combination of the **D3DXASM** flags, specifying assembly options. *ppConstants*

[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned constant declarations. These constants are returned as a vertex shader declaration fragment. It is up to the application to insert the contents of this buffer into their declaration. For pixel shaders this parameter is meaningless because constant declarations are included in the assembled shader. This parameter is ignored if it is NULL.

ppCompiledShader

[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned compiled object code. This parameter is ignored if it is NULL. *ppCompilationErrors*

[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned ASCII error messages. This parameter is ignored if it is NULL.

Return Values

If the function succeeds, the return value is <u>D3D_OK</u>.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL D3DXERR_INVALIDDATA E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8core.h. **Import Library:** Use D3dx8.lib.

Microsoft DirectX 8.1 (C++)

D3DCAPS8

Represents the capabilities of the hardware exposed through the Microsoft® Direct3D® object.

typedef struct _D3DCAPS	8 {
D3DDEVTYPE	DeviceType;
UINT	AdapterOrdinal;
DWORD	Caps;
DWORD	Caps2;
DWORD	Caps3;
DWORD	PresentationIntervals;
DWORD	CursorCaps;
DWORD	DevCaps;
DWORD DWORD DWORD DWORD DWORD DWORD DWORD DWORD DWORD DWORD DWORD DWORD DWORD DWORD DWORD	<pre>PrimitiveMiscCaps; RasterCaps; ZCmpCaps; SrcBlendCaps; DestBlendCaps; AlphaCmpCaps; ShadeCaps; TextureCaps; TextureFilterCaps; CubeTextureFilterCaps; VolumeTextureFilterCaps; TextureAddressCaps; VolumeTextureAddressCaps;</pre>
DWORD	LineCaps;
DWORD	MaxTextureWidth, MaxTextureHeight;
DWORD	MaxVolumeExtent;
DWORD	MaxTextureRepeat;
DWORD	MaxTextureAspectRatio;
DWORD	MaxAnisotropy;
float	MaxVertexW;
float	GuardBandLeft;
float	GuardBandTop;
float	GuardBandRight;

float	GuardBandBottom;
float	ExtentsAdjust;
DWORD	StencilCaps;
DWORD	FVFCaps;
DWORD	TextureOpCaps;
DWORD	MaxTextureBlendStages;
DWORD	MaxSimultaneousTextures;
DWORD	VertexProcessingCaps;
DWORD	MaxActiveLights;
DWORD	MaxUserClipPlanes;
DWORD	MaxVertexBlendMatrices;
DWORD	MaxVertexBlendMatrixIndex;
float	MaxPointSize;
DWORD	MaxPrimitiveCount;
DWORD	MaxVertexIndex;
DWORD	MaxStreams;
DWORD	MaxStreamStride;
DWORD	VertexShaderVersion;
DWORD	MaxVertexShaderConst;
DWORD float } D3DCAPS8;	PixelShaderVersion; MaxPixelShaderValue;

Members

DeviceType

Member of the **D3DDEVTYPE** enumerated type, which identifies what type of resources are used for processing vertices.

AdapterOrdinal

Adapter on which this Direct3DDevice object was created. This ordinal is valid only to pass to methods of the **IDirect3D8** interface that created this Direct3DDevice object. The **IDirect3D8** interface can always be retrieved by calling **IDirect3DDevice8::GetDirect3D**.

Caps

The following driver-specific capability.

D3DCAPS_READ_SCANLINE

Display hardware is capable of returning the current scan line.

Caps2

The following driver-specific capabilities.

D3DCAPS2_CANCALIBRATEGAMMA

The system has a calibrator installed that can automatically adjust the gamma ramp so that the result is identical on all systems that have a calibrator. To invoke the calibrator when setting new gamma levels, use the D3DSGR_CALIBRATE flag when calling the

IDirect3DDevice8::SetGammaRamp method. Calibrating gamma ramps incurs some processing overhead and should not be used frequently.

D3DCAPS2_CANRENDERWINDOWED

The driver is capable of rendering in windowed mode.

D3DCAPS2_CANMANAGERESOURCE

The driver is capable of managing resources. On such drivers, D3DPOOL_MANAGED resources will be managed by the driver. To have Direct3D override the driver so that Direct3D manages resources, use the D3DCREATE_DISABLE_DRIVER_MANAGEMENT flag when calling **IDirect3D8::CreateDevice**

D3DCAPS2_DYNAMICTEXTURES

The driver supports dynamic textures.

D3DCAPS2_FULLSCREENGAMMA

The driver supports dynamic gamma ramp adjustment in full-screen mode.

D3DCAPS2_NO2DDURING3DSCENE

When the D3DCAPS2_NO2DDURING3DSCENE capability is set by the driver, it means that 2-D operations cannot be performed between calls to **IDirect3DDevice8::BeginScene** and **IDirect3DDevice8::EndScene**.

Typically, this capability is set by hardware that partitions the scene and then renders each partition in sequence. The partitioning is performed in the driver, and the hardware contains a small color and depth buffer that corresponds to the size of the image partition. Typically, on this type of rendering hardware, once each part of the image is rendered, the data in the color buffers are written to video memory and the contents of the depth buffer are discarded. Also, note that 3-D rendering does not start until **EndScene** is encountered. Next, the scene is processed in regions. Therefore, the processing order cannot be guaranteed. For example, the first region that is processed, typically the upper left corner of the window, might include the last triangle in the frame. This differs from more traditional graphics systems in which each command is processed sequentially in the order that it was sent. The 2-D operations are implied to occur at some fixed point in the processing. In the systems that set D3DCAPS2_NO2DDURING3DSCENE, the processing order is not guaranteed. Therefore, the display adapter might discard 2-D operations that are encountered during 3-D rendering.

In general, it is recommended that 2-D operations be performed outside of a **BeginScene** and **EndScene** pair. If 2-D operations are to be performed between a **BeginScene** and **EndScene** pair, then it is necessary to check the D3DCAPS2_NO2DDURING3DSCENE capability. If it is set, the application must expect that any 2-D operation that occurs between **BeginScene** and **EndScene** will be discarded. For more information on writing applications for systems that set D3DCAPS2_NO2DDURING3DSCENE, see Remarks.

D3DCAPS2_RESERVED

Reserved; not used.

Caps3

The following driver-specific capabilities.

D3DCAPS3_ALPHA_FULLSCREEN_FLIP_OR_DISCARD The device will work as expected with the D3DRS_ALPHABLENDENABLE render state when a full-screen application uses D3DSWAPEFFECT_FLIP or D3DRS_SWAPEFFECT_DISCARD. D3DRS_ALPHABLENDENABLE works as expected when using D3DSWAPEFFECT_COPY and D3DSWAPEFFECT_COPYSYNC.

D3DCAPS3_RESERVED

Reserved; not used.

PresentationIntervals

Bit mask of values representing what presentation swap intervals are available.

D3DPRESENT_INTERVAL_IMMEDIATE

The driver supports an immediate presentation swap interval.

D3DPRESENT_INTERVAL_ONE

The driver supports a presentation swap interval of every screen refresh.

D3DPRESENT_INTERVAL_TWO

The driver supports a presentation swap interval of every second screen refresh.

D3DPRESENT_INTERVAL_THREE

The driver supports a presentation swap interval of every third screen refresh.

D3DPRESENT_INTERVAL_FOUR

The driver supports a presentation swap interval of every fourth screen refresh.

CursorCaps

Bit mask indicating what hardware support is available for cursors. D3DCURSORCAPS_COLOR

A full-color cursor is supported in hardware. Specifically, this flag indicates that the driver supports at least a hardware color cursor in high-resolution modes (with scan lines greater than or equal to 400).

D3DCURSORCAPS_LOWRES

A full-color cursor is supported in hardware. Specifically, this flag indicates that the driver supports a hardware color cursor in both high-resolution and low-resolution modes (with scan lines less than 400).

Direct3D does not define alpha-blending cursor capabilities.

DevCaps

Flags identifying the capabilities of the device.

D3DDEVCAPS_CANBLTSYSTONONLOCAL

Device supports blits from system-memory textures to nonlocal videomemory textures.

D3DDEVCAPS_CANRENDERAFTERFLIP

Device can queue rendering commands after a page flip. Applications do not change their behavior if this flag is set; this capability simply means that the device is relatively fast.

D3DDEVCAPS_DRAWPRIMTLVERTEX

Device exports a DrawPrimitive-aware hardware abstraction layer (HAL).

D3DDEVCAPS_EXECUTESYSTEMMEMORY

Device can use <u>execute buffers</u> from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY

Device can use <u>execute buffers</u> from video memory.

D3DDEVCAPS_HWRASTERIZATION

Device has hardware acceleration for scene rasterization.

D3DDEVCAPS_HWTRANSFORMANDLIGHT

Device can support transformation and lighting in hardware.

D3DDEVCAPS_NPATCHES

Device supports N-patches.

D3DDEVCAPS_PUREDEVICE

Device can support rasterization, transform, lighting, and shading in hardware.

D3DDEVCAPS_QUINTICRTPATCHES

Device supports quintic Bézier curves and B-splines.

D3DDEVCAPS_RTPATCHES

Device supports rectangular and triangular patches.

D3DDEVCAPS_RTPATCHHANDLEZERO

When this device capability is set, the hardware architecture does not require caching of any information, and uncached patches (handle zero) will be drawn as efficiently as cached ones. Note that setting D3DDEVCAPS_RTPATCHHANDLEZERO does not mean that a patch with handle zero can be drawn. A handle-zero patch can always be drawn whether this cap is set or not.

D3DDEVCAPS_SEPARATETEXTUREMEMORIES Device is texturing from separate memory pools.

D3DDEVCAPS_TEXTURENONLOCALVIDMEM

Device can retrieve textures from non-local video memory.

D3DDEVCAPS_TEXTURESYSTEMMEMORY

Device can retrieve textures from system memory.

D3DDEVCAPS_TEXTUREVIDEOMEMORY

Device can retrieve textures from device memory.

D3DDEVCAPS_TLVERTEXSYSTEMMEMORY

Device can use buffers from system memory for transformed and lit vertices.

D3DDEVCAPS_TLVERTEXVIDEOMEMORY

Device can use buffers from video memory for transformed and lit vertices.

PrimitiveMiscCaps

General capabilities for this primitive. This member can be one or more of the following flags.

D3DPMISCCAPS_BLENDOP

Device supports the alpha-blending operations defined in the **D3DBLENDOP** enumerated type.

D3DPMISCCAPS_CLIPPLANESCALEDPOINTS

Device correctly clips scaled points of size greater than 1.0 to userdefined clipping planes.

D3DPMISCCAPS_CLIPTLVERTS

Device clips post-transformed vertex primitives.

D3DPMISCCAPS_COLORWRITEENABLE

Device supports per-channel writes for the render target color buffer through the D3DRS_COLORWRITEENABLE state.

D3DPMISCCAPS_CULLCCW

The driver supports counterclockwise culling through the D3DRS_CULLMODE state. (This applies only to triangle primitives.) This flag corresponds to the D3DCULL_CCW member of the D3DCULL enumerated type.

D3DPMISCCAPS_CULLCW

The driver supports clockwise triangle culling through the D3DRS_CULLMODE state. (This applies only to triangle primitives.) This flag corresponds to the D3DCULL_CW member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_CULLNONE

The driver does not perform triangle culling. This corresponds to the D3DCULL_NONE member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_LINEPATTERNREP

The driver can handle values other than 1 in the **wRepeatFactor** member of the **D3DLINEPATTERN** structure. (This applies only to line-drawing primitives.)

D3DPMISCCAPS_MASKZ

Device can enable and disable modification of the depth buffer on pixel operations.

D3DPMISCCAPS_TSSARGTEMP

Device supports D3DTA_TEMP for temporary register.

RasterCaps

Information on raster-drawing capabilities. This member can be one or more of the following flags.

D3DPRASTERCAPS_ANISOTROPY

Device supports anisotropic filtering.

D3DPRASTERCAPS_ANTIALIASEDGES

Device can antialias lines forming the convex outline of objects. For more information, see <u>D3DRS_EDGEANTIALIAS</u>.

D3DPRASTERCAPS_COLORPERSPECTIVE

Device iterates colors perspective correct.

D3DPRASTERCAPS_DITHER

Device can dither to improve color resolution.

D3DPRASTERCAPS_FOGRANGE

Device supports range-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene.

D3DPRASTERCAPS_FOGTABLE

Device calculates the fog value by referring to a lookup table containing fog values that are indexed to the depth of a given pixel.

D3DPRASTERCAPS_FOGVERTEX

Device calculates the fog value during the lighting operation, and interpolates the fog value during rasterization.

```
D3DPRASTERCAPS_MIPMAPLODBIAS
```

Device supports level-of-detail (LOD) bias adjustments. These bias adjustments enable an application to make a mipmap appear crisper or less sharp than it normally would. For more information about LOD bias in mipmaps, see <u>D3DTSS_MIPMAPLODBIAS</u>.

D3DPRASTERCAPS_PAT

The driver can perform patterned drawing lines or fills with D3DRS_LINEPATTERN for the primitive being queried.

D3DPRASTERCAPS_STRETCHBLTMULTISAMPLE

Device provides limited multisample support through a stretch-blt implementation. When this capability is set,

D3DRS_MULTISAMPLEANTIALIAS cannot be turned on and off in the middle of a scene. Multisample masking cannot be performed if this flag is set.

D3DPRASTERCAPS_WBUFFER

Device supports depth buffering using w.

D3DPRASTERCAPS_WFOG

Device supports w-based fog. W-based fog is used when a perspective projection matrix is specified, but affine projections still use z-based fog. The system considers a projection matrix that contains a nonzero value in the [3][4] element to be a perspective projection matrix.

D3DPRASTERCAPS_ZBIAS

Device supports z-bias values. These are integer values assigned to polygons that allow physically coplanar polygons to appear separate. For more information, see <u>D3DRS_ZBIAS</u>.

D3DPRASTERCAPS_ZBUFFERLESSHSR

Device can perform hidden-surface removal (HSR) without requiring

the application to sort polygons and without requiring the allocation of a depth buffer. This leaves more video memory for textures. The method used to perform HSR is hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no depth-buffer surface is associated with the rendering-target surface and the depth-buffer comparison test is enabled (that is, when the state value associated with the D3DRS_ZENABLE enumeration constant is set to TRUE).

D3DPRASTERCAPS_ZFOG

Device supports z-based fog.

D3DPRASTERCAPS_ZTEST

Device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels have been rendered.

ZCmpCaps

Z-buffer comparison capabilities. This member can be one or more of the following flags.

D3DPCMPCAPS_ALWAYS

Always pass the z test.

D3DPCMPCAPS_EQUAL

Pass the z test if the new z equals the current z.

D3DPCMPCAPS_GREATER

Pass the z test if the new z is greater than the current z.

D3DPCMPCAPS_GREATEREQUAL

Pass the z test if the new z is greater than or equal to the current z. D3DPCMPCAPS_LESS

Pass the z test if the new z is less than the current z.

D3DPCMPCAPS_LESSEQUAL

Pass the z test if the new z is less than or equal to the current z. D3DPCMPCAPS NEVER

Always fail the z test.

D3DPCMPCAPS NOTEQUAL

Pass the z test if the new z does not equal the current z.

SrcBlendCaps

Source-blending capabilities. This member can be one or more of the following flags. (The RGBA values of the source and destination are indicated by the subscripts s and d.)

D3DPBLENDCAPS_BOTHINVSRCALPHA

```
Source <u>blend factor</u> is (1–A<sub>s</sub>, 1–A<sub>s</sub>, 1–A<sub>s</sub>, 1–A<sub>s</sub>), and destination blend
     factor is (A<sub>s</sub>, A<sub>s</sub>, A<sub>s</sub>, A<sub>s</sub>); the destination blend selection is overridden.
D3DPBLENDCAPS BOTHSRCALPHA
     The driver supports the D3DBLEND_BOTHSRCALPHA blend mode.
     (This blend mode is obsolete. For more information, see
     D3DBLEND.)
D3DPBLENDCAPS DESTALPHA
     Blend factor is (A_d, A_d, A_d, A_d).
D3DPBLENDCAPS DESTCOLOR
     Blend factor is (R_d, G_d, B_d, A_d).
D3DPBLENDCAPS INVDESTALPHA
     Blend factor is (1-A_d, 1-A_d, 1-A_d, 1-A_d).
D3DPBLENDCAPS INVDESTCOLOR
     Blend factor is (1-R_d, 1-G_d, 1-B_d, 1-A_d).
D3DPBLENDCAPS INVSRCALPHA
     Blend factor is (1–A<sub>s</sub>, 1–A<sub>s</sub>, 1–A<sub>s</sub>, 1–A<sub>s</sub>).
D3DPBLENDCAPS INVSRCCOLOR
     Blend factor is (1-R_d, 1-G_d, 1-B_d, 1-A_d).
D3DPBLENDCAPS ONE
     Blend factor is (1, 1, 1, 1).
D3DPBLENDCAPS SRCALPHA
     Blend factor is (A_s, A_s, A_s, A_s).
D3DPBLENDCAPS SRCALPHASAT
     Blend factor is (f, f, f, 1); f = min(A_s, 1-A_d).
D3DPBLENDCAPS SRCCOLOR
     Blend factor is (R_s, G_s, B_s, A_s).
D3DPBLENDCAPS ZERO
     Blend factor is (0, 0, 0, 0).
```

DestBlendCaps

Destination-blending capabilities. This member can be the same capabilities that are defined for the **SrcBlendCaps** member.

AlphaCmpCaps

Alpha-test comparison capabilities. This member can include the same capability flags defined for the **ZCmpCaps** member. If this member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests. Otherwise, the flags identify the individual comparisons that are

supported for alpha testing.

ShadeCaps

Shading operations capabilities. It is assumed, in general, that if a device supports a given command at all, it supports the D3DSHADE_FLAT mode (as specified in the D3DSHADEMODE enumerated type). This flag specifies whether the driver can also support Gouraud shading and whether alpha color components are supported. When alpha components are not supported, the alpha value of colors generated is implicitly 255. This is the maximum possible alpha (that is, the alpha component is at full intensity).

The color, specular highlights, fog, and alpha interpolants of a triangle each have capability flags that an application can use to find out how they are implemented by the device driver.

This member can be one or more of the following flags.

D3DPSHADECAPS_ALPHAGOURAUDBLEND

Device can support an alpha component for Gouraud-blended transparency (the D3DSHADE_GOURAUD state for the **D3DSHADEMODE** enumerated type). In this mode, the <u>alpha color</u> <u>component</u> of a primitive is provided at vertices and interpolated across a face, along with the other color components.

D3DPSHADECAPS_COLORGOURAUDRGB

Device supports Gouraud shading. In this mode, the red, green, and blue components for a primitive are provided at vertices and interpolated across a face.

D3DPSHADECAPS_FOGGOURAUD

Device supports Gouraud shading of fog.

D3DPSHADECAPS_SPECULARGOURAUDRGB

Device supports Gouraud shading of specular highlights.

TextureCaps

Miscellaneous texture-mapping capabilities. This member can be one or more of the following flags.

D3DPTEXTURECAPS_ALPHA

Alpha in texture pixels is supported.

D3DPTEXTURECAPS_ALPHAPALETTE

Device can draw alpha from texture palettes.

D3DPTEXTURECAPS_CUBEMAP

Supports cube textures

D3DPTEXTURECAPS_CUBEMAP_POW2

Device requires that cube texture maps have dimensions specified as powers of 2.

- D3DPTEXTURECAPS_MIPCUBEMAP Device supports mipmapped cube textures.
- D3DPTEXTURECAPS_MIPMAP

Device supports mipmapped textures.

D3DPTEXTURECAPS_MIPVOLUMEMAP

Device supports mipmapped volume textures.

- D3DPTEXTURECAPS_NONPOW2CONDITIONAL
 - Conditionally supports the use of textures with dimensions that are not powers of 2. A device that exposes this capability can use such a texture if all of the following requirements are met.
 - The texture addressing mode for the texture stage is set to D3DTADDRESS_CLAMP.
 - Texture wrapping for the texture stage is disabled (D3DRS_WRAP*n* set to 0).
 - Mipmapping is not in use (use magnification filter only).
 - Texture formats must not be DXT1-5

A texture that is not a power of two cannot be set at a stage that will be read based on a shader computation (such as the bem, beml, or texm3x3 instructions in pixel shaders versions 1.0 to 1.3). For example, these textures can be used to store bumps that will be fed into texture reads, but not the environment maps that are used in texbem, texbeml, or texm3x3spec. This means that a texture with dimensions that are not powers of two cannot be addressed or sampled using texture coordinates computed within the shader. This type of operation is known as a *dependent read* and cannot be performed on these kinds of textures.

D3DPTEXTURECAPS_PERSPECTIVE

Perspective correction texturing is supported.

D3DPTEXTURECAPS_POW2

All textures must have widths and heights specified as powers of 2. This requirement does not apply to either cube textures or volume textures.

D3DPTEXTURECAPS_PROJECTED

Supports the D3DTTFF_PROJECTED texture transformation flag.

When applied, the device divides transformed texture coordinates by the last texture coordinate. If this capability is present, then the projective divide occurs per pixel. If this capability is not present, but the projective divide needs to occur anyway, then it is performed on a per-vertex basis by the Direct3D runtime.

D3DPTEXTURECAPS_SQUAREONLY

All textures must be square.

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE

Texture indices are not scaled by the texture size prior to interpolation. D3DPTEXTURECAPS_VOLUMEMAP

Device supports volume textures.

D3DPTEXTURECAPS_VOLUMEMAP_POW2

Device requires that volume texture maps have dimensions specified as powers of 2.

TextureFilterCaps

Texture-filtering capabilities for a Direct3DTexture object. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the

IDirect3DDevice8 interface. This member can be any combination of the following per-stage texture-filtering flags.

D3DPTFILTERCAPS_MAGFAFLATCUBIC

Device supports per-stage flat cubic filtering for magnifying textures. The flat cubic magnification filter is represented by the

D3DTEXF_FLATCUBIC member of the

<u>D3DTEXTUREFILTERTYPE</u> enumerated type.

D3DPTFILTERCAPS_MAGFANISOTROPIC

Device supports per-stage anisotropic filtering for magnifying textures. The anisotropic magnification filter is represented by the

D3DTEXF_ANISOTROPIC member of the

D3DTEXTUREFILTERTYPE enumerated type.

D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC

Device supports the per-stage Gaussian cubic filtering for magnifying textures. The Gaussian cubic magnification filter is represented by the D3DTEXF_GAUSSIANCUBIC member of the

D3DTEXTUREFILTERTYPE enumerated type.

D3DPTFILTERCAPS_MAGFLINEAR

Device supports per-stage bilinear interpolation filtering for magnifying textures. The bilinear interpolation magnification filter is represented by the D3DTEXF_LINEAR member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MAGFPOINT

Device supports per-stage point-sample filtering for magnifying textures. The point-sample magnification filter is represented by the D3DTEXF_POINT member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MINFANISOTROPIC

Device supports per-stage anisotropic filtering for minifying textures. The anisotropic minification filter is represented by the

D3DTEXF_ANISOTROPIC member of the

D3DTEXTUREFILTERTYPE enumerated type.

D3DPTFILTERCAPS_MINFLINEAR

Device supports per-stage bilinear interpolation filtering for minifying textures. The bilinear minification filter is represented by the D3DTEXF LINEAR member of the **D3DTEXTUREFILTERTYPE**

enumerated type.

D3DPTFILTERCAPS_MINFPOINT

Device supports per-stage point-sample filtering for minifying textures. The point-sample minification filter is represented by the D3DTEXF_POINT member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MIPFLINEAR

Device supports per-stage trilinear interpolation filtering for mipmaps. The trilinear interpolation mipmapping filter is represented by the D3DTEXF_LINEAR member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MIPFPOINT

Device supports per-stage point-sample filtering for mipmaps. The point-sample mipmapping filter is represented by the

D3DTEXF_POINT member of the **D3DTEXTUREFILTERTYPE** enumerated type.

CubeTextureFilterCaps

Texture-filtering capabilities for a Direct3DCubeTexture object. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the

IDirect3DDevice8 interface. This member can be any combination of the per-stage texture-filtering flags defined for the **TextureFilterCaps** member.

VolumeTextureFilterCaps

Texture-filtering capabilities for a Direct3DVolumeTexture object. Per-stage

filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the

IDirect3DDevice8 interface. This member can be any combination of the per-stage texture-filtering flags defined for the **TextureFilterCaps** member.

TextureAddressCaps

Texture-addressing capabilities for Direct3DTexture objects. This member can be one or more of the following flags.

D3DPTADDRESSCAPS_BORDER

Device supports setting coordinates outside the range [0.0, 1.0] to the border color, as specified by the D3DTSS_BORDERCOLOR texture-stage state.

D3DPTADDRESSCAPS_CLAMP

Device can clamp textures to addresses.

D3DPTADDRESSCAPS_INDEPENDENTUV

Device can separate the texture-addressing modes of the u and v coordinates of the texture. This ability corresponds to the D3DTSS_ADDRESSU and D3DTSS_ADDRESSV render-state values.

D3DPTADDRESSCAPS_MIRROR

Device can mirror textures to addresses.

D3DPTADDRESSCAPS_MIRRORONCE

Device can take the absolute value of the texture coordinate (thus, mirroring around 0), and then clamp to the maximum value.

D3DPTADDRESSCAPS_WRAP

Device can wrap textures to addresses.

VolumeTextureAddressCaps

Texture-addressing capabilities for Direct3DVolumeTexture objects. This member can be one or more of the flags defined for the

TextureAddressCaps member.

LineCaps

Defines the capabilities for line-drawing primitives.

D3DLINECAPS_ALPHACMP

Supports alpha-test comparisons.

D3DLINECAPS_BLEND

Supports source-blending.

D3DLINECAPS_FOG

Supports fog.

D3DLINECAPS_TEXTURE

Supports texture-mapping.

D3DLINECAPS_ZTEST

Supports z-buffer comparisons.

MaxTextureWidth and MaxTextureHeight

Maximum texture width and height for this device.

MaxVolumeExtent

Maximum volume extent.

MaxTextureRepeat

This number represents the maximum range of the integer bits of the postnormalized texture coordinates. A texture coordinate is stored as a 32-bit signed integer using 27 bits to store the integer part and 5 bits for the floating point fraction. The maximum integer index, 2²⁷, is used to determine the maximum texture coordinate, depending on how the hardware does texture-coordinate scaling.

Some hardware reports the cap

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE. For this case, the device defers scaling texture coordinates by the texture size until after interpolation and application of the texture address mode, so the number of times a texture can be wrapped is given by the integer value in **MaxTextureRepeat**.

Less desirably, on some hardware

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE is *not* set and the device scales the texture coordinates by the texture size (using the highest level of detail) prior to interpolation. This limits the number of times a texture can be wrapped to **MaxTextureRepeat** / textureSize.

Example:

```
Given MaxTextureRepeat = 32k and texture size = 4 KB:
if the hardware sets D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE
    # of times a texture can be wrapped = MaxTextureRepeat
    // which is 32k in this example
else
    # of times a texture can be wrapped = MaxTextureRepeat / textu
    // which is 2<sup>27</sup>/4k
```

MaxTextureAspectRatio

Maximum texture aspect ratio supported by the hardware, typically a power of 2.

MaxAnisotropy

Maximum valid value for the D3DTSS_MAXANISOTROPY texture-stage state.

MaxVertexW

Maximum W-based depth value that the device supports.

GuardBandLeft, GuardBandTop, GuardBandRight, and

GuardBandBottom

Screen space coordinates of the guard-band clipping region. Coordinates inside this rectangle but outside the viewport rectangle are automatically clipped.

ExtentsAdjust

Number of pixels to adjust the extents rectangle outward to accommodate antialiasing kernels.

StencilCaps

Flags specifying supported stencil-buffer operations. Stencil operations are assumed to be valid for all three stencil-buffer operation render states

(D3DRS_STENCILFAIL, D3DRS_STENCILPASS, and

D3DRS_STENCILFAILZFAIL).

D3DSTENCILCAPS_DECR

The **D3DSTENCILOP_DECR** operation is supported.

D3DSTENCILCAPS_DECRSAT

The **D3DSTENCILOP_DECRSAT** operation is supported. D3DSTENCILCAPS INCR

D3DSTENCILCAPS_INCR

The **D3DSTENCILOP_INCR** operation is supported.

D3DSTENCILCAPS_INCRSAT

The **D3DSTENCILOP_INCRSAT** operation is supported. D3DSTENCILCAPS_INVERT

The **D3DSTENCILOP_INVERT** operation is supported.

D3DSTENCILCAPS_KEEP

The **D3DSTENCILOP_KEEP** operation is supported.

D3DSTENCILCAPS_REPLACE

The **D3DSTENCILOP_REPLACE** operation is supported. D3DSTENCILCAPS ZERO

The **D3DSTENCILOP_ZERO** operation is supported.

For more information, see the **<u>D3DSTENCILOP</u>** enumerated type.

FVFCaps

Flexible vertex format capabilities.

D3DFVFCAPS_DONOTSTRIPELEMENTS

It is preferable that vertex elements not be stripped. That is, if the vertex format contains elements that are not used with the current

render states, there is no need to regenerate the vertices. If this capability flag is not present, stripping extraneous elements from the vertex format provides better performance.

D3DFVFCAPS_PSIZE

Point size is determined by either the render state or the vertex data.

If **D3DFVFCAPS_PSIZE** is set, point size can come from **D3DFVF_PSIZE** data in the FVF vertex declaration. Otherwise, point size is determined by the render state <u>D3DRS_POINTSIZE</u>.

If the application provides point size in both (the render state and the FVF data), the vertex data overrides the render-state data.

D3DFVFCAPS_TEXCOORDCOUNTMASK

Masks the low **WORD** of **FVFCaps**. These bits, cast to the **WORD** data type, describe the total number of texture coordinate sets that the device can simultaneously use for multiple texture blending. (You can use up to eight texture coordinate sets for any vertex, but the device can blend using only the specified number of texture coordinate sets.)

TextureOpCaps

Combination of flags describing the texture operations supported by this device. The following flags are defined.

D3DTEXOPCAPS_ADD

The **D3DTOP_ADD** texture-blending operation is supported. D3DTEXOPCAPS_ADDSIGNED

The **D3DTOP_ADDSIGNED** texture-blending operation is supported.

D3DTEXOPCAPS_ADDSIGNED2X

The **D3DTOP_ADDSIGNED2X** texture-blending operation is supported.

D3DTEXOPCAPS_ADDSMOOTH

The **D3DTOP_ADDSMOOTH** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDCURRENTALPHA

The **D3DTOP_BLENDCURRENTALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDDIFFUSEALPHA

The **D3DTOP_BLENDDIFFUSEALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDFACTORALPHA

The **D3DTOP_BLENDFACTORALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDTEXTUREALPHA

The **D3DTOP_BLENDTEXTUREALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDTEXTUREALPHAPM

The **D3DTOP_BLENDTEXTUREALPHAPM** texture-blending operation is supported.

D3DTEXOPCAPS_BUMPENVMAP

The **D3DTOP_BUMPENVMAP** texture-blending operation is supported.

D3DTEXOPCAPS_BUMPENVMAPLUMINANCE

The **D3DTOP_BUMPENVMAPLUMINANCE** texture-blending operation is supported.

D3DTEXOPCAPS_DISABLE

The **D3DTOP_DISABLE** texture-blending operation is supported. D3DTEXOPCAPS_DOTPRODUCT3

The **D3DTOP_DOTPRODUCT3** texture-blending operation is supported.

D3DTEXOPCAPS_LERP

The **D3DTOP_LERP** texture-blending operation is supported. D3DTEXOPCAPS_MODULATE

The **D3DTOP_MODULATE** texture-blending operation is supported. D3DTEXOPCAPS_MODULATE2X

The **D3DTOP_MODULATE2X** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATE4X

The **D3DTOP_MODULATE4X** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR The **D3DTOP_MODULATEALPHA_ADDCOLOR** textureblending operation is supported.

D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA The **D3DTOP_MODULATECOLOR_ADDALPHA** textureblending operation is supported.

D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR The **D3DTOP_MODULATEINVALPHA_ADDCOLOR** textureblending operation is supported. D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA The **D3DTOP_MODULATEINVCOLOR_ADDALPHA** textureblending operation is supported.

D3DTEXOPCAPS_MULTIPLYADD

The **D3DTOP_MULTIPLYADD** texture-blending operation is supported.

D3DTEXOPCAPS_PREMODULATE

The **D3DTOP_PREMODULATE** texture-blending operation is supported.

D3DTEXOPCAPS_SELECTARG1

The **D3DTOP_SELECTARG1** texture-blending operation is supported.

D3DTEXOPCAPS_SELECTARG2

The **D3DTOP_SELECTARG2** texture-blending operation is supported.

D3DTEXOPCAPS_SUBTRACT

The **D3DTOP_SUBTRACT** texture-blending operation is supported. **MaxTextureBlendStages**

Maximum number of texture-blending stages supported. This value is the number of blenders available. In the DirectX 8.x programmable pipeline, this corresponds to the number of instructions supported by pixel shaders on this particular implementation.

MaxSimultaneousTextures

Maximum number of textures that can be simultaneously bound to the texture blending stages. This value is the number of textures that can be used in a single pass. If the same texture is used in two blending stages, it counts as two when compared against the MaxSimultaneousTextures value. In the programmable pipeline, this indicates the number of texture registers supported by pixel shaders on this particular piece of hardware, and the number of texture declaration instructions that can be present.

VertexProcessingCaps

Vertex processing capabilities. For a given physical device, this capability might vary across Direct3DDevice objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

D3DVTXPCAPS_DIRECTIONALLIGHTS

Device supports directional lights.

D3DVTXPCAPS_LOCALVIEWER

Device supports local viewer. D3DVTXPCAPS_MATERIALSOURCE7 Device supports selectable vertex color sources.

D3DVTXPCAPS_POSITIONALLIGHTS

Device supports positional lights (including point lights and spotlights).

D3DVTXPCAPS_TEXGEN

Device can generate texture coordinates.

D3DVTXPCAPS_TWEENING

Device supports vertex tweening.

D3DVTXPCAPS_NO_VSDT_UBYTE4

Device does not support the D3DVSDT_UBYTE4 vertex declaration type.

MaxActiveLights

Maximum number of lights that can be active simultaneously. For a given physical device, this capability might vary across Direct3DDevice objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

MaxUserClipPlanes

Maximum number of user-defined clipping planes supported. This member can range from 0 through D3DMAXUSERCLIPPLANES. For a given physical device, this capability may vary across Direct3DDevice objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

MaxVertexBlendMatrices

Maximum number of matrices that this device can apply when performing multimatrix vertex blending. For a given physical device, this capability may vary across Direct3DDevice objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

MaxVertexBlendMatrixIndex

DWORD value that specifies the maximum matrix index that can be indexed into using the per-vertex indices. The number of matrices is **MaxVertexBlendMatrixIndex** + 1, which is the size of the matrix palette. If normals are present in the vertex data that needs to be blended for lighting, then the number of matrices is half the number specified by this capability flag. If **MaxVertexBlendMatrixIndex** is set to zero, the driver does not support indexed vertex blending. If this value is not zero then the valid range of indices is zero through **MaxVertexBlendMatrixIndex**.

A zero value for **MaxVertexBlendMatrixIndex** indicates that the driver does not support indexed matrices.

When software vertex processing is used, 256 matrices could be used for

indexed vertex blending, with or without normal blending.

For a given physical device, this capability may vary across Direct3DDevice objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

MaxPointSize

Maximum size of a point primitive. If set to 1.0f then device does not support point size control. The range is greater than or equal to 1.0f.

MaxPrimitiveCount

Maximum number of primitives, or vertices for each DrawPrimitive call. Note that when Direct3D is working with a DirectX 6.0 or DirectX 7.0 driver, this field is set to 0xFFFF. This means that not only the number of primitives but also the number of vertices is limited by this value.

MaxVertexIndex

Maximum size of indices supported for hardware vertex processing. It is possible to create 32-bit index buffers by specifying D3DFMT_INDEX32; however, you will not be able to render with the index buffer unless this value is greater than 0x0000FFFF.

MaxStreams

Maximum number of concurrent data streams for

IDirect3DDevice8::SetStreamSource. The valid range is 1–16. Note that if this value is 0, the driver is not a DirectX 8 driver.

MaxStreamStride

Maximum stride for **IDirect3DDevice8::SetStreamSource**.

VertexShaderVersion

Vertex shader version, indicating the level of vertex shader supported by the device. Only vertex shaders with version numbers equal to or less than this will succeed in calls to **IDirect3DDevice8::CreateVertexShader**. The level of shader is specified to **CreateVertexShader** as the first token in the vertex shader token stream.

- DirectX 7.0 functionality is 0
- DirectX 8.x functionality is 01

The main version number is encoded in the second byte. The low byte contains a sub-version number.

MaxVertexShaderConst

The number of vertex shader <u>input registers</u> that are reserved for constants.

PixelShaderVersion

Two numbers that represent the pixel shader main and sub versions. For more information about the versions supported in DirectX 8.*x*, see the pixel shader <u>version instruction</u>.

MaxPixelShaderValue

Maximum value of pixel shader arithmetic component. This value indicates the internal range of values supported for pixel color blending operations. Within the range that they report to, implementations must allow data to pass through pixel processing unmodified (unclamped). Normally, the value of this member is an absolute value. For example, a 1.0 indicates that the range is -1.0 to 1, and an 8.0 indicates that the range is -8.0 to 8.0. The value must be ≥ 1.0 for any hardware that supports pixel shaders.

Remarks

The **MaxTextureBlendStages** and **MaxSimultaneousTextures** members might seem very similar, but they contain different information. The **MaxTextureBlendStages** member contains the total number of texture-blending stages supported by the current device, and the **MaxSimultaneousTextures** member describes how many of those stages can have textures bound to them by using the **IDirect3DDevice8::SetTexture** method.

When the driver fills this structure, it can set values for execute-buffer capabilities, even when the interface being used to retrieve the capabilities (such as **IDirect3DDevice8**) does not support execute buffers.

For systems that set the D3DCAPS2_NO2DDURING3DSCENE capability flag, performance problems may occur if you use a texture and then modify it during a scene. This is true on all hardware, but it is more severe on hardware that exposes the D3DCAPS2_NO2DDURING3DSCENE capability. If D3DCAPS2_NO2DDURING3DSCENE is present on the hardware, application-based texture management should ensure that no texture used in the current **BeginScene** and **EndScene** block is evicted unless absolutely necessary. In the case of extremely high texture usage within a scene, the results are undefined. This occurs when you modify a texture that you have used in the scene and there is no spare texture memory available. For such systems, the contents of the z buffer become invalid at **EndScene**. Applications should not call **IDirect3DDevice8::CopyRects** to or from the back buffer on this type of hardware inside a **BeginScene** and **EndScene** pair. In addition, applications

should not try to access the z buffer if the D3DPRASTERCAPS_ZBUFFERLESSHSR capability flag is set. Finally, applications should not lock the back buffer or the z buffer inside a **BeginScene** and **EndScene** pair.

The following flags concerning mipmapped textures are not supported in DirectX 8.x.

- D3DPTFILTERCAPS_NEAREST
- D3DPTFILTERCAPS_LINEAR
- D3DPTFILTERCAPS_MIPNEAREST
- D3DPTFILTERCAPS_MIPLINEAR
- D3DPTFILTERCAPS_LINEARMIPNEAREST
- D3DPTFILTERCAPS_LINEARMIPLINEAR

Requirements

Header: Declared in D3d8caps.h.

See Also

IDirect3D8::GetDeviceCaps, IDirect3DDevice8::GetDeviceCaps

Microsoft DirectX 8.1 (C++)

IDirect3DDevice8::GetDeviceCaps

Retrieves the capabilities of the rendering device.

```
HRESULT GetDeviceCaps(
   D3DCAPS8* pCaps
);
```

Parameters

pCaps

[out] Pointer to a **D3DCAPS8** structure, describing the returned device.

Return Values

If the method succeeds, the return value is <u>D3D_OK</u>.

If the method fails, the return value can be <u>D3DERR_INVALIDCALL</u>.

Remarks

GetDeviceCaps retrieves the software vertex pipeline capabilities when the device is being used in software vertex processing mode. Software vertex processing mode is selected when a device has been created with D3DCREATE_SOFTWAREVERTEXPROCESSING, or when a device has been created with D3DCREATE_MIXEDVERTEXPROCESSING and D3DRS_SOFTWAREVERTEXPROCESSING is set to TRUE.

Requirements

Header: Declared in D3d8.h. **Import Library:** Use D3d8.lib.

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

Instruction Modifiers

Use instruction modifiers to change the output of an instruction. For instance, use them to multiply or divide the result by a factor of two, or to clamp the result between zero and one. Instruction modifiers are applied after the instruction executes but before writing the result to the destination register.

A list of the modifiers is shown below.

Modifier	Description	Suntay	Version					
withile		Syntax	1.0	1.1	1.2	? 1. 3	3 1.4	
_x2	Multiply by 2	register_x2	Х	Х	Х	Х	Х	
_x4	Multiply by 4	register_x4	Х	Х	Х	Х	Х	
_x8	Multiply by 8	register_x8					Х	
_d2	Divide by 2	register_d2	Х	Х	Х	Х	Х	
_d4	Divide by 4	register_d4					Х	
_d8	Divide by 8	register_d8					Х	
_sat	Saturate (clamp from 0 and 1)) register_sat	X	Х	Х	Х	Х	

- The multiply modifier multiplies the register data by a power of two after it is read. This is the same as a shift left.
- The divide modifier divides the register data by a power of two after it is read. This is the same as a shift right.
- The saturate modifier clamps the range of register values from zero to one.

Instruction modifiers can be used on arithmetic instructions. They may not be used on texture address instructions.

Examples

Multiply modifier

This example loads the destination register (dest) with the sum of the two colors in the source operands (src0 and src1) and multiplies the result by two.

```
add_x2 dest, src0, src1
```

This example combines two instruction modifiers. First, two colors in the source operands (src0 and src1) are added. The result is then multiplied by two, and clamped between 0.0 to 1.0 for each component. The result is saved in the destination register.

add_x2_sat dest, src0, src1

Divide modifier

This example loads the destination register (dest) with the sum of the two colors in the source operands (src0 and src1) and divides the result by two.

add_d2 dest, src0, src1

Saturate modifier

For arithmetic instructions, the saturation modifier clamps the result of this instruction into the range 0.0 to 1.0 for each component. The following example shows how to use this instruction modifier.

dp3_sat r0, t0_bx2, v0_bx2 ; t0 is bump, v0 is light direction

This operation occurs after any multiply or divide instruction modifier. _sat is most often used to clamp dot product results. However, it also enables consistent emulation of multipass methods where the frame buffer is always in the range 0 to 1, and of Microsoft® DirectX® 6.0 and 7.0 multitexture syntax, in which saturation is defined to occur at every stage.

This example loads the destination register (dest) with the sum of the two colors in the source operands (src0 and src1), and clamps the result into the range 0.0 to 1.0 for each component.

```
add_sat dest, src0, src1
```

This example combines two instruction modifiers. First, two colors in the source operands (src0 and src1) are added. The result is multiplied by two and clamped between 0.0 to 1.0 for each component. The result is saved in the destination register.

add_x2_sat dest, src0, src1

Microsoft Directx 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

Source Register Modifiers

Use source register modifiers to change the value read from a register before an instruction executes. The contents of a source register are left unchanged. Modifiers are useful for adjusting the range of register data in preparation for the instruction. A set of modifiers called *selectors* copies or replicates the data from a single channel (r,g,b,a) into the other channels.

Version 1.4 shaders have modifier functionality specific to shader instructions <u>texld</u> and <u>texcrd</u>. These modifiers affect version 1.4 texture registers and are detailed in <u>Texture Register Modifiers</u>.

Source register modifiers	s Syntax	Version						
Source register modifiers		1.0	1.1	1.2	1.3	1.4		
<u>Bias</u>	register_bias	Х	Х	Х	Х	Х		
Invert	1 - register	Х	Х	Х	Х	Х		
<u>Negate</u>	- register	Х	Х	Х	Х	Х		
<u>Scale×2</u>	register_x2					Х		
Signed Scaling	register_bx2	Х	Х	Х	Х	Х		

Source register modifiers can be used only on arithmetic instructions. They cannot be used on texture address instructions. The exception to this is the <u>signed scale</u> modifier (_bx2). For version 1.0 and 1.1, signed scale can be used on the source argument of any texm* instruction. For version 1.2 or 1.3, signed scale can be used on the source argument of any texture address instruction.

Some modifier specific restrictions:

- Negate can be combined with either the bias, signed scaling, or scale×2 modifier. When combined, negate is executed last.
- Invert cannot be combined with any other modifier.
- Invert, negate, bias, signed scaling, and scale×2 can be combined with any of the selectors.
- Source register modifiers should not be used on constant registers because they will cause undefined results. For version 1.4, modifiers on constants are not allowed and will fail validation.

Microsoft DirectX 8.1 (**pixel shader versions 1.1, 1.2, 1.3, 1.4**)

Source Register Selectors

This modifier replicates a single channel of a source-register argument to all channels.

source register.channel

Source register selectors	s Syntax 1.(Version 0 1.1 1.2 1.3			
Red replicate	source register.r				Х	
Green replicate	source register.g				Х	
Blue replicate	source register.b	Х	Х	Х	Х	
Alpha replicate	source register.a X	Χ	Х	Х	Х	

- Red replicate. Replicates the red channel to all channels.
- Green replicate. Replicates the green channel to all channels.
- Blue replicate. Replicates the blue channel to all channels.
- Alpha replicate. Replicates the alpha channel to to all channels.

Register

Source register. For more about register types, see <u>Registers</u>.

In version 1.1, 1.2, and 1.3, blue replicate is available only on the source register of arithmetic instruction, which uses an alpha <u>destination register write mask</u>.

Remarks

Source register selectors are applied before any <u>source register modifiers</u> and before the instruction executes.

Copying the contents of a channel into one or more other channels is commonly referred to as "swizzling."

These selectors are valid on source registers for arithmetic instructions. The four selectors operate on different channels.

An alternate syntax for the r,g,b,a channels is x,y,z,w.

Source selectors and source modifiers may be combined freely. In this example, register r0 uses the invert, bias, and signed scaling modifier, as well as the green selector. The contents of the source register are unaffected; the modifier modifies only the data read.

-r0_bx2.g

To understand the order of the execution of these modifiers and selectors, see <u>Order of Operations</u>.

This operator can be used in conjunction with the <u>Invert</u> or <u>Negate</u> operators.

Alpha replicate functionality is analogous to the D3DTA_ALPHAREPLICATE flag in the Microsoft® DirectX® 6.0 and 7.0 multitexture syntax.

Example

The examples below illustrate each of the four selectors.

```
// Replicate the red color channel to the all channels before
// doing the multiply.
mul r0, r0, r1.r // the result is r1.rgba = r1.r
// Replicate the green color channel to the all channels before
// doing the multiply.
mul r0, r0, r1.g
                   // the result is r1.rgba = r1.g
// Replicate the blue color channel to the all channels before
// doing the multiply.
mul r0, r0, r1.b
                 // the result is r1.rgba = r1.b
// For ps 1.1, 1.2, 1.3, the blue replicate example
// would require a destination write mask.
mul r0.a, r0, r1.b
// alpha replicate
mul r0, r0, r1.a ; Replicate the alpha color channel to all chann
```

Microsoft DirectX 8.1 (**pixel shader versions 1.0, 1.1, 1.2, 1.3, 1.4**)

Destination Register Write Masks

destination register.writemask

Write masks control which channels (red, green, blue, alpha) are updated in the destination register.

Register

Destination register. For more about register types, see <u>Registers</u>.

Remarks

The following destination write masks are available.

Write Mask	Suntax	Vei			Ve	rsion	
wille widsk	Syntax		1.0 1.1 1.2 1.3			}	1.4
red, green, blue, alpha	destination register.rgba	Х	Х	Х	Х	Х	
none	destination register	Х	Х	Х	Х	Х	
color (red, green, blue)	destination register.rgb	Х	Х	Х	Х	Х	
alpha	destination register.a	Х	Х	Х	Х	Х	
red	destination register.r					Х	
green	destination register.g					Х	
blue	destination register.b					Х	
arbitrary	destination register.rgba					See n	ote below.

Note The arbitrary mask allows any set of channels to be combined to produce a mask. The channels must be listed in the order r, g, b, a—for example, register.rba, which updates the red, blue, and alpha channels of the destination. The arbitrary mask is available only in version 1.4.

If no destination write mask is specified, the destination write mask defaults to the rgba case. In other words, all channels in the destination register are updated.

An alternate syntax for the r,g,b,a channels is x,y,z,w.

For versions 1.0 to 1.3, the dp3 arithmetic instruction can use only the .rgb or .rgba output write masks.

Destination register write masks are supported for arithmetic operations only. They cannot be used on texture addressing instructions, with the exception of the version 1.4 instructions, <u>texcrd</u> and <u>texld</u>.

Examples

default

The default is to write all four color channels.

// All four color channels can be written by explicitly listing them mul r0.rgba, t0, v0 $\,$

// Or, the default mask can be used to write all four channels. mul r0, t0, v0 $\,$

alpha write mask

The alpha write mask is used to control writing to the alpha channel.

// The alpha write mask is also referred to as the scalar write mask
// because it uses the scalar pipeline.
add r0.a, t1, v1

So this instruction effectively puts the sum of the alpha component of t1 and the alpha component of v1 into r0.a.

color write mask

The color write mask is used to control writing to the color channels.

// The color write mask is also referred to as the vector write mask
// because it uses the vector pipeline.
mul r0.rgb, t0, v0

arbitrary write mask

For version 1.4, destination write masks can be used in any combination as long as the masks are ordered r,g,b,a.

// This example updates the red, blue, and alpha channels. mov r0.rba, r1 $\,$

co-issued instructions

A co-issued instruction allows two potentially different instructions to be issued simultaneously. This is accomplished by executing the instructions in the alpha pipeline and the RGB pipeline.

```
// For example, the default example shown above:
mul r0, t0, v0
// is also equivalent to the following co-issued instruction
  mul r0.rgb, t0, v0
+ mul r0.a, t0, v0
```

The advantage of pairing instructions this way is that it allows different operations to be performed in the vector and scalar pipeline in parallel. Microsoft Directx 8.1 (**pixel shader version 1.4**)

Texture Register Modifiers for texld and texcrd

Two pixel shader version 1.4 texture address instructions, <u>texld</u> and <u>texcrd</u>, have custom syntax. These instructions support their own set of source register modifiers, source register selectors, and destination-register write masks, as shown below.

Source Register Modifiers for texld and texcrd

These modifiers provide projective divide functionality by dividing the x and y values by either the z or w values.

	Source register modifiers	Description	Syntax	Version
_d	Z	Divide x,y components by	source register_dz	1.4
_d	b	Ζ	source register_db	1.4
_d	W	Divide x,y components by	source register_dw	1.4
_d	a	W	source register_da	1.4

Register

Source register. For more about register types, see <u>Registers</u>.

Remarks

The _dz or _db modifier does the following:

x' = x/z (x' = 1.0 if z == 0) y' = y/z (y' = 1.0 if z == 0) z' is undefined w' is undefined The _dw or _da modifier does the following:

x' = x/w (x' = 1.0 if w == 0)
y' = y/w (y' = 1.0 if w == 0)
z' is undefined
w' is undefined

Note For pixel shader version 1.4, the D3DTTFF_PROJECTED flag under D3DTSS_TEXTURETRANSFORMFLAGS is ignored because a projective divide is accomplished by the source register modifiers listed previously.

Source Register Selectors for texld and texcrd

Selectors replicate the contents of one channel into other channels. Replicating the contents of a channel to one or more other channels is commonly referred to as "swizzling."

Source register selectors	Description	Syntax	Version
xyz	Maps x,y,z data to channels	source register.xyz	1.4
rgb	x,y,z,z	source register.rgb	1.4
xyw	Maps x,y,w data to channels x,y,w,w	source register.xyw	1.4
rga		source register.rga	1.4

Register

Source register. For more about register types, see <u>Registers</u>.

Remarks

The **texld** and **texcrd** instructions never read more than the first three components. So, these selectors provide the option of taking the third component from either the third or the fourth component of the source register.

Destination Register Write Masks for texld and texcrd

Write masks control which channels (red, green, blue, alpha) are updated in the destination register.

Destination register write masks	Description	Syntax	Used by	Version
xyzw		destination register.xyzw	texld only	1.4
rgba	Updates the x,y,z,w channels	destination register.rgba	texld only	1.4
none		destination register	texld only	1.4
xyz	Updates the x,y,z channels	destination register.xyz	texcrd only	1.4
rgb		destination register.rgb	texcrd only	1.4
xy	Updates the x,y	destination register.xy	texcrd only	1.4
rg	channels	destination register.rg	texcrd only	1.4

Register

Destination register. For more about register types, see <u>Registers</u>.

D3DXMESH

Flags used to specify creation options for a mesh.

enum _D3DXMESH {		
D3DXMESH_32BIT	=	0x001,
D3DXMESH_DONOTCLIP	=	0x002,
D3DXMESH_POINTS	=	0x004,
D3DXMESH_RTPATCHES	=	0x008,
D3DXMESH_NPATCHES	=	0x4000,
D3DXMESH_VB_SYSTEMMEM	=	0x010,
D3DXMESH_VB_MANAGED	=	0x020,
D3DXMESH_VB_WRITEONLY	=	0x040,
D3DXMESH_VB_DYNAMIC	=	0x080,
D3DXMESH_VB_SOFTWAREPROCESSING	=	0x8000,
D3DXMESH_IB_SYSTEMMEM	=	0x100,
D3DXMESH_IB_MANAGED	=	0x200,
D3DXMESH_IB_WRITEONLY	=	0x400,
D3DXMESH_IB_DYNAMIC	=	0x800,
D3DXMESH_IB_SOFTWAREPROCESSING	=	0×10000,
D3DXMESH_VB_SHARE	_	0x1000,
D3DXMESH_VB_SHARE		0×1000,
DSDAMESH_0SENWONE I	-	0,2000,
D3DXMESH_SYSTEMMEM	=	0x110,
D3DXMESH_MANAGED	=	0x220,
D3DXMESH_WRITEONLY	=	0x440,
D3DXMESH_DYNAMIC	=	0x880,
D3DXMESH_SOFTWAREPROCESSING	=	0x18000
};		

};

Constants

D3DXMESH_32BIT

The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to (2^32)-1 faces and vertices.

D3DXMESH_DONOTCLIP

Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_POINTS

Use the D3DUSAGE_POINTS usage flag for vertex and index buffers. D3DXMESH_RTPATCHES

Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers. D3DXMESH_NPATCHES

Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-patch enhancement using Microsoft® Direct3D®.

D3DXMESH_VB_SYSTEMMEM

Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers. D3DXMESH_VB_MANAGED

Use the D3DPOOL_MANAGED memory class for vertex buffers. D3DXMESH_VB_WRITEONLY

Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers. D3DXMESH_VB_DYNAMIC

Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers. D3DXMESH VB SOFTWAREPROCESSING

Use the D3DUSAGE_SOFTWAREPROCESSING for flag for vertex buffers.

D3DXMESH_IB_SYSTEMMEM

Use the D3DPOOL_SYSTEMMEM memory class for index buffers. D3DXMESH_IB_MANAGED

Use the D3DPOOL_MANAGED memory class for index buffers. D3DXMESH_IB_WRITEONLY

Use the D3DUSAGE_WRITEONLY usage flag for index buffers. D3DXMESH_IB_DYNAMIC

Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_SOFTWAREPROCESSING

Use the D3DUSAGE_SOFTWAREPROCESSING usage flag for index buffers.

D3DXMESH_VB_SHARE

Forces the cloned meshes to share vertex buffers.

D3DXMESH_USEHWONLY

Use hardware processing only. This flag should be specified only for a hardware processing device. On a mixed-mode device, this flag will cause the system to either use hardware only, or if the hardware is not capable, it will approximate using the software capabilities.

D3DXMESH_SYSTEMMEM

Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_MANAGED

Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED. D3DXMESH_WRITEONLY Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY. D3DXMESH_DYNAMIC Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC. D3DXMESH_SOFTWAREPROCESSING Equivalent to specifying both D3DXMESH_VB_SOFTWAREPROCESSING and D3DXMESH_VB_SOFTWAREPROCESSING and

Requirements

Header: Declared in D3dx8mesh.h.

ID3DXMesh

Applications use the methods of the **ID3DXMesh** interface to manipulate mesh objects.

To obtain the **ID3DXMesh** interface, call either the **D3DXCreateMesh** or **D3DXCreateMeshFVF** function. The methods of the **ID3DXMesh** interface can be organized into the following groups.

Locking	LockAttributeBuffer		
Locking	<u>UnlockAttributeBuffer</u>		
Ontimization	<u>Optimize</u>		
Optimization	<u>OptimizeInplace</u>		

Remarks

This interface inherits additional functionality from the **ID3DXBaseMesh** interface.

This interface, like all COM interfaces, inherits additional functionality from the **<u>IUnknown Interface</u>**.

The **LPD3DXMESH** type is defined as a pointer to the **ID3DXMesh** interface, as shown below.

typedef struct ID3DXMesh *LPD3DXMESH;

Requirements

Header: Declared in D3dx8mesh.h. **Import Library:** Use D3dx8.lib.

See Also

Mesh Functions

PALETTEENTRY

Specifies the color and usage of an entry in a logical palette.

```
typedef struct tagPALETTEENTRY {
  BYTE peRed;
  BYTE peGreen;
  BYTE peBlue;
  BYTE peFlags;
} PALETTEENTRY;
```

Members

peRed

The red intensity value for the palette entry.

peGreen

The green intensity value for the palette entry.

peBlue

The blue intensity value for the palette entry.

peFlags

The alpha intensity value for the palette entry. Note that as of Microsoft® DirectX® 8.0, this member is treated differently than documented in the Microsoft® Platform Software Development Kit (SDK).

Requirements

Header: Declared in Wingdi.h; include Windows.h.

D3DX_NORMALMAP Flags

Control the generation of normal maps.

```
#define D3DX_NORMALMAP_MIRROR_U (1 << 16)
#define D3DX_NORMALMAP_MIRROR_V (2 << 16)
#define D3DX_NORMALMAP_MIRROR (3 << 16)
#define D3DX_NORMALMAP_INVERTSIGN (8 << 16)
#define D3DX_NORMALMAP_COMPUTE_OCCLUSION (16 << 16)</pre>
```

Constants

```
D3DX_NORMALMAP_MIRROR_U
```

Indicates that pixels off the edge of the texture on the U-axis should be mirrored, not wrapped.

```
D3DX_NORMALMAP_MIRROR_V
```

Indicates that pixels off the edge of the texture on the V-axis should be mirrored, not wrapped.

D3DX_NORMALMAP_MIRROR Same as specifying D3DX_NORMALMAP_MIRROR_U | D3DX_NORMALMAP_MIRROR_V.

D3DX_NORMALMAP_INVERTSIGN

Inverts the direction of each normal.

```
D3DX_NORMALMAP_COMPUTE_OCCLUSION
```

Computes the per-pixel occlusion term and encodes it into the alpha. An alpha of 1 means that the pixel is not obscured in any way, and an alpha of 0 means that the pixel is completely obscured.

Requirements

Header: Declared in D3d8tex.h.

See Also

D3DXComputeNormalMap

D3DX_CHANNEL Flags

The following flags are used to specify which channels in a texture to operate on.

#define	D3DX_CHANNEL_RED	1
#define	D3DX_CHANNEL_BLUE	2
#define	D3DX_CHANNEL_GREEN	4
#define	D3DX_CHANNEL_ALPHA	8
#define	D3DX_CHANNEL_LUMINANCE	16

Constants

D3DX_CHANNEL_RED Indicates the red channel should be used.

D3DX_CHANNEL_BLUE

Indicates the blue channel should be used.

D3DX_CHANNEL_GREEN

Indicates the green channel should be used.

D3DX_CHANNEL_ALPHA

Indicates the alpha channel should be used.

D3DX_CHANNEL_LUMINANCE

Indicates the luminances of the red, green, and blue channels should be used.

Requirements

Header: Declared in D3d8tex.h.

IUnknown Interface

All COM objects support an interface called **IUnknown**. This interface provides Microsoft® DirectX® with control of the object's lifetime and the ability to retrieve other interfaces implemented by the object. **IUnknown** has three methods.

- <u>AddRef</u> increments the object's reference count by 1 when an interface or another application binds itself to the object.
- **QueryInterface** queries the object about the features that it supports by requesting pointers to a specific interface.
- **<u>Release</u>** decrements the object's reference count by 1. When the count reaches 0, the object is deallocated.

The **AddRef** and **Release** methods maintain an object's reference count. For example, if you create a Microsoft Direct3D® object, the object's reference count is set to 1. Every time a function returns a pointer to an interface for that object, the function must call **AddRef** through that pointer to increment the reference count. Match each **AddRef** call with a call to **Release**. Before the pointer can be destroyed, you must call **Release** through that pointer. After an object's reference count reaches 0, the object is destroyed, and all interfaces to it become invalid.

The **QueryInterface** method determines whether an object supports a specific interface. If an object supports an interface, **QueryInterface** returns a pointer to that interface. You then can use the methods of that interface to communicate with the object. If **QueryInterface** successfully returns a pointer to an interface, it implicitly calls **AddRef** to increment the reference count, so your application must call **Release** to decrement the reference count before destroying the pointer to the interface.

Requirements

Windows NT/2000/XP: Requires Windows NT 3.1 or later. Windows 98/Me: Requires Windows 98 or later.

Header: Declared in Unknwn.h.

D3DXIMAGE_FILEFORMAT

Describes the supported image file formats.

```
typedef enum _D3DXIMAGE_FILEFORMAT
{
    D3DXIFF_BMP
                        = 0,
    D3DXIFF_JPG
D3DXIFF_TGA
                        = 1,
                        = 2,
    D3DXIFF_PNG
                       = 3,
    D3DXIFF_DDS
                       = 4,
                      = 5,
    D3DXIFF_PPM
    D3DXIFF_DIB
                       = 6,
    D3DXIFF FORCE DWORD = 0x7ffffff
```

} D3DXIMAGE_FILEFORMAT;

Constants

D3DXIFF_BMP Microsoft® Windows® bitmap file format. D3DXIFF_JPG Joint Photographic Experts Group compressed file. D3DXIFF_TGA Truevision Targa image file. D3DXIFF_PNG Portable Network Graphics file format. D3DXIFF_DDS Microsoft DirectDraw® surface file format. D3DXIFF_PPM Portable pixmap file format. D3DXIFF_DIB Windows bitmap file format. D3DXIFF_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Header: Declared in D3dx8tex.h.

LPD3DXFILL2D

Function type used by the texture fill functions.

Parameters

pOut

[out] Pointer to a vector, which the function uses to return its result. X, Y, Z, and W will be mapped to R, G, B, and A respectively.

pTexCoord

[in] Pointer to a vector containing the coordinates of the texel currently being evaluated. Texture coordinate components for texture and volume textures range from 0 to 1. Texture coordinate components for cube textures range from -1 to 1.

pTexelSize

[in] Pointer to a vector containing the dimensions of the current texel. *pData*

[in] Pointer to user data.

Requirements

Header: Declared in D3dx8tex.h.

See Also

D3DXFillTexture

LPD3DXFILL3D

Function type used by the texture fill functions.

Parameters

pOut

[out] Pointer to a vector, which the function uses to return its result. X, Y, Z, and W will be mapped to R, G, B, and A respectively.

pTexCoord

[in] Pointer to a vector containing the coordinates of the texel currently being evaluated. Texture coordinate components for texture and volume textures range from 0 to 1. Texture coordinate components for cube textures range from -1 to 1.

pTexelSize

[in] Pointer to a vector containing the dimensions of the current texel. *pData*

[in] Pointer to user data.

Requirements

Header: Declared in D3dx8tex.h.

See Also

D3DXFillCubeTexture, D3DXFillVolumeTexture

Sample Framework

The Microsoft® DirectX® 8.1 and Microsoft® Direct3D® Software Development Kit (SDK) graphics sample framework is an evolution from the DirectX 7.0 graphics sample framework. The SDK samples are installed by default in \Dxsdk\Samples\Multimedia. The folders of interest are Common and Direct3D. The sample framework is contained in the Common folder and the Direct3D samples based on the graphics framework are contained in the Direct3D folder.

The graphics framework consists of five source modules.

- **d3dapp.cpp** exposes the application interface used for samples. Of particular interest is class CD3DApplication.
- **d3dfile.cpp** furnishes .x file support, to enable samples to load .x files. Of particular interest are classes CD3Dmesh and CD3DFrame.
- **d3dfont.cpp** furnishes basic font output support, to enable things like statistics views. Of particular interest is class CD3DFont.
- **d3dutil.cpp** provides generally useful 3 dimensional functions, such as material, light, and texture helper functions.
- **dxutil.cpp** provides generally useful DirectX functions, such as media, registry, and timer helper functions.

Corresponding header files are located in the Common\Include folder.

Each sample implements a subclass of CD3DApplication, which is typically named CMyD3DApplication, and set of overridables that are shown below.

```
// Overridable functions for the 3 dimensional scene created by the
virtual HRESULT ConfirmDevice(D3DCAPS8*,DWORD,D3DFORMAT)
                                                             { return
virtual HRESULT OneTimeSceneInit()
                                                              return
virtual HRESULT InitDeviceObjects()
                                                             { return
virtual HRESULT RestoreDeviceObjects()
                                                              return
virtual HRESULT FrameMove()
                                                             { return
virtual HRESULT Render()
                                                             { return
virtual HRESULT InvalidateDeviceObjects()
                                                             { return
virtual HRESULT DeleteDeviceObjects()
                                                             { return
virtual HRESULT FinalCleanup()
                                                             { return
```

The prototypes for these methods are contained in d3dapp.h in the CD3Dapplication class. The samples create a new Direct3D applicatio

those methods that are needed by the application.

Derived Class Example

This example uses a subset of the overrideable methods. The class CMyD3DApplication contains the following methods. Each of these methods is explained below.

```
class CMyD3DApplication : public CD3DApplication
{
    public:
        CMyD3DApplication();

protected:
    HRESULT ConfirmDevice( D3DCAPS8*, DWORD, D3DFORMAT );
    HRESULT DeleteRestoreDeviceObjects();
    HRESULT RestoreDeviceObjects();
    HRESULT FrameMove();
    HRESULT Render();

private:
    LPDIRECT3DVERTEXBUFFER8 m_pVB; // Vertex buffer to
};
```

Constructor

The constructor initializes the window title, enables depth buffering and initializes the vertex buffer.

```
CMyD3DApplication::CMyD3DApplication()
{
    m_strWindowTitle = _T("D3D Example"); // title bar string
    m_bUseDepthBuffer = TRUE; // enable depth buff
    m_pVB = NULL; // initialize
}
```

The window title is a wide character string that is visible in the title bar or the window class when the application is invoked. It is optional.

The base class contains a member variable for enabling depth buffering. The default value for this boolean variable is FALSE, which disables depth buffering.

The window title is a wide character string that is visible in the title bar or the window class when the application is invoked. It is optional.

ConfirmDeviceObjects

DeleteDeviceObjects

DeleteDeviceObjects is called when the application is exiting, or if the device is being changed. You use this method to delete device dependent objects, such as the vertex buffer.

```
HRESULT CVShader1::DeleteDeviceObjects()
{
    m_pQuadVB->Release();
    m_pQuadVB = NULL;
    return S_OK;
}
```

RestoreDeviceObjects

This method is called when the application needs to restore device memory objects and device states. This is required after a DirectX device is created or resized. This method does most of the work of creating objects and initializing render states.

```
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    // Create the vertex buffer. Allocate enough memory (from the de
        // to hold the custom vertices. Specify the flexible vertex
                data it contains.
        11
    if( FAILED( m_pd3dDevice->CreateVertexBuffer( NUM_VERTS*sizeof(C
                          0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &
    {
        return E_FAIL;
    }
    // Fill the vertex buffer. First, lock the vertex buffer to get
        // vertices. This mechanism is required because vertex buffe
        // memory. Then use memcpy to do a fast data copy.
    VOID* pVertices;
    if( FAILED( m_pVB->Lock( 0, sizeof(g_Vertices),
     (BYTE**)&pVertices;, 0 ) ))
        return E_FAIL;
   memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
   m pVB->Unlock();
    // Set the projection matrix. The size of the back buffer comes
        // class
```

```
D3DXMATRIX matProj;
FLOAT fAspect = m_d3dsdBackBuffer.Width /
                (FLOAT)m_d3dsdBackBuffer.Height;
D3DXMatrixPerspectiveFovLH( &matProj;, D3DX_PI/4, fAspect,
                             1.0f, 100.0f );
m_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj; );
// Set up the view matrix. A view matrix can be defined from an
// point, a look at point and an up direction vector. In this ex
// the eye position is (0,1,-4) the look at point is (0,0,0) and
// up vector is (0,1,0.
D3DXMATRIX matView;
D3DXMatrixLookAtLH( &matView;, &D3DXVECTOR3;( 0.0f, 1.0f, -4.0f )
                              &D3DXVECTOR3;( 0.0f, 0.0f, 0.0f ),
                              &D3DXVECTOR3;( 0.0f, 1.0f, 0.0f )
m_pd3dDevice->SetTransform( D3DTS_VIEW, &matView; );
// Set up default texture states
// Set up render states ( this is only one example renderstate )
m pd3dDevice->SetRenderState( D3DRS CULLMODE,
                                                  D3DCULL NONE )
return S_OK;
```

This method creates the vertex buffer and copies the vertex data into it. It creates the view and projection matrices, which define the camera orientation to the object in the vertex buffer. Texture stage states can be set in this method although none are present in this example. Render states that are not likely to change are set. These determine how the scene renders.

FrameMove

}

This method contains actions that happen every frame such as animation. In this example, it adds a y axis rotation to the world transform.

```
HRESULT CMyD3DApplication::FrameMove()
{
    // For our world matrix, just rotate the object about the y-axis
    D3DXMATRIX matWorld;
    D3DXMatrixRotationY( &matWorld;, ::TimeGetTime()/150.0f );
    m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld; );
    return S_OK;
}
```

The Windows method ::TimeGetTime() is used to return the current time. Once it is divided by 150, this generates a incremental angle to rotate the object.

Render

This method is called when it is time to render the output. This function clears the view port and render the scene. It also renders state changes.

The Render method first clears the viewport using Clear. Then, within the BeginScne/EndScene pair it uses SetStreamSource to inform the runtime that it uses vertex buffer m_pVB with a stride of the size of the custom vertex type. Then, it informs the runtime that it uses a flexible vertex format (FVF) shader, the simplest type. Finally it invokes DrawPrimitive to render the quad.

Other functions

DeleteDeviceObjects is called when the application exits, or if the device changes. You use this method to delete device dependent objects.

ConfirmDevice checks the device for some minimum set of capabilities. It is called during the device initialization.

InvalidateDeviceObjects is called when the device dependent objects might be

removed. Device dependent objects such as vertex buffers are usually added to this method.

OneTimeSceneInit is provided for code that needs to run during the initial application startup.

DirectXDev for graphics, networking, and input at http://DISCUSS.MICROSOFT.COM/archives/DIRECTXDEV.html.

Application Wizard

The Microsoft® DirectX® AppWizard creates a small C++ template application that can integrate the common DirectX components—Microsoft Direct3D®, Microsoft DirectInput®, Microsoft DirectMusic®, Microsoft DirectSound®, and Microsoft DirectPlay®. It provides basic, easy-to-build-upon functionally and demonstrates the use of each component.

Along with the template, AppWizard includes the same C++ classes found in the DirectX software development kit (SDK) common directory. These C++ classes perform minimal wrapping of DirectX to get you up and running quickly. All the DirectX SDK samples make use of these classes. For example, the Direct3D samples use D3dapp.*, the DirectMusic samples use Dmutil.*, the DirectSound samples use Dsutil.*, and the DirectPlay samples use Netconnect.*.

The AppWizard target audience is amateur C++ game developers, 3-D hobbyists, and audio hobbyists (who do not need 3-D).

Running AppWizard

AppWizard is launched from Microsoft Visual Studio® 6.0 when you create a new project. It is one of the project types available when you choose **File**, **New**, then the **Projects** tab, as shown in the following example.

When you enter a project name and click **OK**, the wizard presents a series of dialog boxes to allow you to configure the DirectX services that the application will need. The number of dialog boxes generated by the wizard depend on which DirectX technologies you select (Direct3D, DirectInput, and DirectPlay each generate their own dialog boxes). The files included in the project depend on the DirectX technologies selected, as well as the Microsoft Windows® technologies selected—Microsoft Win32®, Microsoft Foundation Classes (MFC), and Graphics Device Interface (GDI). You select these technologies on the first page of the wizard, which is shown in the following example.

The wizard has three steps: Step 1 is the opening page, step 2 is the Direct3D page, and step 3 is the DirectInput page. The **DirectMusic** and **DirectSound** options do not generate additional pages but the **DirectPlay Peer-to-Peer** option will generate an additional page.

The wizard uses one of four base templates, based on the options you select.

- 1. For an application that is not MFC-based and uses Direct3D, choose **Single document window** and **Direct3D**. This is the standard case and is similar to the application used in the <u>dolphin sample</u>.
- 2. For an application that is MFC-based and uses Direct3D, choose **MFC dialog based** and **Direct3D**. The application will be similar to the <u>MFC fog sample</u>.
- 3. For a Windows application that uses GDI but not MFC, choose **Single document window**. This application can use **CreateWindow** and basic **GameLoop** during idle time.

4. For an application that uses GDI and is MFC-based, choose **MFC dialog based**. The application will be similar to the MFC AppWizard dialog template.

The **Direct3D**, **DirectInput**, and **DirectPlay** check boxes each add another page to the wizard. These pages are described in the following sections.

Direct3D Page

The following Direct3D page is shown immediately after the first page if you selected the **Direct3D** check box in step 1.

The Direct3D page is step 2 because step 1 was the first page of the wizard.

The Direct3D exclusive options are:

- 1. **Blank**. No Direct3D object is created.
- 2. **Triangle**. Creates a simple object with two back-to-back triangles.
- 3. **Teapot**. Creates a complex object using <u>D3DXCreateTeapot</u>.

If **Direct3D fonts** is checked, the project includes D3dfont.cpp and D3dfont.h. Otherwise, the project uses <u>D3DXFont</u> to display 3-D fonts.

If **3D meshes** is checked, the project includes D3dfile.cpp and D3dfile.h.

DirectInput Page

The following **DirectInput** page is shown immediately after the Direct3D page if you selected the **DirectInput** check box in step 1.

The DirectInput page is step 3. The options on this page are:

- **DirectInput action mapping**. Uses <u>Action Mapping</u> to gather input data.
- **DirectInput keyboard device**. Uses a simple <u>keyboard device</u> to gather input data.

If you checked **DirectInput** in step 1, the project will include Diutil.cpp and Diutil.cpp files. The DirectX SDK sample, Root\Samples\bin\Donut3D.exe, uses these class files.

The application will use either Action Mapper or a keyboard device object to record key state. If neither is checked, DirectInput will use WM_KEYDOWN messages to record key state.

DirectMusic and DirectSound Check Boxes

These check boxes appear on page 1 of the wizard. They do no add pages to the wizard.

If **DirectMusic** is checked, the project will include Dmutil.cpp and Dmutil.h. The DirectMusic samples use these files.

If **DirectSound** is checked, the project will include Dsutil.cpp and Dsutil.h. The DirectSound samples use these files to load and play sounds.

If both **DirectMusic** and **DirectSound** are checked, DirectMusic is used to load and play the sounds, but Dsutil.* is still included in the project.

If either **DirectMusic** or **DirectSound** is checked, Bounce.wav is included in the project. Pressing the A key will play the sound.

If neither **DirectMusic** nor **DirectSound** is checked, bounce.wav is not included and the A key is not recorded, nor is the Help string displayed.

DirectPlay Page

The following DirectPlay page is shown immediately after the DirectInput page if you selected the **DirectPlay Peer-to-Peer** check box in step 1.

The DirectPlay page is step 4, assuming that step 1 was the first page of the wizard, step 2 was the Direct3D page, and step 3 was the DirectInput page.

If **DirectVoice** is checked, the project will include Netvoice.cpp and Netvoice.h. The DirectPlay Voice samples use these files. The project will integrate the NetConnect DirectPlay connection dialog boxes into the application. When the **NetConnect** dialog boxes finish, an active DirectPlay connection or failure results. If successful, the arrow key state is passed between all players using DirectPlay.

If DirectInput Action Mapper is used, DirectPlay sends the axis data across the network; otherwise it sends the state of the four arrow keys.

DirectX Graphics C/C++ Tutorials

The tutorials in this section show how to use Microsoft® Direct3D® and Direct3DX in a C/C++ application for common tasks. The tasks are divided into required steps. In some cases, steps are organized into substeps for clarity.

The following tutorials are provided.

- Tutorial 1: Creating a Device
- Tutorial 2: Rendering Vertices
- Tutorial 3: Using Matrices
- <u>Tutorial 4: Creating and Using Lights</u>
- <u>Tutorial 5: Using Texture Maps</u>
- <u>Tutorial 6: Using Meshes</u>

Note The sample code in these tutorials is from source projects whose location is provided in each tutorial.

The sample files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you need to add the *vtable* and *this* pointers to the interface methods.

Some comments in the included sample code might differ from the source files in the Microsoft Platform Software Development Kit (SDK). Changes are made for brevity only and are limited to comments to avoid changing the behavior of the sample code.

See Also

DirectX Graphics C/C++ Samples

Direct3D C/C++ Reference

This section contains reference information for the API elements provided by Microsoft® Direct3D®. Reference material is divided into the following categories.

- <u>Interfaces</u>
- <u>Functions</u>
- <u>Macros</u>
- <u>Vertex Shader Declarator Macros</u>
- <u>Structures</u>
- Enumerated Types
- <u>Other Types</u>
- <u>Texture Argument Flags</u>
- Flexible Vertex Format Flags
- <u>Return Values</u>

Direct3DX C/C++ Reference

This section contains reference information for the API elements provided by the Direct3DX utility library. Reference material is divided into the following categories.

- <u>Interfaces</u>
- <u>Functions</u>
- <u>Macros</u>
- <u>Structures</u>
- Enumerated Types and Flags
- <u>C++ Specific Features</u>
- <u>Return Values</u>

X File C/C++ Reference

This section contains reference information for the application programming interface (API) elements you use to work with Microsoft® DirectX® .x files.

- <u>Interfaces</u>
- <u>Functions</u>
- <u>Structures</u>
- <u>Return Values</u>
- <u>X File Format Reference</u>

Getting Started with Direct3D

This section provides a brief introduction to the three-dimensional (3-D) graphics functionality in the Microsoft® Direct3D® application programmer interface (API). Here you will find an overview of the *graphics pipeline* and tutorials to help you get basic Direct3D functionality up and running quickly.

- <u>Direct3D Architecture</u>
- <u>DirectX Graphics C/C++ Tutorials</u>

Using Direct3D

This section explains the operation of the Microsoft® Direct3D® fixed function pipeline. The pipeline consists of several building blocks. These blocks are detailed in the following sections.

- <u>Vertex Data</u>
- <u>Transforms</u>
- <u>Viewports and Clipping</u>
- Lights and Materials
- <u>Textures</u>
- <u>Rendering</u>
- <u>About Devices</u>

Programmable Pipeline

Using shaders and effects, developers can now program the pipeline. These topics are covered in the following sections.

- <u>Vertex Shaders</u>
- <u>Pixel Shaders</u>
- <u>Effects</u>

Advanced Topics

Microsoft® Direct3D® provides a powerful set of tools that you can use to increase the realistic appearance of a 3-D scene. This section presents information on common special effects that can be produced with Direct3D, but the range of possible effects is not limited to those presented here. The discussion in this section is organized into the following topics.

- <u>Antialiasing</u>
- Bump Mapping
- Environment Mapping
- <u>Geometry Blending</u>
- Indexed Vertex Blending
- Matrix Stacks
- <u>Stencil Buffer Techniques</u>
- <u>Vertex Tweening</u>
- **Object Geometry**

Direct3D Appendix

This section contains additional material that covers topics such as file formats and tips for performance improvements.

• DDS File Format

This section explains the DDS file format in detail.

• Device States

This section explains device states, which are used to set rendering and texturing attributes.

• <u>Programming Tips</u>

These tips are derived from lessons learned from programming topics such as troubleshooting a program, implementing multithreading, or optimizing code for performance.

• <u>X Files</u>

This section explains X files in depth, including their architecture, the file format, and some samples of file loading and saving.

• Mesh View Help

This section outlines the functionality that is available in the mesh view executable. This handy executable can be used to experiment with meshes by applying different mesh utility operations. This application is part of the SDK install.

Microsoft DirectX 8.1 (version 1.0, 1.1)

Create a Vertex Shader

This example creates a vertex shader that applies a constant color to an object. The example will show the contents of the shader file as well as the code required in the application to set up the Microsoft® Direct3D® pipeline for the shader data.

To create a vertex shader

- Step 1: Declare the vertex data
- Step 2: Design the shader functionality
- Step 3: Check for vertex shader support
- Step 4: Declare the shader registers
- Step 5: Create the shader
- Step 6: Render the output pixels

If you already know how to build and run Direct3D samples, you can cut and paste code from this example into your existing application.

Step 1: Declare the vertex data

This example uses a quadrilateral that is made up of two triangles. The vertex data will contain (x,y,z) position and a diffuse color. The **D3DFVF_CUSTOMVERTEX** macro is defined to match the vertex data. The vertex data is declared in a global array of vertices (g_Vertices). The four vertices are centered about the origin, and each vertex is given a different diffuse color.

```
// Declare vertex data structure.
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD diffuseColor;
};
// Declare custom FVF macro.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)
// Declare the vertex position and diffuse color data.
CUSTOMVERTEX g_Vertices[]=
{
```

```
// x y z diffuse color
{ -1.0f, -1.0f, 0.0f, 0xffff0000 }, // red - bottom right
{ +1.0f, -1.0f, 0.0f, 0xff00ff00 }, // green - bottom left
{ +1.0f, +1.0f, 0.0f, 0xff0000ff }, // blue - top left
{ -1.0f, +1.0f, 0.0f, 0xffffff00 }, // red and green = yellow
};
```

Step 2: Design the shader functionality

This shader applies a constant color to each vertex. The shader file VertexShader.vsh follows:

vs.1.0 // version instruction m4x4 oPos, v0, c0 // transform vertices by view/projection matrix mov oD0, c4 // load constant color

This file contains three instructions.

The first instruction in a shader file must be the shader version declaration. This instruction (vs) declares the vertex shader version, which is 1.0 in this case.

The second instruction (m4x4) transforms the object vertices using the view/projection matrix. The matrix is loaded into four constant registers c0, c1, c2, c3 (as shown below).

The third instruction (mov) copies the constant color in register c4 to the output diffuse color register oD0. This results in coloring the output vertices.

Step 3: Check for vertex shader support

The device capability can be queried for vertex shader support before using a vertex shader.

```
D3DCAPS8 caps;
m_pd3dDevice->GetDeviceCaps(∩); // initialize m_pd3dDevice bef
if( D3DSHADER_VERSION_MAJOR( caps.VertexShaderVersion ) < 1 )
return E_FAIL;
```

The caps structure returns the functional capabilities of the hardware after **GetDeviceCaps** is called. Use the **D3DSHADER_VERSION_MAJOR** macro to test the supported version number. If the version number is less than 1.0, this call will fail. The result of this method should be used to control whether or not

vertex shaders are invoked by an application.

Step 4: Declare the shader registers

The shader is created by declaring the shader registers and compiling the shader file. Once created, Direct3D returns a shader handle, which is an integer number that is used to identify the shader.

```
// Create the shader declaration.
DWORD dwDecl[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
    D3DVSD_REG( D3DVSDE_DIFFUSE, D3DVSDT_D3DCOLOR ),
    D3DVSD_END()
};
```

The vertex declaration declares the mapping between input data streams and vertex buffers. Multiple streams can be declared in the shader declaration, up to the number specified in the **MaxStreams** cap. Vertex buffers are associated with input streams using **SetStreamSource** as illustrated in step 5.

Step 5: Create the shader

The shader is assembled and created next.

```
// Create the vertex shader.
TCHAR strPath[512]; // location of the s
LPD3DXBUFFER pCode; // assembled shader
DXUtil_FindMediaFile( strPath, _T("VertexShader.vsh") );
D3DXAssembleShaderFromFile( strPath, 0, NULL, &pCode;, NULL ); // a
m_pd3dDevice->CreateVertexShader( dwDecl, (DWORD*)pCode->GetBufferPo
pCode->Release();
```

Once the shader file is located, **D3DXAssembleShaderFromFile** reads and validates the shader instructions. **CreateVertexShader** takes the shader declaration and the assembled instructions and creates the shader. It returns the shader handle, which is used to render the output.

CreateVertexShader can be used to create programmable or fixed function shaders. Programmable shaders are generated if a pointer to a shader declaration is passed as the second parameter. Otherwise, a fixed function vertex shader is

generated if NULL is passed as the second parameter.

Step 6: Render the output pixels

Here is an example of the code that could be used in the render loop to render the object, using the vertex shader. The render loop updates the vertex shader constants as a result of changes in the 3-D scene and draws the output vertices with a call to DrawPrimitive.

```
// Turn lighting off. This is included for clarity but is not requir
m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
// Update vertex shader constants from view projection matrix data.
D3DXMATRIX mat, matView, matProj;
D3DXMatrixMultiply( &mat;, &matView;, &matProj; );
D3DXMatrixTranspose( &mat;, &mat; );
m_pd3dDevice->SetVertexShaderConstant( 0, &mat;, 4 );
// Declare and define the constant vertex color.
float color[4] = {0,1,0,0};
m_pd3dDevice->SetVertexShaderConstant( 4, &color;, 1 );
// Render the output.
m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetVertexShader( m_hVertexShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

Lighting is turned off just to make it clear that the vertex color is from the shader only. This statement is optional in this example.

The view and projection matrixes contain camera position and orientation data. Getting updated data and updating the shader constant registers is included in the render loop because the scene might change between rendered frames.

As usual, DrawPrimitive renders the output data using the vertex data provided from **SetStreamSource**. SetVertexShader is called to tell Direct3D to use the vertex shader. The result of the vertex shader is shown in the following image. It shows the constant color on the plane object.

Microsoft DirectX 8.1 (shader versions 1.0, 1.1)

Shader2 - Apply vertex colors

This example applies the vertex color from the vertex data to the object. The vertex data contains position data as well as diffuse color data. This is reflected in the vertex declaration and the fixed vertex function macro. These are shown below.

```
struct CUSTOMVERTEX POS COLOR
{
    float
DWORD
               x, y, z;
                 diffuseColor;
};
#define D3DFVF_CUSTOMVERTEX_POS_COLOR (D3DFVF_XYZ|D3DFVF_DIFFUSE)
// Create vertex data with position and texture coordinates.
CUSTOMVERTEX_POS_COLOR g_Vertices[]=
{
    // x
                      Z
                              diffuse
              У
                                                  // - bottom right -
// - bottom left -
// - top left - blu
// - top right - wh
                                          },
},
},
},
    { -1.0f, 0.25f, 0.0f, 0xffff0000,
    { 0.0f, 0.25f, 0.0f, 0xff00ff00,
    { 0.0f, 1.25f, 0.0f, 0xff0000ff,
    { -1.0f, 1.25f, 0.0f, 0xffffffff,
};
```

The vertex shader declaration needs to reflect the position and color data also.

```
DWORD dwDecl2[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(0, D3DVSDT_FLOAT3), // Register 0 will contain t
    D3DVSD_REG(1, D3DVSDT_D3DCOLOR ), // Register 1 will contain t
    D3DVSD_END()
};
```

One way for the shader to get the transformation matrix is from a constant register. This is done by calling SetVertexShaderConstant.

```
D3DXMATRIX mat;
D3DXMatrixMultiply( &mat;, &m;_matView, &m;_matProj );
D3DXMatrixTranspose( &mat;, &mat; );
hr = m_pd3dDevice->SetVertexShaderConstant( 1, &mat;, 4 );
if(FAILED(hr)) return hr;
```

This declaration declares one stream of data, which contains the position and the

color data. The color data is assigned to vertex register 7.

Lastly, here is the shader file.

```
vs.1.0 ; version instruction
m4x4 oPos, v0, c0 ; transform vertices by view/projection matrix
mov oD0, v1 ; load color from register 7 to diffuse color
```

It contains three instructions. The first is always the version instruction. The second instruction transforms the vertices. The third instruction moves the color in the vertex register to the output diffuse color register. The result is output vertices using the vertex color data.

The resulting output looks like the following:

```
// Here is an example of the class used to produce this vertex shade
// This example is for illustration only. It has not been optimized
// CVShader.h
// Use vertex color to color the object.
CUSTOMVERTEX_POS_COLOR g_VerticesVS[]=
{
                 // x
                                                                                                                     diffuse
                                                                                          Z
                                                                 V
                 // $\lambda y = 2 $\lambda v = 0.000 $\lambda v = 0.0000 $\lambda v = 0.0000000$\lambda v = 0.00000000$\lambda v = 0.0000000$\lambda v = 0.000000$\lambda v = 0.00000$\lambda v = 0.00000$\lambda v = 0.00000$\lambda v = 0.0000$\lambda v = 0.000$\lambda v =
                                                                                                                                                                                                 // - bottom left -
// - top left - blu
// - top right - re
};
class CVShader
{
public:
                 CVShader();
                 HRESULT ConfirmDevice( D3DCAPS8* pCaps, DWORD dwBehavior, D3DFOR
                 HRESULT DeleteDeviceObjects();
                 HRESULT Render();
                 HRESULT RestoreDeviceObjects(LPDIRECT3DDEVICE8 l_pd3dDevice);
                 HRESULT InitMatrices();
                 HRESULT UpdateVertexShaderConstants();
```

private:

```
LPDIRECT3DVERTEXBUFFER8
                                 m_pQuadVB;
    LPDIRECT3DDEVICE8
                                 m_pd3dDevice;
    DWORD
                                                 m_hVertexShader;
    D3DXMATRIX
                                         m_matView;
    D3DXMATRIX
                                         m_matProj;
};
CVShader::CVShader()
{
    m_pQuadVB
                                 = NULL;
    m_pd3dDevice
                                 = NULL;
    m_hVertexShader
                                 = 0;
}
HRESULT CVShader::ConfirmDevice( D3DCAPS8* pCaps, DWORD dwBehavior,
                                           D3DFORMAT Format )
{
    if( D3DSHADER_VERSION_MAJOR( pCaps->VertexShaderVersion ) < 1 )
        return E_FAIL;
    return S_OK;
}
HRESULT CVShader::DeleteDeviceObjects()
{
    SAFE_RELEASE(m_pQuadVB);
    HRESULT hr;
    if(m_hVertexShader > 0)
    {
        hr = m_pd3dDevice->DeleteVertexShader(m_hVertexShader);
        if(FAILED(hr))
        {
            ::MessageBox(NULL,"","DeleteVertexShader failed",MB_OK);
            return E_FAIL;
        }
        m_hVertexShader = 0;
    }
    // local to this class
    m_pd3dDevice = NULL;
    return S_OK;
}
HRESULT CVShader::InitMatrices()
```

```
{
    HRESULT hr;
    D3DXVECTOR3 from( 0.0f, 0.0f, 3.0f );
    D3DXVECTOR3 at( 0.0f, 0.0f, 0.0f );
    D3DXVECTOR3 up( 0.0f, 1.0f, 0.0f );
    D3DXMATRIX matWorld;
    D3DXMatrixIdentity( &matWorld; );
    hr = m pd3dDevice->SetTransform( D3DTS WORLD, &matWorld; );
    D3DXMatrixIdentity( &m;_matView );
    D3DXMatrixLookAtLH( &m;_matView, &from;, &at;, &up; );
    m_pd3dDevice->SetTransform( D3DTS_VIEW, &m;_matView );
    D3DXMatrixIdentity( &m;_matProj );
    D3DXMatrixPerspectiveFovLH( &m;_matProj, D3DX_PI/4, 1.0f, 0.5f,
    m pd3dDevice->SetTransform( D3DTS PROJECTION, &m; matProj );
    return S_OK;
}
HRESULT CVShader::Render()
{
    if(m_pQuadVB)
    {
        HRESULT hr;
        hr = m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
        UpdateVertexShaderConstants();
        hr = m_pd3dDevice->SetStreamSource( 0, m_pQuadVB,
                    sizeof(CUSTOMVERTEX_POS_COLOR) );
        hr = m_pd3dDevice->SetVertexShader( m_hVertexShader );
        hr = m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
        }
    return S_OK;
}
HRESULT CVShader::RestoreDeviceObjects(LPDIRECT3DDEVICE8 1_pd3dDevic
{
    HRESULT hr;
```

```
if(l_pd3dDevice == NULL)
{
    ::MessageBox(NULL,"","Invalid D3D8 Device ptr",MB_OK);
    return E_FAIL;
```

```
}
else
    m_pd3dDevice = l_pd3dDevice;
InitMatrices();
// Create guad Vertex Buffer.
hr = m_pd3dDevice->CreateVertexBuffer( 4*sizeof(CUSTOMVERTEX_POS
                                     D3DUSAGE_WRITEONLY,
                                     D3DFVF CUSTOMVERTEX POS COLO
                                     D3DPOOL_DEFAULT,
                                     &m;_pQuadVB );
if( FAILED(hr) ) return hr;
// Fill the quad VB.
CUSTOMVERTEX_POS_COLOR1* pVertices = NULL;
hr = m_pQuadVB->Lock( 0, 4*sizeof(CUSTOMVERTEX_POS_COLOR), (BYTE
if(FAILED(hr)) return hr;
for( DWORD i=0; i<4; i++ )</pre>
    pVertices[i] = g_VerticesVS2[i];
m_pQuadVB->Unlock();
// Create the vertex shader.
             strVertexShaderPath[512];
TCHAR
LPD3DXBUFFER pCode;
DWORD dwDecl2[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(0, D3DVSDT_FLOAT3),
        D3DVSD_REG(1, D3DVSDT_D3DCOLOR),
    D3DVSD_END()
};
// Find the vertex shader file.
DXUtil_FindMediaFile( strVertexShaderPath, _T("VShader.vsh") );
// Assemble the vertex shader from the file.
if( FAILED( hr = D3DXAssembleShaderFromFile( strVertexShaderPath
                                              0, NULL, &pCode;, N
    return hr;
// Create the vertex shader.
if(SUCCEEDED(hr = m_pd3dDevice->CreateVertexShader( dwDecl2,
                                     (DWORD*)pCode->GetBufferPoin
    pCode->Release();
```

```
return hr;
}
HRESULT CVShader::UpdateVertexShaderConstants()
{
    HRESULT hr;
    D3DXMATRIX mat;
    D3DXMatrixMultiply( &mat;, &m;_matView, &m;_matProj );
    D3DXMatrixTranspose( &mat;, &mat; );
    hr = m_pd3dDevice->SetVertexShaderConstant( 1, &mat;, 4 );
    return hr;
}
```

Microsoft DirectX 8.1 (version 1.0, 1.1)

Shader3 - Apply a texture map

This example applies a texture map to the object.

The vertex data contains object position data as well as texture position or uv data. This causes changes to the vertex declaration structure and the fixed vertex function macro. The vertex data is also shown below.

```
struct CUSTOMVERTEX_POS_TEX1
{
                        x, y, z; // object position data
tu1, tv1; // texture position data
      float
                                                               // texture position data
      float
};
#define D3DFVF_CUSTOMVERTEX_POS_TEX1 (D3DFVF_XYZ|D3DFVF_TEX1)
CUSTOMVERTEX POS TEX1 q Vertices[]=
{
                       У
      // x
                                            u1
                                                      ν1
                                 Ζ
      // \ \ y 2 \ 0.1 \ y 1
{ -0.75f, -0.5f, 0.0f, 0.0f, 0.0f }, // - bottom right
{ 0.25f, -0.5f, 0.0f, 1.0f, 0.0f }, // - bottom left
{ 0.25f, 0.5f, 0.0f, 1.0f, -1.0f }, // - top left
{ -0.75f, 0.5f, 0.0f, 0.0f, -1.0f }, // - top right
};
```

D3DUtil_CreateTexture(m_pd3dDevice, _T("earth.bmp"), &m;_pTexture0,

The texture image must be loaded. In this case, the file "earth.bmp" contains a 2-D texture map of the earth and will be used to color the object.

The vertex shader declaration needs to reflect the object position and texture position data.

```
DWORD dwDecl2[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
    D3DVSD_REG(8, D3DVSDT_FLOAT2 ),
    D3DVSD_END()
};
```

This declaration declares one stream of data that contains the object position and the texture position data. The texture position data is assigned to vertex register 8.

The rendering code tells Microsoft® Direct3D® where to find the data stream and the shader, and sets up texture stages because a texture map is being applied.

```
m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX_POS
m_pd3dDevice->SetVertexShader( m_hVertexShader );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODU
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTU
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFU
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
m_pd3dDevice->SetTexture( 0, NULL );
```

Because there is a single texture, the texture stage states need to be set for texture state 0. These methods tell Direct3D that the texel values will be used to provide diffuse color for the object vertices. In other words, a 2-D texture map will be applied like a decal.

Here is the shader file.

```
vs.1.0 ; version instruction
m4x4 oPos, v0, c0 ; transform vertices by view/projection matr
mov oT0, v8 ; move texture color to output textu
```

The shader file contains three instructions. The first is always the version instruction. The second instruction transforms the vertices. The third instruction moves the texture colors from register v8 to the output diffuse color register. That results in a texture mapped object, which is shown below.

Microsoft DirectX 8.1 (version 1.0, 1.1)

Shader4 - Apply a texture map with lighting

This example uses a vertex shader to apply a texture map and add lighting to the scene. The object used is a sphere. The sample code applies a texture map of the earth to the sphere and applies diffuse lighting to simulate night and day.

The code sample adds to the Shader3 example by adding lighting to a texture mapped object. Refer to Shader3 for information about loading the texture map and setting up the texture stage states.

There is a detailed explanation of the sample code framework at <u>Sample</u> <u>Framework</u>. You can cut and paste the sample code into the sample framework to quickly get a working sample.

Create Vertex Shader

The vertex data has been modified from the Shader3 sample to include vertex normals. For lighting to appear, the object must have vertex normals. The data structure for the vertex data and the flexible vertex format (FVF) macro used to declare the data type are shown below.

A shader declaration defines the input vertex registers and the data associated with them. This matches the FVF macro used to create the vertex buffer data later.

```
DWORD dwDecl[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG(0, D3DVSDT_FLOAT3), // position
    D3DVSD_REG(4, D3DVSDT_FLOAT3), // normal
    D3DVSD_REG(7, D3DVSDT_D3DCOLOR), // diffuse color
```

```
D3DVSD_REG(8, D3DVSDT_FLOAT2), // texture coordinate
D3DVSD_END()
};
```

This declares one stream of data, which contains the vertex position, normal, diffuse color, and texture coordinates. The integer in each line is the register number that will contain the data. So, the texture coordinate data will be in register v8, for instance.

Next, create the shader file. You can create a shader from an ASCII text string or load it from a shader file that contains the same instructions. This example uses a shader file.

You always enter the version instruction first. The last instruction moves the texture data to the output register oT0. After you write the shader instructions, you can use them to create the shader.

```
// Now that the file exists, use it to create a shader.
TCHAR strVertexShaderPath[512];
LPD3DXBUFFER pCode;
DXUtil_FindMediaFile( strVertexShaderPath, _T("VShader3.vsh") );
D3DXAssembleShaderFromFile( strVertexShaderPath, 0, NULL, &pCode, NU
m_pd3dDevice->CreateVertexShader( dwDecl, (DWORD*)pCode->GetBufferPo
pCode->Release();
```

After the file is located, Direct3D creates the vertex shader and returns a shader handle and the assembled shader code. This sample uses a shader file, which is one method for creating a shader. The other method is to create an ASCII text string with the shader instructions in it. For an example, see <u>Programmable</u> <u>Shaders for DirectX 8.0</u>.

Vertex Shader Constants

You can define vertex shader constants outside of the shader file as shown in the following example. Here, you use constants to provide the shader with a view/projection matrix, a diffuse light color, RGBA, and the light direction vector.

```
float constants[4] = {0, 0.5f, 1.0f, 2.0f};
m_pd3dDevice->SetVertexShaderConstant( 0, &constants;, 1 );
D3DXMATRIX mat;
D3DXMatrixMultiply( &mat;, &m;_matView, &m;_matProj );
D3DXMatrixTranspose( &mat;, &mat; );
m_pd3dDevice->SetVertexShaderConstant( 4, &mat;, 4 );
float color[4] = {1,1,1,1};
m_pd3dDevice->SetVertexShaderConstant( 8, &color;, 1 );
float lightDir[4] = {-1,0,1,0}; // fatter slice
m_pd3dDevice->SetVertexShaderConstant( 12, &lightDir;, 1 );
```

You can also define constants inside a shader using the <u>def</u> instruction

Render

After you write the shader instructions, connect the vertex data to the correct vertex registers and initialize the constants, you should render the output. Rendering code tells Direct3D where to find the vertex buffer data stream and provides Direct3D with the shader handle. Because you use a texture, you must set texture stages to tell Direct3D how to use the texture data.

```
// Identify the vertex buffer data source.
m_pd3dDevice->SetStreamSource(0, m_pVB, sizeof(CUSTOMVERTEX_POS_NORM
// Identify the shader.
m_pd3dDevice->SetVertexShader( m_hVertexShader );
// Define the texture stage(s) and set the texture(s) used
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODU
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTU
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFU
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
// Draw the object.
```

```
DWORD dwNumSphereVerts = 2 * m_dwNumSphereRings*(m_dwNumSphereSegmen
m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, dwNumSphereVerts
```

// Set the texture stage to NULL after the render commands. Leaving

```
// out will cause a memory leak.
m_pd3dDevice->SetTexture( 0, NULL );
```

The output image follows:

With the texture map applied, the sphere looks like the planet Earth. The lighting creates a bright to dark gradient on the face of the globe.

Additional Code

There is additional code required to support this example. Shown below are a few of the other methods for creating the sphere object, loading the texture, and checking for the correct version of pixel shader support.

```
// Confirm that the hardware supports version 1 shader instructions.
if( D3DSHADER_VERSION_MAJOR( pCaps->VertexShaderVersion ) < 1 )
    return E_FAIL;
// Load texture map for the sphere object.
LPDIRECT3DTEXTURE8 m_pTexture0;
D3DUtil_CreateTexture( m_pd3dDevice, _T("earth.bmp"), &m;_pTexture0,
// Create the sphere object.
DWORD dwNumSphereVerts = 2*m_dwNumSphereRings*(m_dwNumSphereSegments
// once for the top, once for the bottom vertices
// Get the World-View(WV) matrix set.
D3DXMATRIX matWorld, matView, matWorldView;
m_pd3dDevice->GetTransform( D3DTS_WORLD, &matWorld; );
m_pd3dDevice->GetTransform( D3DTS_VIEW, &matView; );
D3DXMatrixMultiply( &matWorldView;, &matWorld;, &matView; );
m_pd3dDevice->CreateVertexBuffer(
    dwNumSphereVerts*sizeof(CUSTOMVERTEX_POS_NORM_COLOR1_TEX1),
    D3DUSAGE_WRITEONLY,
    D3DFVF_CUSTOMVERTEX_POS_NORM_COLOR1_TEX1,
    D3DPOOL_DEFAULT,
    &m;_pVB)
                );
CUSTOMVERTEX_POS_NORM_COLOR1_TEX1* pVertices;
HRESULT hr;
hr = m_pVB->Lock(0, dwNumSphereVerts*sizeof(pVertices),
                                         (BYTE**)&pVertices;, 0);
if(SUCCEEDED(hr))
```

```
{
    FLOAT fDeltaRingAngle = ( D3DX PI / m dwNumSphereRings );
    FLOAT fDeltaSegAngle = ( 2.0f * D3DX_PI / m_dwNumSphereSegments
    // Generate the group of rings for the sphere.
    for( DWORD ring = 0; ring < m_dwNumSphereRings; ring++ )</pre>
    {
        FLOAT r0 = sinf( (ring+0) * fDeltaRingAngle );
FLOAT r1 = sinf( (ring+1) * fDeltaRingAngle );
        FLOAT y0 = cosf( (ring+0) * fDeltaRingAngle );
        FLOAT y1 = cosf( (ring+1) * fDeltaRingAngle );
        // Generate the group of segments for the current ring.
        for( DWORD seg = 0; seg < (m_dwNumSphereSegments+1); seg++ )</pre>
        {
             FLOAT x0 = r0 * sinf( seg * fDeltaSegAngle );
            FLOAT z0 = r0 * cosf( seg * fDeltaSegAngle );
             FLOAT x1 = r1 * sinf(seq * fDeltaSeqAngle);
             FLOAT z1 = r1 * cosf( seg * fDeltaSegAngle );
             // Add two vertices to the strip, which makes up the sph
             // (using the transformed normal to generate texture coo
             pVertices -> x = x0;
             pVertices ->y = y0;
             pVertices -> z = z0;
             pVertices ->nx = x0;
             pVertices -> ny = y0;
             pVertices -> nz = z0;
             pVertices->color = HIGH_WHITE_COLOR;
             pVertices->tu = -((FLOAT)seg)/m_dwNumSphereSegments;
             pVertices->tv = (ring+0)/(FLOAT)m_dwNumSphereRings;
             pVertices++;
             pVertices -> x = x1;
             pVertices -> y = y1;
             pVertices -> z = z1;
             pVertices ->nx = x1;
             pVertices -> ny = y1;
             pVertices -> nz = z1;
             pVertices->color = HIGH_WHITE_COLOR;
             pVertices->tu = -((FLOAT)seg)/m dwNumSphereSegments;
             pVertices->tv = (ring+1)/(FLOAT)m_dwNumSphereRings;
             pVertices++;
        }
        hr = m_pVB->Unlock();
    }
}
```

Microsoft DirectX 8.1 (C++)

Create a Pixel Shader

This example uses a pixel shader to apply a Gouraud interpolated diffuse color to a geometric plane. The example will show the contents of the shader file as well as the code required in the application to set up the Microsoft® Direct3D® pipeline for the shader data.

To create a pixel shader

- 1. <u>Check for pixel shader support</u>.
- 2. <u>Declare the vertex data</u>.
- 3. <u>Design the pixel shader</u>.
- 4. <u>Create the pixel shader</u>.
- 5. <u>Render the output pixels</u>.

If you already know how to build and run Direct3D samples, you should be able to cut and paste code from this example into your existing application.

Step 1: Check for pixel shader support

To check for pixel shader support, use the following code. This example checks for pixel shader version 1.1.

```
D3DCAPS8 caps;
m_pd3dDevice->GetDeviceCaps(∩); // init m_pd3dDevice before u
if( D3DPS_VERSION(1,1) != caps.PixelShaderVersion )
return E_FAIL;
```

The caps structure returns the functional capabilities of the pipeline. Use the D3DPS_VERSION macro to test for the supported shader version number. If the version number is less than 1.1, this call will fail. If the hardware does not support the shader version that is tested, the application will have to fall back to another rendering approach (perhaps a lower shader version is available).

Step 2: Declare the vertex data

This example uses a plane, which is made up of two triangles. The data structure for each vertex will contain position and diffuse color data. The

D3DFVF_CUSTOMVERTEX macro defines a data structure to match the vertex data. The actual vertex data is declared in a global array of vertices called g_Vertices[]. The four vertices are centered about the origin, and each vertex is given a different diffuse color.

```
// Declare vertex data structure.
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD diffuseColor;
};
// Declare custom FVF macro.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)
// Declare the vertex position and diffuse color data.
CUSTOMVERTEX q Vertices[]=
{
11
                          diffuse color
        Х
               У
                      Ζ
    { -1.0f, -1.0f, 0.0f, 0xffff0000 },
                                         // red - bottom left
    { +1.0f, -1.0f, 0.0f, 0xff00ff00 }, // green - bottom right
    { +1.0f, +1.0f, 0.0f, 0xff0000ff }, // blue - top right
    { -1.0f, +1.0f, 0.0f, 0xffffffff }, // white - top left
};
```

Step 3: Design the pixel shader

This shader moves the Gouraud interpolated diffuse color data to the output pixels. The shader file PixelShader.txt follows:

ps.1.0 // version instruction mov r0,v0 // Move the diffuse vertex color to the output registe

The first instruction in a pixel shader file declares the pixel shader version, which is 1.0.

The second instruction moves the contents of the color register (v0) into the output register (r0). The color register contains the vertex diffuse color because the vertex data is declared to contain the interpolated diffuse color in step 1. The output register determines the pixel color used by the render target (because there is no additional processing, such as fog, in this case).

Step 4: Create the pixel shader

The pixel shader is created from the pixel shader instructions. In this example, the instructions are contained in a separate file. The instructions could also be used in a text string.

TCHAR strPath[512]; // used to locate the shader fi LPD3DXBUFFER pCode; // points to the assembled shad DXUtil_FindMediaFile(strPath, _T("PixelShader.txt"));

This function is a helper function used by the <u>Sample Framework</u>. The sample framework is the foundation on which many of the samples are built.

```
D3DXAssembleShaderFromFile( strPath, 0, NULL, &pCode;, NULL ); // m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer(),
```

Once the shader is created, the handle m_hPixelShader is used to refer to it.

Step 5: Render the output pixels

Rendering the output pixels is very similar to using the fixed function pipeline sequence of calls except that the pixel shader handle is now used to set the shader.

```
// Turn lighting off for this example. It will not contribute to the
// The pixel color will be determined solely by interpolating the ve
m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
```

```
m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
m_pd3dDevice->SetPixelShader( m_hPixelShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

The source of the vertex data is set with <u>SetStreamSource</u>. In this example, <u>SetVertexShader</u> uses the same Flexible Vertex Format (FVF) macro used during vertex data declaration to tell Direct3D to do fixed function vertex processing. Vertex shaders and pixel shaders may be used together or separately. The fixed function pipeline can also be used instead of either pixel or vertex shaders. <u>SetPixelShader</u> tells Direct3D which pixel shader to use, <u>DrawPrimitive</u> tells Direct3D how to draw the plane.

The gouraud shaded pixels are shown in the following image.

<u>Pixel Shader Examples</u> contains examples that show how to apply texture maps and blend between textures and vertex colors.

Microsoft DirectX 8.1 (C++)

Texture Considerations

The pixel shader completely replaces the pixel-blending functionality specified by the Microsoft® DirectX® 6.0 and 7.0 multi-texturing application programming interfaces (APIs), specifically those operations defined by the D3DTSS_COLOROP, D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAOP, D3DTSS_ALPHAARG1, and D3DTSS_ALPHAARG2 texture stage states, and associated arguments and modifiers. When a procedural pixel shader is set, these states are ignored.

Pixel Shaders and Texture Stage States

When pixel shaders are in operation, the following texture stage states are still observed.

- <u>D3DTSS_ADDRESSU</u>
- <u>D3DTSS_ADDRESSV</u>
- <u>D3DTSS_ADDRESSW</u>
- <u>D3DTSS_BUMPENVMAT00</u>
- <u>D3DTSS_BUMPENVMAT01</u>
- <u>D3DTSS_BUMPENVMAT10</u>
- <u>D3DTSS_BUMPENVMAT11</u>
- <u>D3DTSS_BORDERCOLOR</u>
- <u>D3DTSS_MAGFILTER</u>
- <u>D3DTSS_MINFILTER</u>
- <u>D3DTSS_MIPFILTER</u>
- <u>D3DTSS_MIPMAPLODBIAS</u>
- <u>D3DTSS_MAXMIPLEVEL</u>
- <u>D3DTSS_MAXANISOTROPY</u>
- <u>D3DTSS_BUMPENVLSCALE</u>
- <u>D3DTSS_BUMPENVLOFFSET</u>
- <u>D3DTSS_TEXCOORDINDEX</u>
- <u>D3DTSS_TEXTURETRANSFORMFLAGS</u>

Because these texture stage states are not part of the pixel shader, they are not available at shader compile time so the driver can make no assumptions about them. For example, the driver cannot differentiate between bilinear and trilinear filtering at that time. The application is free to change these states without requiring the regeneration of the currently bound shader.

Pixel Shaders and Texture Sampling

Texture sampling and filtering operations are controlled by the standard texture stage states for minification, magnification, mip filtering, and the wrap addressing modes. For more information, see <u>Texture Stage States</u>. This information is not available to the driver at shader compile time, so shaders must be able to continue operation when this state changes. The application is responsible for setting textures of the correct type (image map, cube map, volume map, etc.) needed by the pixel shader. Setting a texture of the incorrect type will produce unexpected results.

Post-Shader Pixel Processing

Other pixel operations—such as fog blending, stencil operations, and render target blending—occur after execution of the shader. Render target blending syntax is updated to support new features as described in this topic.

Pixel Shader Inputs

Diffuse and specular colors are saturated (clamped) to the range 0 through 1 before use by the shader because this is the range of valid inputs to the shader.

Color values input to the pixel shader are assumed to be perspective correct, but this is not guaranteed in all hardware. Colors generated from texture coordinates by the address shader are always iterated in a perspective correct manner. However, they are also clamped to the range 0 to 1 during iteration.

Pixel Shader Outputs

The result emitted by the pixel shader is the contents of register r0. Whatever it contains when the shader completes processing is sent to the fog stage and render target blender.

Microsoft DirectX 8.1 (C++)

Confirming Pixel Shader Support

You can query members of **D3DCAPS8** to determine the level of support for operations involving pixel shaders. The following table lists the device capabilities related to programmable pixel processing in Microsoft® DirectX® 8.1.

Device capability	Description
MaxPixelShaderValue	Range of values that may be stored in registers is [-MaxPixelShaderValue, MaxPixelShaderValue].
MaxSimultaneousTextures	Number of texture stages that can be used in the programmable pixel shader.
PixelShaderVersion	Level of support for pixel shaders.

The **PixelShaderVersion** capability indicates the level of pixel shader supported. Only pixel shaders with version numbers equal to or less than this value will be successfully created. The major version number is encoded in the second byte of **PixelShaderVersion**. The low byte contains a minor version number. The pixel shader version is indicated by the first token in each shader.

Each implementation sets the **PixelShaderVersion** member to indicate the maximum pixel shader version that it can fully support. This implies that implementations should never fail the creation of a valid shader of the version less than or equal to the version reported by **PixelShaderVersion**.

Setting Pixel Shader Texture Inputs

The texture coordinate data is interpolated from the vertex texture coordinate data and is associated with a specific texture stage. The default association is a one-to-one mapping between texture stage number and texture coordinate declaration order. This means that the first set of texture coordinates defined in the vertex format are, by default, associated with texture stage 0.

Texture coordinates can be associated with any stage, using either of the following two techniques. When using a fixed function vertex shader, the texture stage state flag TSS_TEXCOORDINDEX can be used in IDirect3DDevice8::SetTextureStageState to associate coordinates with a stage. Otherwise, the texture coordinates are output by the vertex shader oTn registers when using a programmable vertex shader.

Setting the Pixel Shader Constant Registers

You can use the following methods to set and retrieve the values in the pixel shader constant registers.

- <u>IDirect3DDevice8::GetPixelShaderConstant</u>
- <u>IDirect3DDevice8::SetPixelShaderConstant</u>

In addition, you can use the **def** instruction to set the constant registers of a pixel shader, inside a pixel shader. This instruction must come before all other instructions except the version instruction.

Compiling and Creating a Pixel Shader

The Direct3DX utility library provides a set of functions to compile pixel shaders. The following functions are provided.

- D3DXAssembleShader
- <u>D3DXAssembleShaderFromFile</u>

The **IDirect3DDevice8::CreatePixelShader** create a pixel shader in DirectX 8.1 from a compiled shader declaration. The compiled shader declaration is obtained from **D3DXAssembleShader**.

A given shader might fail creation because of the restraints of the DirectX 8.1 hardware model.

Microsoft DirectX 8.1 (C++)

Pixel Shader Examples

The topic <u>Create a Pixel Shader</u> provides an example of how to use a pixel shader to apply a single diffuse color. The following are examples of other pixel shader functions. Each example builds on the previous example by adding a piece of additional pixel shader functionality.

- <u>Apply a Texture Map</u>
- Blend a Diffuse Vertex Color with a Texture
- Blend Two Textures Using a Constant Color

Apply a Texture Map

This example applies a texture map to a plane. The differences between this example and the previous example are as follows:

- The vertex data structure and the Flexible Vertex Format (FVF) macro include texture coordinates. The vertex data includes u,v data. The vertex data no longer needs diffuse color because the pixel colors will come from the texture map.
- The texture is linked to texture stage 0 with SetTexture. Because the previous example did not use a texture, there was no SetTexture method used.
- The shader uses the t0 texture register instead of the v0 diffuse color register.

The sample code follows:

```
// Define vertex data structure.
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    FLOAT u1, v1;
};
// Define corresponding FVF macro.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_TEX|D3DFVF_TEXCOC
// Create vertex data with position and texture coordinates.
static CUSTOMVERTEX g_Vertices[]=
{
    //
      Х
                           u1
                                 v1
               У
                     Ζ
    { -1.0f, -1.0f, 0.0f,
                           0, 1, \},
      1.0f, -1.0f, 0.0f, 1, 1, },
       1.0f, 1.0f, 0.0f,
                           1, 0, \},
    { -1.0f, 1.0f, 0.0f,
                           0, 0, },
    // v1 is flipped to meet the top down convention in Windows
    // the upper left texture coordinate is (0,0)
    // the lower right texture coordinate is (1,1).
};
```

```
// Create a texture. This file is in the DirectX 8.1 media from
TCHAR strPath[512];
DXUtil_FindMediaFile( strPath, _T("DX5_Logo.bmp"));
LPDIRECT3DTEXTURE8 m_pTexture0;
D3DUtil_CreateTexture( m_pd3dDevice, strPath, &m;_pTexture0, D3E
```

```
// Create the pixel shader.
DXUtil_FindMediaFile( strPShaderPath, _T("PixelShader2.txt") );
```

This function is a helper function used by the <u>Sample Framework</u>. The sample framework is the foundation on which many of the samples are built.

```
D3DXAssembleShaderFromFile( strPShaderPath, 0, NULL, &pCode;, NL m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer
```

```
// Load the texture and render the output pixels.
m_pd3dDevice->SetTexture( 0, m_pTexture0 ); // load
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

Contents of the file "PixelShader2.txt"// Applies a texture map to object vertices.ps.1.0// Version instruction must be first in ttex t0// Declare texture register t0, which wilmov r0, t0// Move the contents of the texture regis

The resulting image is shown in the following example.

Blend a Diffuse Vertex Color with a Texture

This example blends or modulates the colors in a texture map with the vertex colors. The differences between this example and the previous example are as follows:

- The vertex data structure, the FVF macro, and the vertex data include diffuse color.
- The shader file uses the multiply instruction (mul) to blend or modulate the texture colors with the vertex diffuse color.

The texture create and load code is the same. It is included here for completeness.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color1;
    FLOAT tu1, tv1;
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TE
static CUSTOMVERTEX g_Vertices[]=
{
    // x
                           diffuse
                     Ζ
                                       u1
                                             v1
               У
    { -1.0f, -1.0f, 0.0f, 0xffff0000, 0, 1, }, // red
     1.0f, -1.0f, 0.0f, 0xff00ff00, 1, 1, }, // green
    { 1.0f, 1.0f, 0.0f, 0xff0000ff, 1, 0, }, // blue
    { -1.0f, 1.0f, 0.0f, 0xffffffff, 0, 0, }, // white
    // v1 is flipped to meet the top down convention in Windows
    // the upper left texture coordinate is (0,0)
    // the lower right texture coordinate is (1,1).
};
// Create a texture. This file is in the DirectX 8.1 media from
TCHAR strPath[512];
DXUtil_FindMediaFile( strPath, _T("DX5_Logo.bmp"));
LPDIRECT3DTEXTURE8
                        m_pTexture0;
D3DUtil_CreateTexture( m_pd3dDevice, strPath, &m;_pTexture0, D3E
// Create the pixel shader.
DXUtil_FindMediaFile( strPShaderPath, _T("PixelShader3.txt") );
D3DXAssembleShaderFromFile( strPShaderPath, 0, NULL, &pCode;, NL
```

The inputs to the shader are shown in the following example. The first image shows the vertex colors. The second image shows the texture map.

The resulting image is shown in the following example. It shows the output, which is a blend of the vertex color and the texture image.

Blend Two Textures Using a Constant Color

This example blends two texture maps, using the vertex color, to determine how much of each texture map color to use. The differences between this example and the previous example are as follows:

- The vertex data structure, the FVF macro, and the vertex data include a second set of texture coordinates because there is a second texture. SetTexture is also called twice, using two texture stage states.
- The shader file declares two texture registers and uses the linear interpolate (lrp) instruction to blend the two textures. The values of the diffuse colors determine the ratio of texture0 to texture1 in the output color.

Here is the sample code.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
        DWORD color;
    FLOAT tu1, tv1;
    FLOAT tu2, tv2;
                                // a second set of texture coorc
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3D_FVF_DIFFUSE|D3DFVF_1
static CUSTOMVERTEX g_Vertices[]=
{
    // X
                           color
                                      u1
                                             v1
                                                   u2
                                                         v2
                     Ζ
              У
    { -1.0f, -1.0f, 0.0f, 0xff0000ff, 1.0f, 1.0f, 1.0f, 1.0f },
    { +1.0f, -1.0f, 0.0f, 0xffff0000, 0.0f, 1.0f, 0.0f, 1.0f },
    { +1.0f, +1.0f, 0.0f, 0xffffff00, 0.0f, 0.0f, 0.0f, 0.0f },
    { -1.0f, +1.0f, 0.0f, 0xffffffff, 1.0f, 0.0f, 1.0f, 0.0f },
};
// Create a texture. This file is in the DirectX 8.1 media from
TCHAR strPath[512];
LPDIRECT3DTEXTURE8
                        m_pTexture0, m_pTexture1;
DXUtil_FindMediaFile( strPath, _T("DX5_Logo.bmp"));
D3DUtil_CreateTexture( m_pd3dDevice, strPath, &m;_pTexture0, D3E
DXUtil_FindMediaFile( strPath, _T("snow2.jpg"));
D3DUtil_CreateTexture( m_pd3dDevice, strPath, &m;_pTexture1, D3E
// Load the textures stages.
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
```

```
m_pd3dDevice->SetTexture( 1, m_pTexture1 ); // Use a secc
m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTE>
m_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
m_pd3dDevice->SetPixelShader( m_hPixelShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
Contents of the file "PixelShader5.txt"
ps.1.0 // pixel shader version
tex t0 // texture register t0 is loaded from textur
tex t1 // texture register t1 is loaded from textur
mov r1, t1 // move texture register1 into output regist
lrp r0, v0, t0, r1 // linearly interpolate between t0 and r1 by
// specified in v0
```

The resulting output is as follows:



Microsoft DirectX 8.1 (C++)

Converting Texture Operations

Pixel shaders extend and generalize the multi-texture capabilities of Microsoft® DirectX® 6.0 and 7.0 in the following ways.

- A set of general read/write registers is added to enable more flexible expression. The serial cascade using D3DTA_CURRENT requires the specification of a separate result register argument for each stage.
- The D3DTOP_MODULATE2X and D3DTOP_MODULATE4X texture operations are broken into separate modifiers applicable to any instruction orthogonally. This eliminates the need for separate D3DTOP_MODULATE and D3DTOP_MODULATE2X operations.
- The bias and unbias texture operation modifiers are orthogonal. This eliminates the need for separate add and add bias operations.
- An optional third argument is added to modulate add, so the procedural pixel shader can do arg1×arg2 + arg0. This eliminates the D3DTOP_MODULATEALPHA_ADDCOLOR and D3DTOP_MODULATECOLOR_ADDALPHA texture operations.
- An optional third argument is added to the blend operation, so the procedural pixel shader can use arg0 as the blend proportion between arg1 and arg2. This eliminates the D3DTOP_BLENDDIFFUSEALPHA, D3DTOP_BLENDTEXTUREALPHA,
 D3DTOP_BLENDFACTORALPHA,
 D3DTOP_BLENDTEXTUREALPHAPM, and
 D3DTOP_BLENDCURRENTALPHA texture operations.
- Texture address modifying operations, such as D3DTOP_BUMPENVMAP, are broken out from the color and alpha operations and defined as a third operation type, specifically for operating on texture addresses.

To support this increased flexibility efficiently, the application programming interface (API) syntax is changed from **DWORD** pairs to an ASCII assemble code syntax. This exposes the functionality offered by procedural pixel shaders.

Note When you use pixel shaders, specular add is not specifically controlled by a render state, and it is up to the pixel shader to implement if needed. However, fog blending is still applied by the fixed function pipeline.

Microsoft DirectX 8.1 (C++)

Debugging

MFC Pixel Shader Sample Application

You can use the MFCPixelShader Sample application to learn pixel shader instructions interactively. Programmed into this application are diffuse vertex colors and two texture images. The application has five working pixel shaders that you can select by pushing the buttons in the **Shaders** box. It includes a view window on the left to show the rendered result, an instruction window on the right to allow users to enter instructions for validation, and a third window to view debug output.

As an example, run the application and type the following instructions in the instruction window.

ps.1.0 tex t0 mov r0, t0

This results in the Microsoft® DirectX® 5 logo image in the rendered view window.

This shader applies a texture map. Notice that the compilation result text window says **Success**, which indicates that all the instructions are valid.

Next, remove the second instruction, which is the texture declaration. Once this is deleted, the compilation result says:

(Statement 2) (Validation Error) Read of uninitialized components(*) in t0: *R/X/0 *G/Y/1 *B/Z/2 *A/W/3

This error identifies the statement that fails, *Statement 2*, and why it fails *Uninitialized component in t0*. You can fix this problem by adding Statement 2 again. When you do this, the shader works again.

This is a simple example but it illustrates the usefulness of the tool. By trying different instructions, registers, and instruction sequences, you can better

understand pixel shaders and vertex shaders. The sample application also has an **Open** button, which supports loading of a shader file so that you can load any shader files you have already created.

Shader Debuggers

Some graphics chip companies provide a shader debugging tool on their Web sites. Find these tools by searching the Web or by reading the article listed below. You can attach a debugger to a program while it is running and use the debugger to step through a shader. By setting breakpoints, you can step through the shader code one line at a time and watch register state changes. For more information about vertex shaders and debugging tips, see <u>Using Vertex Shaders:</u> <u>Part 1</u>.

Texture Blending Debugging

Another sample application that is part of the software development kit (SDK) installation is MFCTex. This Microsoft Foundation Classes (MFC) application is a good way to learn how to perform multi-texture blending operations in the fixed function pipeline.

Diagnostic Support

Another option for help with debugging DirectX problems is to use the DirectX Diagnostic Viewer (DXDiag.exe) to create a dump of your machine. This is done by running DxDiag.exe after your machine has crashed and sending the dump to Microsoft, using either the **Report** button on the **More Help** tab or by sending it to directx@microsoft.com. The dump can be used to track down and reproduce the problem.

Additional debug information can be found at <u>http://msdn.microsoft.com/directx</u>

Create an Effect

This example uses an effect file to apply a texture map to an object. The example shows the contents of the effect file, as well as the code required in the application to load and run the file.

To create an effect:

Step 1: Create the Effect File Step 2: Load the Effect File Step 3: Render the Effect

Step 1: Create the effect file

```
/*
 * Step 1: Create the effect file
* This effect file maps a 3-D texture map onto the object.
 * This code needs to be in a file named Effects.fx
 */
Texture DiffuseTexture;
Technique T0
{
   Pass P0
    {
       Texture[0] = NULL;
       PixelShader = NULL;
       VertexShader = XYZ | Normal | Diffuse | Tex1;
       Lighting = False;
       CullMode = None;
       Texture[0] = <DiffuseTexture>;
       ColorOp[0] = SelectArg1;
       ColorArg1[0] = Texture;
       ColorOp[1] = Disable;
   }
}
```

Step 2: Load the Effect File

```
{
    HRESULT hr;
    D3DXTECHNIQUE_DESC technique;
    ID3DXEffect m_pEffect;
    // Assumes that m_pd3dDevice has been initialized
    if(FAILED(hr = D3DXCreateEffectFromFile(m_pd3dDevice, "effect.fx
        return hr;
    if(FAILED(hr = FindNextValidTechnique(NULL, &technique;)))
        return hr;
    m_pEffect->SetTechnique(technique.Index);
    m_pEffect->SetTexture("DiffuseTexture", m_pTexture0);
}
```

Once the effect file is created, **ID3DXEffect::FindNextValidTechnique** returns a technique that has been validated on the hardware.

Step 3: Render the Effect

```
{
    HRESULT hr;
    UINT uPasses;
    if(FAILED(hr = m_pd3dDevice->SetStreamSource(0, m_pVB,
                              sizeof(CUSTOMVERTEX_POS_NORM_COLOR1_TEX
        return hr;
    m_pEffect->Begin(&uPasses;, 0 );
        // The 0 specifies that ID3DXEffect::Begin and ID3DXEffect::
        // save and restore all state modified by the effect.
    for(UINT uPass = 0; uPass < uPasses; uPass++)</pre>
    {
        // Set the state for a particular pass in a technique
        m_pEffect->Pass(uPass);
        m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, dwNumSph
    }
    m_pEffect->End();
}
```

Multiple Techniques

An effect file defines the techniques used. The basic layout of an effect file starts with one or more declarations and then defines each technique for that effect. This sample shows a basic effect file that contains two textures and two techniques. This effect file allows a device that doesn't support single-pass rendering for two textures to use multiple passes to render the textures.

```
// Declare two textures.
texture tex0; // First texture
texture tex1;
                    // Second texture
// Technique 't0' will render the scene in one pass. The color
// for each pixel is calculated to be tex0 + tex1.
                                                    Because it uses
// two textures at once, it will work only on cards that support
// multitexture.
technique t0
{
    pass p0
    {
        Texture[0] = <tex0>;
        ColorOp[0] = SelectArq1;
        ColorArg1[0] = Texture;
        Texture[1] = <tex1>;
        ColorOp[1] = Add;
        ColorArg1[1] = Texture;
        ColorArg2[1] = Current;
        ColorOp[2] = Disable;
    }
}
// Technique 't1' renders the scene in two passes. The first pass s
// each pixel to the color of tex0. The second pass adds in the col
// of tex1. The result should look identical to the first
// technique.
              However, this technique can be used on cards that do
// support multitexture.
technique t1
{
    pass p0
    {
```

```
AlphaBlendEnable = False;
    Texture[0] = <tex0>;
    ColorOp[0] = SelectArg1;
    ColorArg1[0] = Texture;
    ColorOp[1] = Disable;
}
pass p1
{
    AlphaBlendEnable = True;
    SrcBlend = One;
    DestBlend = One;
    Texture[0] = <tex1>;
    ColorOp[0] = SelectArg1;
    ColorArg[0] = Texture;
    ColorOp[1] = Disable;
}
```

}

This example shows the basic syntax and layout of a typical effect file.

```
11
// Sample Effect
// This effect adds two textures, using single pass or multipass tec
11
texture tex0;
texture tex1;
// Single pass
technique t0
{
    pass p0
    {
        Texture[0] = <tex0>;
        Texture[1] = <tex1>;
        ColorOp[0] = SelectArg1;
        ColorArg1[0] = Texture;
        ColorOp[1] = Add;
```

```
ColorArg1[1] = Texture;
        ColorArg2[1] = Current;
        ColorOp[2] = Disable;
    }
}
// Multipass
technique t1
{
    pass p0
    {
        Texture[0] = <tex0>;
        ColorOp[0] = SelectArg1;
        ColorArg1[0] = Texture;
        ColorOp[1] = Disable;
    }
    pass p1
    {
        AlphaBlendEnable = True;
        SrcBlend = One;
        DestBlend = One;
        Texture[0] = <tex1>;
        ColorOp[0] = SelectArg1;
        ColorArg1[0] = Texture;
        ColorOp[1] = Disable;
    }
}
```

Exercise 1: Diffuse Lighting

```
11
// Effect File Workshop Solution for Exercise 1
// Copyright (c) 2001 Microsoft Corporation. All rights reserved.
11
vector lhtR; // Direction of light
vector matD; // Object diffuse material color
matrix mWld; // World
matrix mTot; // Total
// Load model
string XFile = "sphere.x";
// Background color
DWORD BCLR = 0 \times ff333333;
// No pixel shader
pixelshader pNIL;
// Technique names for display in viewer window
string tec0 = "Exercise 1a: Fixed Function Diffuse Lighting";
string tec1 = "Exercise 1b: Vertex Shader Diffuse Lighting";
//////
             Exercise 1a: Fixed Function Diffuse Lighting
                                                             1
//////
          Change diffuse material color to color from model,
                                                             1
//////
          rather than the current white material constant.
                                                             1
// Given: The app has already set the matrices before calling this
technique tec0
{
   pass P0
   {
       // Diffuse, specular, and ambient material colors of object
                                   // Diffuse from objec
       MaterialDiffuse = ;
       MaterialDiffuse = (0.0f, 0.0f, 1.0f, 1.0f); // Diffuse from
       MaterialSpecular = (0.0f, 0.0f, 0.0f, 0.0f);
       MaterialAmbient = (0.0f, 0.0f, 0.0f, 0.0f);
       // Light Properties. lhtR, the light direction, is input fro
```

```
LightType[0]
                       = DIRECTIONAL;
                       = (1.0f, 1.0f, 1.0f, 1.0f);
       LightDiffuse[0]
                       = (0.0f, 0.0f, 0.0f, 0.0f);
       LightSpecular[0]
       LightAmbient[0]
                       = (0.0f, 0.0f, 0.0f, 0.0f);
       LightDirection[0] = ;
                       = 100000.0f;
       LightRange[0]
       // Turn lighting on and use light zero
                       = TRUE;
       LightEnable[0]
       Lighting = TRUE;
       // Assign diffuse color to be used
       ColorOp[0] = SelectArq1;
       ColorArg1[0] = Diffuse;
       Alpha0p[0] = SelectArg1;
       AlphaArg1[0] = Diffuse;
       // Only one color being used
       ColorOp[1] = Disable;
       Alpha0p[1]
                   = Disable;
       // Z-buffering to be used
       ZEnable = true;
       ZWriteEnable = true;
   }
}
//////
             Exercise 1b: Vertex Shader Diffuse Lighting
                                                             1
           Change diffuse material color to color from model,
///////
                                                             1
           rather than the current white material constant.
///////
technique tec1
{
   pass p0
   {
       // Load matrices
       VertexShaderConstant[0] = ;
                                                 // World Matri
                                                 // World*View*
       VertexShaderConstant[4] = ;
       // Material properties of object
       VertexShaderConstant[9] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffu
                                                 // Diffuse fro
       VertexShaderConstant[9] = ;
       VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                       // Specu
       VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                       // Ambie
       // Light Properties. lhtR, the light direction, is input fro
       VertexShaderConstant[13] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffu
```

```
VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specu
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambie
VertexShaderConstant[16] = ;
                                             // Light direc
// Assign diffuse color to be used
ColorOp[0] = SelectArg1;
ColorArg1[0] = Diffuse;
Alpha0p[0] = SelectArg1;
AlphaArg1[0] = Diffuse;
// Only one color being used
ColorOp[1]
           = Disable;
Alpha0p[1]
            = Disable;
// Definition of the vertex shader, declarations then assemb
VertexShader =
decl
{
    stream 0;
    float v0[3];
                     // Position
                     // Normal
    float v3[3];
    float v7[3];
                     // Texture Coord1
                     // Texture coord2
    float v8[3];
}
asm
{
    vs.1.1
                     // Version number
   m4x4 oPos, v0, c4 // Transform point to projection spac
   m3x3 r0,v3,c0 // Transform normal to world Space, p
                      // Dot product against light, r0 now
    dp3 r0,r0,-c16
                           constant in x, y, and z componen
                       11
                      // Modulate against diffuse light col
    mul r0,r0,c13
    mov oD0,r0
                     // Put into diffuse color output.
};
```

```
}
```

}

Exercise 2: Vertex Shader Diffuse Lighting

11 // Effect File Workshop Solution for Exercise 2 // Copyright (c) 2001 Microsoft Corporation. All rights reserved. 11 vector lhtR; // Light direction from app vector matD; // Object diffuse material color matrix mWld; // World
matrix mTot; // Total // Load model string XFile = "f40.x"; // Background color DWORD BCLR = $0 \times ff 0 0 0 0 0;$ // No pixel shader pixelshader pNIL; // Technique names for display in viewer window string tec0 = "Solution 2: Vertex Shader Diffuse Lighting"; ////// Exercise 2: Vertex Shader Diffuse Lighting 1 ////// Light the model taking both diffuse material and 1 diffuse light source into consideration. ////// technique tec0 { pass p0 { //Load matrices VertexShaderConstant[0] = ; // World Mat VertexShaderConstant[4] = ; // World*Vie //Material properties of object VertexShaderConstant[9] // Diffuse fro = ; VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specu VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f);// Ambie

```
// Light Properties. lhtR is input from the shader app
VertexShaderConstant[13] = (1.0f, 0.9f, 0.9f, 1.0f); // Diffu
VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                    // Specu
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambie
VertexShaderConstant[16] = ;
                                                // Light dir
// Assign diffuse color to be used
ColorOp[0] = SelectArg1;
ColorArg1[0] = Diffuse;
AlphaOp[0] = SelectArg1;
AlphaArg1[0] = Diffuse;
// Only one color being used
ColorOp[1] = Disable;
AlphaOp[1] = Disable;
// Definition of the vertex shader, declarations then assemb
VertexShader =
decl
{
    stream 0;
                        // Position
    float v0[3];
    float v3[3];
                        // Normal
    float v7[3];
                       // Texture Coord1
    float v8[3];
                        // Texture coord2
}
asm
{
                        // Version number
    vs.1.1
    m4x4 oPos, v0, c4
                        // Transform point to projection spa
    dp3 r0,v3,-c16
                        // Dot product against untransformed
    mul r0,r0,c13
                        // Modulate against diffuse light co
    mul r0,r0,c9
                        // Modulate against diffuse material
    mov oD0,r0
                        // Put into Diffuse Color output
};
```

```
}
```

}

Exercise 3: Transforms

```
11
// Effect File Workshop Solution for Exercise 3
// Copyright (c) 2001 Microsoft Corporation. All rights reserved.
11
vector lhtR;
              // Light direction from app
              // Object diffuse material color
vector matD;
              // World
matrix mWld;
matrix mTot;
              // Total
// Load model
string XFile = "f40.x";
// Background Color
DWORD BCLR = 0xff000000;
// No pixel shader
pixelshader pNIL;
// Technique names for display in viewer window
string tec0 = "Solution 3: Transforms";
Exercise 3: Transforms
//////
                                                            1
          Transform the vertex normal into world space to take
///////
                                                            1
//////
          light source movement into consideration.
technique tec0
{
   pass p0
   ł
       // Load matrices
       VertexShaderConstant[0] = ;
                                                // World Matri
                                                // World*View*
       VertexShaderConstant[4] = ;
       // Material properties of object
       VertexShaderConstant[9] = ;
                                                // Diffuse fro
       VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specu
       VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                      // Ambie
       // Light Properties. lhtR is input from the shader app
       VertexShaderConstant[13] = (1.0f, 0.9f, 0.9f, 1.0f); // Diffu
       VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                      // Specu
```

```
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambie
VertexShaderConstant[16] = ;
                                                // Light dir
// Assign diffuse color to be used
ColorOp[0] = SelectArg1;
ColorArg1[0] = Diffuse;
AlphaOp[0] = SelectArg1;
AlphaArg1[0] = Diffuse;
// Only one color being used
ColorOp[1] = Disable;
             = Disable;
Alpha0p[1]
// Definition of the vertex shader, declarations then assemb
VertexShader =
decl
{
    stream 0;
    float v0[3];
                        // Position
    float v3[3];
                       // Normal
                       // Texture coord1
    float v7[3];
                       // Texture coord2
    float v8[3];
}
asm
{
    vs.1.1
                        // Version number
    m4x4 oPos, v0, c4
                        // Transform point to projection spa
    mov r0,v3
                        // Copy untransformed normal into r0
                        // Transform normal to world space,
    m3x3 r0,v3,c0
                        //
                             into r0 so preceding mov not ne
    dp3 r0,r0,-c16
                        // Dot product against light, r0 now
                        // in x,y and z components (r,g,b).
    mul
        r0,r0,c13
                        // Modulate against diffuse light co
    mul r0,r0,c9
                        // Modulate against diffuse material
                       // Put into diffuse color output.
    mov oD0,r0
};
```

}

}

Exercise 4: Texturing

```
11
// Effect File Workshop Solution for Exercise 4
// Copyright (c) 2001 Microsoft Corporation. All rights reserved.
11
vector lhtR; // Light direction from app
vector matD; // Object diffuse material of
             // Object diffuse material color
             // World
matrix mWld;
matrix mTot; // Total
texture tDif; // Diffuse texture of object
// Load model
string XFile = "bust.x";
// Background color
DWORD BCLR = 0 \times ff 0 0 0 0 0;
// No pixel shader
pixelshader pNIL;
// Technique names for display in viewer window
string tec0 = "Exercise 4: Texturing";
//////
             Exercise 4: Texturing
                                                              1
//////
          Set up texture to pass onto FF PS
       Modulate between the texture and diffuse color args.
//////
technique tec0
{
   pass p0
   {
       // Load matrices
       VertexShaderConstant[0] = ;
                                                 // World Matri
                                                 // World*View*
       VertexShaderConstant[4] = ;
       // Material properties of object
       VertexShaderConstant[9] = ;
                                                 // Diffuse fro
       VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specu
       VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                       // Ambie
       // Light Properties. lhtR is input from the shader app
```

```
VertexShaderConstant[13] = (0.8f, 0.8f, 0.8f, 0.8f); // Diffu
VertexShaderConstant[14] = (0.0f,0.0f,0.0f,0.0f); // Specu
VertexShaderConstant[15] = (0.3f, 0.3f, 0.3f, 1.0f); // Ambie
                                                  // Light Dir
VertexShaderConstant[16] = ;
// Useful constant(s)
VertexShaderConstant[20] = (-1.0f, -1.0f, 0.5f, 1.0f);
// Assign diffuse texture
Texture[0] = ;
// Set up texture wrapping mode
             = U | V;
wrap0
AddressU[0] = Wrap;
AddressV[0] = Wrap;
// Assign texture color to be used
ColorArq1[0] = Texture;
ColorOp[0] = Modulate; // Modulate between args
ColorArg2[0] = Diffuse; // Add diffuse component as ar
AlphaOp[0] = SelectArg1;
AlphaArg1[0] = Diffuse;
// Ensure remaining stages are disabled
ColorOp[1] = Disable;
AlphaOp[1] = Disable;
// Definition of the vertex shader, declarations then assemb
VertexShader =
decl
{
    stream 0;
    float v0[3];
                      // Position
    float v3[3];
                       // Normal
                      // Texture Coord1
// Texture coord2
    float v7[3];
    float v8[3];
                        // Texture coord2
}
asm
{
                        // Version number
    vs.1.1
    m4x4 oPos, v0, c4
                        // Transform point to projection spa
                        // Transform Normal to World Space,
    m3x3 r0,v3,c0
                        // Dot product against light, r0 now
    dp3 r0,r0,-c16
                              constant in x, y and z component
                         //
    mul r0,r0,c13
                        // Modulate against diffuse light co
    mul r0,r0,c9
                        // Modulate against diffuse material
```

Exercise 5: Vertex Shader Specular Lighting

```
11
// Effect File Workshop Exercise 5
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
vector lhtR; // Light direction
matrix mWld; // World
matrix mTot; // Total
vector matD; // Material diffuse
vector matS; // Material specular
vector vCPS; // Camera position
// Background color
DWORD BCLR = 0 \times FF000000;
pixelshader pNIL;
string XFile = "f40.x";
// Technique names for display in viewer window
string tec0 = "Exercise 5: Vertex Shader Specular Lighting";
technique tec0
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ;
                                                     // World Matrix
                                                       // World*View*Pr
        VertexShaderConstant[4] = ;
        // Material properties of object
        VertexShaderConstant[9] = ;
                                                        // Diffuse
                                                        // Specular
        VertexShaderConstant[10] = ;
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        // Properties of light
        VertexShaderConstant[13] = (1.0f, 0.0f, 0.0f, 1.0f); // Diffuse
        VertexShaderConstant[14] = (0.0f,0.0f,0.0f,0.0f); // Specula
```

```
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
VertexShaderConstant[16] = ;
                                           // Light directi
// Blending Constants
VertexShaderConstant[20] = (0.7f, 0.7f, 0.7f, 0.7f);
VertexShaderConstant[21] = (0.3f, 0.3f, 0.3f, 0.3f);
// Camera Information.
VertexShaderConstant[24] = ;
ColorOp[0] = SelectArg1;
ColorArg1[0] = Diffuse;
AlphaOp[0] = SelectArg1;
AlphaArg1[0] = Diffuse;
ColorOp[1] = Disable;
AlphaOp[1] = Disable;
VertexShader =
decl
{
    stream 0;
    float v0[3]; // Position
    float v3[3]; // Normal
   float v7[3]; // Texture coord1
   float v8[3]; // Texture coord2
}
asm
{
                     // Version number
    vs.1.1
   m4x4 oPos, v0, c4 // Transform point to projection spac
   m4x4 r0,v0,c0 // Transform point to world space
   add r0,-r0,c24 // Get a vector toward the camera pos
                          // This is the negative of the ca
    // Normalize
    dp3 r11.x,r0.xyz,r0.xyz // Load the square into r1
    rsg r11.xyz,r11.x // Get the inverse of the squa
   mul r0.xyz,r0.xyz,r11.xyz // Multiply, r0 = -(camera vec
   add r2.xyz,r0.xyz,-c16 // Get half angle
    // Normalize
    dp3 r11.x,r2.xyz,r2.xyz // Load the square into r1
    rsg r11.xyz,r11.x // Get the inverse of the squa
   mul r2.xyz,r2.xyz,r11.xyz // Multiply, r2 = HalfAngle
   m3x3 r1,v3,c0
                             // Transform normal to world s
    // r2 = half angle, r1 = normal, r3 (output) = intensity
```

dp3 r3.xyzw,r1,r2 // Now raise it several times mul r3,r3,r3 // 2nd mul r3,r3,r3 // 4th mul r3,r3,r3 // 8th mul r3,r3,r3 // 16th // Compute diffuse term dp3 r4, r1, -c16 // Blend it in mul r3,c20,r3 // Kd mul r4,r4,c21 // Ks mul r4,r4,c10 // Specular mad r4,r3,c9,r4 // Diffuse mov oD0,r4 // Put into Diffuse Color }; }

}

Exercise 5B

```
11
// Effect File Workshop Exercise 5B
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
vector lhtR; // Light Direction
matrix mWld; // World
matrix mTot; // Total
vector vCPS; // Camera position
texture tEnv; // Environment texture
texture tDif;
vector matD; // Object Diffuse Material Color
vector matS;
                // Object Specular Material Color
// Background color
DWORD BCLR = 0 \times FF000000;
pixelshader pNIL;
//string XFile = "f40.x";
string XFile = "viper.x";
string BIMG = "lobbyzneg.bmp";
// Technique names for display in viewer window
string tec0 = "Exercise 5b: Vertex Shader Specular Envmap";
technique tec0
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ; // World Matrix
VertexShaderConstant[4] = ; // World*View*Proj Matrix
        // Material properties of object
        VertexShaderConstant[9] = ;
                                                        // Diffuse
        VertexShaderConstant[10] = (0.0f,0.0f,0.0f,0.0f); // Specula
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
```

```
// Properties of light
VertexShaderConstant[13] = (1.0f, 0.0f, 0.0f, 1.0f); // Diffuse
VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
VertexShaderConstant[16] = ;
                                              // Light Direc
// Blending constants
VertexShaderConstant[20] = (-2.0f, -2.0f, -2.0f, -2.0f);
VertexShaderConstant[21] = (0.25f, 0.25f, 0.25f, 0.05f);
VertexShaderConstant[22] = ( 0.75f, 0.75f, 0.75f, 0.95f );
VertexShaderConstant[23] = ( 1.00f, 1.00f, 1.00f, 1.00f );
// Camera information
VertexShaderConstant[24] = ;
ColorOp[0] = Modulate;
ColorArg2[0] = Diffuse;
ColorArg1[0] = Texture;
AlphaOp[0] = SelectArg1;
AlphaArg1[0] = Diffuse;
ColorOp[1] = Disable;
AlphaOp[1] = Disable;
Texture[0]
             = ;
PixelShader = ;
AlphaBlendEnable = True;
SrcBlend = One;
DestBlend = InvSrcAlpha;
VertexShader =
decl
{
    stream 0;
    float v0[3]; // Position
    float v3[3]; // Normal
    float v7[3]; // Texture coord1
    float v8[3]; // Texture coord2
}
asm
{
                       // Version number
    vs.1.1
    m4x4 oPos, v0, c4 // Transform point to projection spac
    m4x4 r0,v0,c0 // Transform point to world space
    add r0,r0,-c24
                       // Get a vector toward the camera pos
                       // This is the camera direction
```

```
// Normalize
             dp3 r11.x,r0.xyz,r0.xyz // Load the square into r1
             rsq r11.xyz,r11.x
                                         // Get the inverse of the squa
             mul r0.xyz,r0.xyz,r11.xyz // Multiply, r0 = (camera vect
                                         // Transform normal to world s
             m3x3 r1,v3,c0
             dp3 r3,r0,r1
                                         // Dot product Cam*Normal
             mul r2,c20,r3
             mad oT0.xyz,r2,r1,r0 // Compute reflection vector
             // (1-cos)<sup>4</sup> = approx fresnel
                                                  // Complement color
             add r0,c23,r3
             mul r1,r0,r0
                                         // Square
             mul r0,r1,r1
                                         // 4th
             mul r0,r0,c22
11
             mov r1,c9
11
             mul r1, r1, c21
                                        // Blend in scaled diffuse mat
             add oD0,r0,r1
                                                  // Put into Diffuse Co
        };
    }
}
technique tec1
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ; // World Matrix
VertexShaderConstant[4] = ; // World*View*Proj Matrix
        // Material properties of object
        VertexShaderConstant[9]
                                                         // Diffuse
                                   = ;
        VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        // Properties of light
        VertexShaderConstant[13] = (1.0f, 0.0f, 0.0f, 1.0f); // Diffuse
        VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
                                                           // Light direc
        VertexShaderConstant[16] = ;
        // Blending Constants
        VertexShaderConstant[20] = (-2.0f, -2.0f, -2.0f, -2.0f);
        VertexShaderConstant[21] = ( 0.25f, 0.25f, 0.25f, 0.05f );
VertexShaderConstant[22] = ( 0.75f, 0.75f, 0.75f, 0.95f );
        VertexShaderConstant[23] = ( 1.00f, 1.00f, 1.00f, 1.00f );
```

```
VertexShaderConstant[24] = ( 1.0f, 1.0f, 1.0f, 1.0f );
        // Camera Information
        VertexShaderConstant[25] = ;
        ColorOp[0]
                     = Modulate;
        ColorArq1[0] = Texture;
        ColorArg2[0] = Diffuse;
        Alpha0p[0] = SelectArg1;
        AlphaArg1[0] = Diffuse;
        ColorOp[1] = Add;
        ColorArg1[1] = Current;
        ColorArg2[1] = Specular;
        ColorOp[2]
                     = Disable;
        Alpha0p[2] = Disable;
        Texture[0]
                     = ;
        PixelShader = ;
11
        AlphaBlendEnable = True;
11
        SrcBlend = One; //SrcAlpha;
11
        DestBlend = InvSrcAlpha;
11
        CullMode = None;
        SpecularEnable = True;
        VertexShader =
        decl
        {
            stream 0;
            float v0[3]; // Position
            float v3[3]; // Normal
            float v7[3]; // Texture coord1
            float v8[3]; // Texture coord2
        }
        asm
        {
                               // Version number
            vs.1.1
            m4x4 oPos, v0, c4 // Transform point to projection spac
                              // Transform point to World Space
            m4x4 r0,v0,c0
            add r0,r0,-c25
                               // Get a vector toward the camera pos
                               11
                                    this is the camera direction
            // Normalize
            dp3 r11.x,r0.xyz,r0.xyz // Load the square into r1
```

```
rsq r11.xyz,r11.x // Get the inverse of the squa
            mul r0.xyz,r0.xyz,r11.xyz // Multiply, r0 = (camera vect
            m3x3 r1,v3,c0
                                       // Transform normal to world s
            dp3 r3,r0,r1
                                      // Dot product Cam*Normal
            mul r2,c20,r3
            mad oT0.xyz,r2,r1,r0 // Compute reflection vector
            // (1-cos)<sup>4</sup> = approx fresnel
            add r0,c23,r3
                                       // Complement color
                                       // Square
            mul r1,r0,r0
            add oD0,r1,c21
            mov r1,c9
                                      // Blend in scaled diffuse mat
            mul oD1,r1,c21
        };
    }
}
technique tec2
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ; // World Matrix
VertexShaderConstant[4] = ; // World*View*Proj Matrix
        // Material properties of object
        VertexShaderConstant[9] = ;
                                                      // Diffuse
        VertexShaderConstant[10] = ;
                                                      // Specular
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        // Properties of light
        VertexShaderConstant[13] = (1.0f, 0.0f, 0.0f, 1.0f); // Diffuse
        VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        VertexShaderConstant[16] = ;
                                                        // Light direc
        // Blending constants
        VertexShaderConstant[20] = (-2.0f, -2.0f, -2.0f, -2.0f);
        VertexShaderConstant[21] = ( 0.25f, 0.25f, 0.25f, 0.05f );
        VertexShaderConstant[22] = ( 0.75f, 0.75f, 0.75f, 0.95f );
        VertexShaderConstant[23] = (1.00f, 1.00f, 1.00f, 1.00f);
        VertexShaderConstant[24] = ( 1.0f, 1.0f, 1.0f, 1.0f );
        // Camera information
        VertexShaderConstant[25] = ;
```

```
ColorOp[0] = Modulate;
       ColorArg1[0] = Texture;
       ColorArg2[0] = Diffuse;
       AlphaOp[0] = SelectArg1;
       AlphaArg1[0] = Diffuse;
       ColorOp[1] = Add;
       ColorArg1[1] = Current;
       ColorArg2[1] = Specular;
       ColorOp[2] = Disable;
       AlphaOp[2] = Disable;
       Texture[0]
                    = ;
       PixelShader = ;
11
       FillMode = Wireframe;
           SpecularEnable = True;
       VertexShader =
       decl
       {
           stream 0;
           float v0[3]; // Position
           float v3[3]; // Normal
           float v7[3]; // Texture coord1
           float v8[3]; // Texture coord2
       }
       asm
       {
                             // Version number
           vs.1.1
           m4x4 oPos, v0, c4 // Transform point to projection spac
           m4x4 r0,v0,c0
                             // Transform point to world Space
           add r0,r0,-c25
                              // Get a vector toward the camera pos
                              // this is the camera direction
            //Normalize
           dp3 r11.x,r0.xyz,r0.xyz // Load the square into r1
            rsq r11.xyz,r11.x // Get the inverse of the squa
           mul r0.xyz,r0.xyz,r11.xyz // Multiply, r0 = (camera vect
                                     // Transform normal to world s
           m3x3 r1,v3,c0
           dp3 r3,r0,r1
                                  // Dot product Cam*Normal
           mul r2,c20,r3
           mad oT0.xyz,r2,r1,r0 // Compute reflection vector
```

```
//(1-\cos)^4 = approx fresnel
             add r0,c23,r3
                                                     // Complement colo
            mul r1,r0,r0
                                                     // Square
11
            add r1, r1, c21
11
            mul oD0, r1, c10
            mad oD0, r1, c10, c10
            mov r1,c9
                                       // blend in scaled diffuse mat
            mul oD1,r1,c21
        };
    }
}
technique tec4
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ; // World Matrix
VertexShaderConstant[4] = ; // World*View*Proj Matrix
        // Material properties of object
        VertexShaderConstant[9] = ;
                                                       // Diffuse
        VertexShaderConstant[10] = ;
                                                       // Specular
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        // Properties of light
        VertexShaderConstant[13] = (1.0f, 0.0f, 0.0f, 1.0f); // Diffuse
        VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        VertexShaderConstant[16] = ;
                                                         // Light direc
        // Blending Constants
        VertexShaderConstant[20] = (-2.0f, -2.0f, -2.0f, -2.0f);
        VertexShaderConstant[21] = ( 1.0f, 1.0f, 1.0f, 1.0f );
        VertexShaderConstant[22] = ( 0.75f, 0.75f, 0.75f, 0.95f );
        VertexShaderConstant[23] = (1.00f, 1.00f, 1.00f, 1.00f);
        // Camera Information
        VertexShaderConstant[24] = ;
11
        FillMode = Wireframe;
        ColorOp[0] = Modulate;
        ColorArg2[0] = Diffuse;
```

```
ColorArg1[0] = Texture;
       AlphaOp[0] = SelectArg1;
       AlphaArg1[0] = Diffuse;
       ColorOp[1] = Disable;
       AlphaOp[1] = Disable;
       Texture[0]
                    = ;
       PixelShader = ;
       AlphaBlendEnable = True;
       SrcBlend = One;
       DestBlend = InvSrcAlpha;
11
       CullMode = None;
       VertexShader =
       decl
       {
           stream 0;
           float v0[3]; // Position
           float v3[3]; // Normal
           float v7[3]; // Texture coord1
           float v8[3]; // Texture coord2
       }
       asm
       {
                             // Version number
           vs.1.1
           m4x4 oPos, v0, c4 // Transform point to projection spac
           m4x4 r0,v0,c0 // Transform point to World Space
           add r0,r0,-c24 // Get a vector toward the camera pos
                              // this is the camera direction
           // Normalize
           dp3 r11.x,r0.xyz,r0.xyz // Load the square into r1
           rsg r11.xyz,r11.x // Get the inverse of the squa
           mul r0.xyz,r0.xyz,r11.xyz // Multiply, r0 = (camera vect
                             // Transform normal to world s
           m3x3 r1,v3,c0
           dp3 r3,r0,r1
                                    // Dot product Cam*Normal
           mul r2,c20,r3
           mad oT0.xyz,r2,r1,r0 // Compute reflection vector
           // (1-\cos)^4 = approx fresnel
           add r0,c23,r3
                                                      // Complemen
           mul r1,r0,r0
                                                      // Square
```

			mul r0,r1,r1	// 4th
//			mul r0,r0,c22 mov r1,c9 add r0, r0, c10 add oD0,r0,r1	// Add in specular // Put into Diffuse Color
	ι	};		
}	ſ			

Exercise 5C

```
11
// Effect File Workshop Exercise 5C
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
                // Light direction
vector lhtR;
matrix mWld; // World
matrix mTot;
                // Total
vector vCPS; // Camera position
texture tEnv;
                // Environment texture
texture tDif;
                // Object diffuse material color
vector matD;
// Background color
DWORD BCLR = 0 \times FF000000;
pixelshader pNIL;
// string XFile = "sphere.x";
// string XFile = "f40.x";
string XFile = "viper.x";
string BIMG = "lobbyzneg.bmp";
// Technique names for display in viewer window
string tec0 = "Exercise 5b: Vertex Shader Specular Envmap";
technique tec0
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ; // World Matrix
VertexShaderConstant[4] = ; // World*View*Proj Matrix
        // Material properties of object
        VertexShaderConstant[9] = ;
                                                       // Diffuse
        VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
```

```
// Properties of light
VertexShaderConstant[13] = (1.0f, 0.0f, 0.0f, 1.0f); // Diffuse
VertexShaderConstant[14] = (0.0f,0.0f,0.0f,0.0f); // Specula
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
                                              // Light direc
VertexShaderConstant[16] = ;
// Blending Constants
VertexShaderConstant[20] = (-2.0f, -2.0f, -2.0f, -2.0f);
VertexShaderConstant[25] = ( 1.0f, 1.0f, 1.0f, 1.0f );
VertexShaderConstant[21] = (0.25f, 0.25f, 0.25f, 0.05f);
VertexShaderConstant[22] = ( 0.75f, 0.75f, 0.75f, 0.95f );
VertexShaderConstant[23] = ( 1.00f, 1.00f, 1.00f, 1.00f );
// Camera information
VertexShaderConstant[24] = ;
ColorOp[0]
             = Modulate;
ColorArg1[0] = Texture;
ColorArg2[0] = Diffuse;
Alpha0p[0] = SelectArg1;
AlphaArq1[0] = Diffuse;
ColorOp[1] = Add;
ColorArg1[1] = Current;
ColorArg2[1] = Specular;
AlphaOp[1] = Disable;
Texture[0]
             = ;
PixelShader = ;
AlphaBlendEnable = True;
SrcBlend = One;//SrcAlpha;
DestBlend = InvSrcAlpha;
CullMode = None;
SpecularEnable = True;
VertexShader =
decl
{
    stream 0;
    float v0[3]; // Position
    float v3[3]; // Normal
    float v7[3]; // Texture coord1
    float v8[3]; // Texture coord2
```

11

11

11

11

```
}
        asm
        {
                             // Version number
            vs.1.1
           m4x4 oPos, v0, c4 // Transform point to projection spac
           m4x4 r0,v0,c0 // Transform point to World Space
            add r0,r0,-c24 // Get a vector toward the camera pos
                               // this is the camera direction
            // Normalize
            dp3 r11.x,r0.xyz,r0.xyz // Load the square into r1
            rsq r11.xyz,r11.x // Get the inverse of the squa
            mul r0.xyz,r0.xyz,r11.xyz // Multiply, r0 = (camera vect
            m3x3 r1,v3,c0
                                     // Transform normal to world s
            dp3 r3,r0,r1
                                  // Dot product Cam*Normal
            mul r2,c20,r3
            mad oT0.xyz,r2,r1,r0 // Compute reflection vector
            // (1-cos)<sup>4</sup> = approx fresnel
                                                  // Complement colo
            add r0,c23,r3
            mul r1,r0,r0
                                                  // Square
11
                                                  // 4th
            mul r0,r1,r1
11
            mul r0,r1,r1
                add oD0,r1,c21
            mul r0,r0,c22
            mov r1,c9
                              // Blend in scaled diffuse mat
            mul oD1,r1,c25
           add oD0,r0,r1 // Put into Diffuse Color
add oD0,r0,r1 // Put into Diffuse Color
11
11
       };
   }
```

Exercise 6: Standard Texture Effect

```
11
// Standard Texture Effect
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
vector matD; // Material diffuse
vector matS; // Material specular
vector matA; // Material ambient
DWORD lhtT; // Light type
vector lhtD; // Light diffuse
vector lhtS; // Light specular
vector lhtA; // Light ambient
vector lhtR; // Light direction
vector lhtP; // Light position
vector vOff; // Emboss offset
vector vVwD; // View direction
vector vCPS; // Camera position
matrix mEnv; // Environment map transform
matrix mWld; // World
matrix mTot; // Total
                           // Blending matrices
matrix mWl0;
matrix mWl1;
matrix mWl2;
matrix mWl3;
matrix mWl4;
matrix mWl5;
matrix mWl6;
matrix mWl7;
matrix mIdt =
Γ
       1.0,0.0,0.0,0.0,
       0.0, 1.0, 0.0, 0.0,
       0.0, 0.0, 1.0, 0.0,
       0.0, 0.0, 0.0, 1.0
];
DWORD BCLR = 0 \times 0000000;
```

```
texture tDif;
               // Diffuse texture
texture tEnv; // Environment texture
texture tEvC;
               // Circulary integratex cube texture
texture tEvL;
texture tDf2;
               // Second texture
texture tDf3;
texture tDf4;
               // Procedural satin texture
texture tSt1;
texture tSt2; // Procedural stain texture2
texture tMt1;
               // Brushed metal texture
texture tFrn; // Fresnel Shader
texture tGlw; // Glow Shader
texture tL10; // Light lookup texture for spec
texture tL80;
texture tL64;
texture tL32;
texture tL16;
texture tfq1;
texture tfg2;
texture tNSE;
texture tNSN;
pixelshader pNIL;
string XFile = "tiny.x";
string Skinned = "true";
// Skinned version, lots of transforms have to happen here
vertexshader sDif =
decl
{
    stream 0;
    float v0[3]; // Blend weights
float v1[3]; // Indices
    ubyte v2[4];
// OR
   d3dcolor v2[1]; // if hardware doesn't support ubyte, use d3dcol
11
    float v3[3];
    float v7[3];
    float v8[3];
}
asm
{
    vs.1.1
    // The indices are put into an color
    // If ubyte is supported, this the right way to do it
```

```
mul r1,v2.xyzw,c41.zzzz
// OR
// If ubyte is not supported, decode from a 32 bit d3dcolor valu
//mul r1,v2.zyxw,c41.wwww
// First compute the last blending weight
mov r0.xyz,v1.xyz;
dp3 r0.w,v1.xyz,c40.xxx;
add r0.w, -r0.w, c40.x
// Now do a bunch of matrix multiples,
    r5 = Position
11
// r6 = Normal
mov a0.x,r1.x
mov r5,v0
m4x3 r5,v0,c[a0.x]; //World matrices start at 0
m3x3 r6,v3,c[a0.x];
// Blend them
mul r5, r5, r0.xxxx
mul r6, r6, r0.xxxx
// Set 2
mov a0.x,r1.y
m4x3 r2,v0,c[a0.x];
m3x3 r3,v3,c[a0.x];
// Add them in
mad r5,r2,r0.yyyy,r5;
mad r6,r3,r0.yyyy,r6;
// Set 3
mov a0.x,r1.z
m4x3 r2,v0,c[a0.x];
m3x3 r3,v3,c[a0.x];
// Add them in
mad r5,r2,r0.zzzz,r5;
mad r6,r3,r0.zzzz,r6;
// Set 4
mov a0.x,r1.w
m4x3 r2,v0,c[a0.x];
m3x3 r3,v3,c[a0.x];
// Add them in
mad r5,r2,r0.www,r5;
```

```
mad r6,r3,r0.www,r6;
    // Compute position
    mov r5.w, c40.x
   m4x4 oPos,r5,c50;
    dp3 r11.x,r6.xyz,r6.xyz // Load the square into r1
    rsq r11.xyz,r11.x
                             // Get the inverse of the square
    mul r6.xyz,r6.xyz,r11.xyz // Multiply
    dp3 r4.xyz,r6,-c48
   mov oD0.xyz,r4.xyz
   mov oT0.xy,v7.xy
};
//Skinned Diffuse 1
string tec0 = "Exercise 6: Skinned Diffuse";
technique tec0
{
    pass P0
    {
        ColorOp[0] = Modulate;
        ColorArg1[0] = Texture;
        ColorArg2[0] = Diffuse;
        AlphaOp[0] = SelectArg1;
        AlphaArg1[0] = Current;
        ColorOp[1]
                     = Disable;
        AlphaOp[1] = Disable;
        MinFilter[0] = Linear;
        MagFilter[0] = Linear;
        MipFilter[0] = Linear;
        VertexShaderConstant[0] = <mWl0>;
        VertexShaderConstant[4] = <mWl1>;
        VertexShaderConstant[8] = <mWl2>;
        VertexShaderConstant[12] = <mWl3>;
        VertexShaderConstant[16] = <mWl4>;
        VertexShaderConstant[20] = <mWl5>;
        VertexShaderConstant[24] = <mWl6>;
        VertexShaderConstant[28] = <mWl7>;
        VertexShaderConstant[32] = <mIdt>;
        VertexShaderConstant[50] = <mTot>;
                                                                // mT
        VertexShaderConstant[48] = <lhtR>;
        VertexShaderConstant[40] = (1.0f, -1.0f, 0.0f, .0f);
```

```
VertexShaderConstant[41] = (0.00390625f, 256.0f, 4.0f,1020.0
VertexShaderConstant[43] = (0.0, 0.0, 1.0, 0.0f);
VertexShaderConstant[44] = (0.0, 0.0, 0.0, 0.0);
VertexShaderConstant[60] = (0.5, 0.5, 0.5, 0.5);
VertexShaderConstant[60] = (1.0f, 1.0f, 1.0f, 1.0f);
VertexShaderConstant[61] = (.50f, 0.43f, 0.38f, 1.0f);
                                                         // Sk
VertexShaderConstant[62] = (0.18f, .10f, 0.15f, 1.0f);
                                                         // Gr
VertexShaderConstant[63] = <matD>;
                                                         // Ob
VerteXShaderConstant[64] = (0.0f, 1.0f, 0.0f, 1.0f);
                                                         // Sk
Texture[0] = <tDif>;
vertexshader = <sDif>;
             = U | V;
wrap0
            = U | V;
wrap1
AddressU[0] = Wrap;
AddressV[0] = Wrap;
AlphaBlendEnable = False;
```

```
}
```

Exercise 7: Multi-Texturing with Shaders

```
11
// Effect File Workshop Exercise 7
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
vector lhtR;
                // Light direction
matrix mWld; // World
matrix mTot; // Total
texture tDif; // Diffuse texture of object
texture tNSE; // Noise texture
// Background color
DWORD BCLR = 0 \times FF0000FF;
pixelshader pNIL;
string XFile = "sphere.x";
// Technique names for display in viewer window
string tec0 = "Exercise 7: Multi-Texturing with shaders";
technique tec0
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = <mWld>; // World Matrix
VertexShaderConstant[4] = <mTot>; // World*View*Proj Matr
        // Material properties of object
        VertexShaderConstant[9] = (1.0f,1.0f,1.0f,1.0f); // Diffuse
        VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        // Properties of light
        VertexShaderConstant[13] = (1.0f, 0.0f, 0.0f, 1.0f); // Diffuse
        VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        VertexShaderConstant[16] = <lhtR>;
                                                               // Light d
```

```
// Useful constant(s)
VertexShaderConstant[20] = (-1.0f, -1.0f, -1.0f, -1.0f);
Texture[0]
            = <tDif>;
          = <tNSE>;
Texture[1]
wrap0
            = U | V;
wrap1
            = U | V;
AddressU[0] = wrap;
AddressV[0] = wrap;
AddressU[1] = wrap;
AddressV[1] = wrap;
MinFilter[0] = Linear;
MagFilter[0] = Linear;
MinFilter[1] = Linear;
MagFilter[1] = Linear;
VertexShader =
decl
{
    stream 0;
                    // Position
    float v0[3];
                     // Normal
    float v3[3];
                     // Texture coord1
    float v7[3];
                 // Texture coord2
    float v8[3];
}
asm
{
                       // Version number
    vs.1.1
    m4x4 oPos, v0, c4 // Transform point to projection spac
                       // Transform normal to world space, p
    m3x3 r0,v3,c0
                       11
                            result into r0
                       // Dot product against light, r0
    dp3 r0,r0,-c16
                       // now has lighting constant in x,y a
                       // components (r,g,b)
                      // Copy texture coordinates to r0
    mov r0.xy,v7.xy
    //mul r0.y,r0.y,c20 // Invert texture coordinates
    mov oT0.xy,r0.xy // Copy texture coordinates to oT0
    mov oT1.xy,r0.xy // Copy texture coordinates to oT1
    mov oD0,r0
                      // Copy diffuse to output
};
PixelShader =
asm
{
```

```
ps.1.1
tex t0 // Get texture sample from stage 0
tex t1 // Get texture sample from stage 1
mul_x2 r0,t1,t0; // Blend them together in an interest
};
}
```

Exercise 8: Texturing with Lights

```
11
// Effect File Workshop Exercise 8
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
vector lhtR; // Light direction
matrix mWld; // World
matrix mTot; // Total
texture tDif; // Diffuse texture of object
texture tNSE; // Noise Texture
vector vCPS; // Camera position
// Background color
DWORD BCLR = 0 \times FF0000FF;
pixelshader pNIL;
string XFile = "sphere.x";
// Technique names for display in viewer window
string tec0 = "Exercise 8: Texturing with lights";
technique tec0
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = <mWld>;
                                                            // World M
        VertexShaderConstant[4] = <mTot>;
                                                             // World*V
        // Material properties of object
        VertexShaderConstant[9] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffuse
        VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        // Properties of light
        VertexShaderConstant[13] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffuse
        VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        VertexShaderConstant[16] = <lhtR>;
                                                             //Light Di
```

```
// Useful constants
VertexShaderConstant[20] = (1.0f, 1.0f, 1.0f, 1.0f);
VertexShaderConstant[21] = (1.0f, 1.0f, 1.0f, 0.0f);
// Camera Information
VertexShaderConstant[24] = <vCPS>;
Texture[0] = <tDif>;
Texture[1] = <tNSE>;
             = U | V;
wrap0
            = U | V;
wrap1
AddressU[0] = wrap;
AddressV[0] = wrap;
AddressU[1] = wrap;
AddressV[1] = wrap;
MinFilter[0] = Linear;
MagFilter[0] = Linear;
MinFilter[1] = Linear;
MagFilter[1] = Linear;
VertexShader =
decl
{
    stream 0;
                            // Position
// Normal
    float v0[3];
    float v3[3];
                             // Texture coord1
    float v7[3];
                             // Texture coord2
    float v8[3];
}
asm
{
                             // Version number
    vs.1.1
                         // Transform point to proj-
// Transform point to world sp
    m4x4 oPos, v0, c4
    m4x4 r0,v0,c0
    add r0,-r0,c24
                             // Get a vector toward the car
                              //
                                   this is the negative of t
    // Normalize
    dp3 r11.x,r0.xyz,r0.xyz // Load the square into r1
                             // Get the inverse of the squa
    rsq r11.xyz,r11.x
    mul r0.xyz,r0.xyz,r11.xyz // Multiply, r0 = -(camera vec
    add r2.xyz,r0.xyz,-c16 // Get half angle
    // Normalize
    dp3 r11.x,r2.xyz,r2.xyz // Load the square into r1
    rsq r11.xyz,r11.x // Get the inverse of the squa
    mul r2.xyz,r2.xyz,r11.xyz // Multiply, r2 = HalfAngle
```

```
// Transform normal to world s
    m3x3 r1,v3,c0
     // r2 = half angle, r1 = normal, r3 (output) = intensity
     dp3 r3.xyzw,r2,r1
     // Now raise it several times
    mul r3,r3,r3
                                   // 2nd
    mul r3,r3,r3
mul r3,r3,r3
mul r3,r3,r3
                                  // 4th
                                 // 8th
                                   // 16th
    mul r3,r3,c20
    // Compute diffuse term
    dp3 r4, r1, -c16
    mul r4, r4, c21
    mov oD0,r4
    mov oD1,r3
    mov oT0.xy,v7.xy // Copy texture coordinates to
mov oT1.xy,v7.xy // Copy texture coordinates to
};
PixelShader =
asm
{
   ps.1.1
   tex t0
                                    // Sample texture 0
                                    // Sample texture 1
   tex t1
                                   // Blend them together
   mul_x2 r1,t1,t0;
   mov
            r0,r1
           r0,r1,v0; // Modulate diffuse
r1,r1,v1; // Modulate specular
r0,r0,v1; // Blend them together
   mul
   mul
   add
};
```

```
}
```

Exercise 9: Bump Mapping

```
11
// Effect File Workshop Exercise 9
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
vector lhtR; // Light direction
matrix mWld; // World
                // Total
matrix mTot;
texture tDif; // Diffuse texture of object
texture tDf3; // Normal map for earth
                 // Light lookup texture for spec
texture tL10;
texture tL80;
texture tL64;
texture tL32;
texture tL16;
vector vCPS; // Camera Position
// Background color
DWORD BCLR = 0 \times FF0000FF;
pixelshader pNIL;
string XFile = "sphere.x";
// Technique names for display in viewer window
string tec0 = "Exercise 9a: Dot 3 Bump Mapping";
string tec1 = "Exercise 9b: Dot 3 Specular Bump Mapping";
string tec2 = "Exercise 9c: Table Lookup Specular Bump Mapping";
technique tec0
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ; // World Matrix
VertexShaderConstant[4] = ; // World*View*Proj Matrix
        // Material properties of object
        VertexShaderConstant[9] = (0.8f, 0.8f, 0.8f, 0.8f); // Diffuse
        VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
```

```
VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
// Properties of light
VertexShaderConstant[13] = (0.7f, 0.7f, 0.7f, 0.7f); // Diffuse
VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
VertexShaderConstant[16] = ; //Light Direction
vertexShaderConstant[20] = (.5f,.5f,.5f,.5f);
// Camera information
VertexShaderConstant[24] = ;
Texture[0]
             = ;
Texture[1] = ;
            = U | V;
wrap0
           = U | V;
wrap1
AddressU[0] = wrap;
AddressV[0] = wrap;
AddressU[1] = wrap;
AddressV[1] = wrap;
MinFilter[0] = Linear;
MagFilter[0] = Linear;
MinFilter[1] = Linear;
MagFilter[1] = Linear;
VertexShader =
decl
{
    stream 0;
    float v0[3]; // Position
    float v3[3]; // Normal
    float v7[3]; // Texture Coord1
    float v8[3]; // Tangent
}
asm
{
    vs.1.1
    // Transform position
    m4x4 oPos,v0,c4
    // Transform normal and tangent
    m3x3 r7,v8,c0
    m3x3 r8,v3,c0
    // Cross product
    mul r0, -r7.zxyw, r8.yzxw;
    mad r5, -r7.yzxw, r8.zxyw, -r0;
```

```
// Transform the light vector
            dp3 r6.x,r7,-c16
            dp3 r6.y,r5,-c16
            dp3 r6.z,r8,-c16
            // Multiply by a half to bias, then add half
            mad r6.xyz,r6.xyz,c20,c20
            mov oT0.xy,v7.xy
            mov oT1.xy,v7.xy
            mov oD0.xyz,r6.xyz
        };
        PixelShader =
        asm
        {
            ps.1.1
            tex t0
                         // Sample texture
                         // Sample normal
            tex t1
            mov r0,t1
            dp3 r0,t1_bx2,v0_bx2; // Dot(light,normal)
            mul r0,t0,r0
                                   // Modulate against base color
        };
    }
}
technique tec1
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ; // World matrix
VertexShaderConstant[4] = ; // World*View*Proj matrix
        // Material properties of object
        VertexShaderConstant[9] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffuse
        VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        // Properties of light
        VertexShaderConstant[13] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffuse
        VertexShaderConstant[14] = (0.0f,0.0f,0.0f,0.0f); // Specula
        VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
        VertexShaderConstant[16] = ;
                                                       // Light directi
        VertexShaderConstant[20] = (.5f,.5f,.5f,.5f);
```

```
// Camera information
VertexShaderConstant[24] = ;
Texture[0]
             = ;
Texture[1] = ;
wrap0 = U | V;
wrap1 = U | V;
AddressU[0] = wrap;
AddressV[0] = wrap;
AddressU[1] = wrap;
AddressV[1] = wrap;
MinFilter[0] = Linear;
MagFilter[0] = Linear;
MinFilter[1] = Linear;
MagFilter[1] = Linear;
VertexShader =
decl
{
    stream 0;
    float v0[3]; // Position
    float v3[3]; // Normal
    float v7[3]; // Texture Coord1
    float v8[3]; // Tangent
}
asm
{
    vs.1.1
    // Transform position
    m4x4 oPos,v0,c4
    // Transform normal and tangent
    m3x3 r7,v8,c0
    m3x3 r8,v3,c0
    // Cross product
    mul r0,-r7.zxyw,r8.yzxw;
    mad r5, -r7.yzxw, r8.zxyw, -r0;
    // Transform position
    m4x4 r2,v0,c0
    // Get a vector toward the camera
    add r2,-r2,c24
    dp3 r11.x,r2.xyz,r2.xyz // Load the square into r11
```

```
rsg r11.xyz,r11.x // Get the inverse of the squa
            mul r2.xyz,r2.xyz,r11.xyz // Multiply, r0 = -(camera vec
            add r2.xyz,r2.xyz,-c16 // Get half angle
            // Normalize
            dp3 r11.x,r2.xyz,r2.xyz // Load the square into r1
            rsq r11.xyz,r11.x // Get the inverse of the squa
            mul r2.xyz,r2.xyz,r11.xyz // Multiply, r2 = HalfAngle
            // Transform the half angle vector
            dp3 r6.x,r7,r2
            dp3 r6.y,r5,r2
            dp3 r6.z,r8,r2
            // Multiply by a half to bias, then add half
            mad r6.xyz,r6.xyz,c20,c20
            mov oT0.xy,v7.xy
            mov oT1.xy,v7.xy
            mov oD0.xyz,r6.xyz
        };
        PixelShader =
        asm
        {
            ps.1.1
                                   // Sample base map
            tex t0
                                   // Sample normal
            tex t1
            dp3 r0,t1_bx2,v0_bx2; // Dot(normal,half)
            mul r1,r0,r0; // Raise it to 32nd power
            mul r0,r1,r1;
            mul r1,r0,r0;
            mul r0,r1,r1;
            mul r0,t0,r0
        };
   }
technique tec2
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = ; // World Matrix
VertexShaderConstant[4] = ; // World*View*Proj Matrix
```

{

```
// Material properties of object
VertexShaderConstant[9] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffuse
VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
// Properties of light
VertexShaderConstant[13] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffuse
VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
VertexShaderConstant[16] = ;
                                             //Light Directio
vertexShaderConstant[20] = (.5f, .5f, .5f, .5f);
// Camera Information
VertexShaderConstant[24] = ;
Texture[0]
             = ;
             = ;
Texture[2]
             = ;
Texture[3]
             = U | V;
wrap0
             = 0;
wrap1
wrap2
             = 0;
             = U | V;
wrap3
AddressU[0] = wrap;
AddressV[0] = wrap;
AddressU[1] = clamp;
AddressV[1] = clamp;
AddressU[2] = clamp;
AddressV[2] = clamp;
MinFilter[0] = Linear;
MagFilter[0] = Linear;
MinFilter[1] = Linear;
MagFilter[1] = Linear;
VertexShader =
decl
{
    stream 0;
    float v0[3]; // Position
    float v3[3]; // Normal
    float v7[3]; // Texture Coord1
    float v8[3]; // Tangent
}
asm
{
    vs.1.1
```

```
// Transform position
   m4x4 oPos,v0,c4
    // Transform normal and tangent
   m3x3 r7,v8,c0
   m3x3 r8,v3,c0
    // Cross product
   mul r0, -r7.zxyw, r8.yzxw;
   mad r5, -r7.yzxw, r8.zxyw, -r0;
    // Transform position
   m4x4 r2,v0,c0
    // Get a vector toward the camera
   add r2,-r2,c24
    dp3 r11.x,r2.xyz,r2.xyz // Load the square into r11
    rsq r11.xyz,r11.x // Get the inverse of the squa
   mul r2.xyz,r2.xyz,r11.xyz // Multiply, r0 = -(camera vec
   add r2.xyz,r2.xyz,-c16 // Get half angle
    // Normalize
   dp3 r11.x,r2.xyz,r2.xyz // Load the square into r1
    rsq r11.xyz,r11.x // Get the inverse of the squa
   mul r2.xyz,r2.xyz,r11.xyz // Multiply, r2 = HalfAngle
    // Transform the half angle vector
    dp3 r6.x,r7,r2
    dp3 r6.y,r5,r2
   dp3 r6.z,r8,r2
   mov oT0.xy,v7.xy // Coordinate to samp normal fr
                           // Not a tex coordinate, but ha
   mov oT1.xyz,r6
   mov oT2.xyz,r6
                            // Angle
   mov oT3.xy,v7.xy
PixelShader =
asm
    ps.1.1
    tex t0
                               // Sample normal
   texm3x2pad t1, t0_bx2 // Look it up in a table
    texm3x2tex t2, t0 bx2
    tex t3
                           // Sample base color
   mov r0,t2
```

};

{

mul r0,r0,t3 // Blend terms }; } }

Exercise 10: Anisotropic Bump Mapping

```
11
// Effect File Workshop Exercise 10
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
vector lhtR;
                  // Light direction
matrix mWld; // World
matrix mTot; // Total
texture tDif; // Diffuse texture of object
texture tDf4; // Normal map for earth
texture tSt2; // Anisotropic lighting table
vector vCPS; // Camera position
// Background color
DWORD BCLR = 0 \times FF0000FF;
pixelshader pNIL;
string XFile = "bust.x";
// Technique names for display in viewer window
string tec0 = "Exercise 10: Anisotropic Bump Mapping";
technique tec0
{
     pass p0
     {
         // Load matrices
         VertexShaderConstant[0] = <mWld>;
                                                                    // World M
                                                                    // World*V
         VertexShaderConstant[4] = <mTot>;
         // Material properties of object
         VertexShaderConstant[9] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffuse
         VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
         VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
         // Properties of light
         VertexShaderConstant[13] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffuse
         VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specula
```

```
VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambient
VertexShaderConstant[16] = ;
                                                    // Light d
vertexShaderConstant[20] = (.5f,.5f,.5f,.5f);
vertexShaderConstant[40] = (1.0f, 1.0f, 1.0f, 1.0f);
// Camera information
VertexShaderConstant[24] = <vCPS>;
PixelShaderConstant[0] = (0.5f, 0.2f, 0.2f, 0.2f);
Texture[0] = \langle tDf4 \rangle;
Texture[3] = <tSt2>;
wrap0
            = U | V;
wrap1
             = 0;
            = 0;
wrap2
             = 0;
wrap3
AddressU[0] = wrap;
AddressV[0] = wrap;
AddressU[1] = wrap;
AddressV[1] = wrap;
AddressU[3] = wrap;
AddressV[3] = wrap;
MinFilter[0] = Linear;
MagFilter[0] = Linear;
MinFilter[1] = Linear;
MagFilter[1] = Linear;
VertexShader =
decl
{
    stream 0;
    float v0[3];
                              // Position
                             // Normal
    float v3[3];
    float v7[3];
                             // Texture Coord1
                              // Tangent
    float v8[3];
}
asm
{
    vs.1.1
    // Transform position
    m4x4 oPos,v0,c4
    // Transform normal and tangent
    m3x3 r7,v8,c0
    m3x3 r8,v3,c0
```

```
// Cross product
    mul r0, -r7.zxyw, r8.yzxw;
    mad r5, -r7.yzxw, r8.zxyw, -r0;
    // Transform position
    m4x4 r2,v0,c0
    // Get a vector toward the camera
    add r2,-r2,c24
    dp3 r11.x,r2.xyz,r2.xyz // Load the square into r11
    rsq r11.xyz,r11.x // Get the inverse of the squa
    mul r2.xyz,r2.xyz,r11.xyz // Multiply, r0 = -(camera vec
     // Transform the view angle vector
    dp3 r6.x,r7,r2
    dp3 r6.y,r5,r2
    dp3 r6.z,r8,r2
    // Transform the light vector
    dp3 r2.x,r7,-c16
    dp3 r2.y,r5,-c16
    dp3 r2.z,r8,-c16
                           // Coordinates to samp normal
// Light
// View angle
    mov oT0.xy,v7.xy
    mov oT1.xyz,r2
    mov oT2.xyz,r6
                            // Garbage in this register
    mov oT3.xyz,c40
};
PixelShader =
asm
{
   ps.1.1
   tex t0
                           // 3x3 transform
   texm3x3pad t1, t0_bx2
                              // These generate a texcoord w
   texm3x3pad t2, t0_bx2
   texm3x3tex t3, t0_bx2
                              // u = dot(light,normal)
                              // v = dot(view, normal)
                              // w = some positive number
   mov r0,t3;
   mad r0,c0,t3.a,r0;
                              // Alpha has the diffuse, so a
                              // to specular for final resul
   };
```

Exercise 11: Area Lighting

```
11
// Effect File Workshop Exercise 11
// Copyright (c) 2000 Microsoft Corporation. All rights reserved.
11
vector lhtR; // Light Direction
vector matD; // Material Diffuse
               // Material Diffuse
matrix mWld; // World
matrix mTot; // Total
texture tDif; // Diffuse texture of object
// Background color
DWORD BCLR = 0 \times FF000000;
pixelshader pNIL;
string XFile = "skullhiv.x";
// Technique names for display in viewer window
string tec0 = "Exercise 11a: Area Lighting";
string tec1 = "Exercise 11b: Area and Diffuse Lighting";
technique tec0
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = <mWld>;
                                                             // World
        VertexShaderConstant[4] = <mTot>;
                                                               // World
        // Material properties of object
        VertexShaderConstant[9] = <matD>;
                                                              // Diffu
        VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f); // Specu
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambie
        // Properties of light
        VertexShaderConstant[13] = (1.0f, 1.0f, 1.0f, 1.0f); // Diffu
        VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f); // Specu
        VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f); // Ambie
        VertexShaderConstant[16] = <lhtR>;
                                                               // Light
```

```
VertexShaderConstant[20] = (.5f,.5f,.5f,.5f);
VertexShaderConstant[40] = (1.0f, 1.0f, 1.0f, 1.0f);
VertexShaderConstant[41] = (1.00f, 0.86f, 0.75f, 1.0f); // sky
VertexShaderConstant[42] = (0.25f, .25f, 0.15f, 1.0f); // grou
                                                         // obje
VertexShaderConstant[43] = <matD>;
VerteXShaderConstant[44] = (0.0f, -1.0f, 0.0f, 1.0f); // sky
   Texture[0] = <tDif>;
             = U | V;
wrap0
wrap1
              = 0;
              = 0;
wrap2
AddressU[0] = wrap;
AddressV[0] = wrap;
MinFilter[0] = Linear;
MagFilter[0] = Linear;
MinFilter[1] = Linear;
MagFilter[1] = Linear;
ColorOp[0] = Modulate;
ColorArg1[0] = Diffuse;
ColorArg2[0] = Texture;
AlphaOp[0] = SelectArg1;
AlphaArg1[0] = Diffuse;
VertexShader =
decl
{
    stream 0;
                       // Position
// Normal
// Texture Coord1
// Tangent
    float v0[3];
    float v3[3];
    float v7[3];
    float v8[3];
}
asm
{
    vs.1.1
    m4x4 oPos,v0,c4 // Transform position
m3x3 r0,v3,c0 // Transform normal to world space
    dp3 r0,r0,-c44
                          // Dot product against sky vector
    // The dot product was between -1 to 1. We want
         to re-range this to 0 to 1
    //
    mad r0,r0,c20,c20
    // Now lerp between the two sky colors.
    mov r1,c42
```

```
// To save a clock, this delta shou
            sub r1,c41,r1
                                  // saved in a register
            mad r0,r1,r0,c42
                                  // Now lerp
                                  // This modulates against a darkeni
            sub r1,c40,v7.zzz
                                       or occlusion term, embedded in
                                  11
                                       coordinate, we want 1-z though
                                  11
                                  // Darken sky term
            mul r0,r0,r1
                                  // Now modulate against object's co
            mul oD0,r0,c9
            mov oT0.xy,v7.xy
        };
    }
}
technique tec1
{
    pass p0
    {
        // Load matrices
        VertexShaderConstant[0] = <mWld>;
                                                               // Worl
        VertexShaderConstant[4] = <mTot>;
                                                                // Worl
        //Material properties of object
        VertexShaderConstant[9] = <matD>;
                                                               // Diff
        VertexShaderConstant[10] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                               // Spec
        VertexShaderConstant[11] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                                // Ambi
        //properties of light
        VertexShaderConstant[13] = (.6f, .6f, .6f, 1.0f);
                                                                // Diff
        VertexShaderConstant[14] = (0.0f, 0.0f, 0.0f, 0.0f);
                                                               // Spec
                                                               // Ambi
        VertexShaderConstant[15] = (0.0f, 0.0f, 0.0f, 0.0f);
        VertexShaderConstant[16] = <lhtR>;
                                                                // Ligh
        VertexShaderConstant[20] = (.5f,.5f,.5f,.5f);
        VertexShaderConstant[40] = (1.0f, 1.0f, 1.0f, 1.0f);
        VertexShaderConstant[41] = (.80f, 0.76f, 0.65f, 1.0f);
                                                               // Sky
        VertexShaderConstant[42] = (0.25f, .25f, 0.15f, 1.0f);
                                                               // Grou
        VertexShaderConstant[43] = <matD>;
                                                               // Obie
        VerteXShaderConstant[44] = (0.0f, -1.0f, 0.0f, 1.0f);
                                                                // Sky
        FillMode = Wireframe;
        Texture[0]
                     = <tDif>;
        wrap0
                     = U | V;
        wrap1
                     = 0;
        wrap2
                     = 0;
```

```
AddressU[0] = wrap;
AddressV[0] = wrap;
MinFilter[0] = Linear;
MagFilter[0] = Linear;
MinFilter[1] = Linear;
MagFilter[1] = Linear;
ColorOp[0] = Modulate;
ColorArg1[0] = Diffuse;
ColorArg2[0] = Texture;
Alpha0p[0] = SelectArg1;
AlphaArg1[0] = Diffuse;
VertexShader =
decl
{
    stream 0;
                      // Position
    float v0[3];
                      // Normal
    float v3[3];
                       // Texture Coord1
    float v7[3];
                 // Tangent
    float v8[3];
}
asm
{
    vs.1.1
    m4x4 oPos,v0,c4 // Transform position
m3x3 r9,v3,c0 // Transform normal to world space
    dp3 r0,r9,-c44 // Dot product against sky vector
    // The dot product was between -1 to 1. We want
    11
         to re-range this to 0 to 1
    mad r0,r0,c20,c20
    //now lerp between the two sky colors
    mov r1,c42
                        // To save a clock, this delta shoul
     sub r1,c41,r1
                        11
                             be saved in a register
                        // Now lerp
    mad r0,r1,r0,c42
                        // This modulates against a darkenin
    sub r1,c40,v7.zzz
                        11
                             or occlusion term, embedded in
                        //
                             coordinate, we want 1-z though
                        // Darken sky term
    mul r2,r0,r1
    dp3 r1,r9,-c16
                        // Dot against light vector
    max r1, c15, r1
    // Complement, square, complement
```

sub r1, c40, r1 mul r1, r1, r1 mul r1, r1, r1 sub r1, c40, r1 mad r0,r1,c13,r2 // Combine with light diffuse mul oD0,r0,c9 // Now modulate against object's col mov oT0.xy,v7.xy };

Ambient Lighting

Ambient lighting provides constant lighting for a scene. It lights all object vertices the same because it is not dependent on any other lighting factors such as vertex normals, light direction, light position, range, or attenuation. It is the fastest type of lighting but it produces the least realistic results. Microsoft® Direct3D® contains a single global ambient light property that you can use without creating any light. Alternatively, you can set any light object to provide ambient lighting. The ambient lighting for a scene is described by the following equation.

Ambient Lighting = $M_c * [G_a + sum(L_{ai})]$

The parameters are defined in the following table.

Parameter	Default value	Туре	Description
M _c	(0,0,0,0)	D3DCOLORVALUE	Material ambient color.
G _a	(0,0,0,0)	D3DCOLORVALUE	Global ambient color.
L _{ai}	(0,0,0,0)	D3DVECTOR	Light ambient color, of the ith light.
sum	N/A	N/A	Summation of the ambient light from the light objects.

The value for M_c is one of three values: one of the two possible vertex colors in a vertex declaration, or the material ambient color. The value is:

- vertex color1, if AMBIENTMATERIALSOURCE = D3DMCS_COLOR1, and the first vertex color is supplied in the vertex declaration.
- vertex color2, if AMBIENTMATERIALSOURCE = D3DMCS_COLOR2, and the second vertex color is supplied in vertex declaration.
- material ambient color

Note If either AMBIENTMATERIALSOURCE option is used, and the vertex color is not provided, then the material ambient color is used.

To use the material ambient color, use SetMaterial as shown in the example code

below.

G_a is the global ambient color. It is set using SetRenderState(D3DRENDERSTATE_AMBIENT). There is one global ambient color in a Direct3D scene. This parameter is not associated with a Direct3D light object.

 L_{ai} is the ambient color of the ith light in the scene. Each Direct3D light has a set of properties, one of which is the ambient color. The term, sum(L_{ai}) is a sum of all the ambient colors in the scene.

Example

In this example, the object is colored using the scene ambient light and a material ambient color. The code is shown below.

```
#define GRAY_COLOR 0x00bfbfbf
// create material
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl;, sizeof(D3DMATERIAL8) );
mtrl.Ambient.r = 0.75f;
mtrl.Ambient.g = 0.0f;
mtrl.Ambient.b = 0.0f;
mtrl.Ambient.a = 0.0f;
m_pd3dDevice->SetMaterial( &mtrl; );
m_pd3dDevice->SetRenderState( D3DRS_AMBIENT, GRAY_COLOR);
```

According to the equation, the resulting color for the object vertices is a combination of the material color and the light color.

These two images show the material color, which is gray, and the light color, which is bright red.

The resulting scene is shown below. The only object in the scene is a sphere. Ambient light lights all object vertices with the same color. It is not dependent on the vertex normal or the light direction. As a result, the sphere looks like a 2-D circle because there is no difference in shading around the surface of the object.

To give objects a more realistic look, apply diffuse or specular lighting in addition to ambient lighting.

Diffuse Lighting

After adjusting the light intensity for any attenuation effects, Microsoft® Direct3D® computes how much of the remaining light reflects from a vertex, given the angle of the vertex normal and the direction of the incident light. Direct3D skips to this step for directional lights because they do not attenuate over distance. The system considers two reflection types, diffuse and specular, and uses a different formula to determine how much light is reflected for each. After calculating the amounts of light reflected, Direct3D applies these new values to the diffuse and specular reflectance properties of the current material. The resulting color values are the diffuse and specular components that the rasterizer uses to produce Gouraud shading and specular highlighting.

Diffuse lighting is described by the following equation.

Diffuse Lighting = $sum[V_d^*L_d^*(N \cdot L_{dir})^*Atten^*Spot]$

Parameter	Default value	Туре	Description
sum	N/A	N/A	Summation of each light's diffuse component.
V _d	(0,0,0,0)	D3DCOLORVALUE	Vertex diffuse color.
L _d	(0,0,0,0)	D3DCOLORVALUE	Light diffuse color.
Ν	N/A	D3DVECTOR	Vertex normal.
L _{dir}	(0,0,0,0)	D3DCOLORVALUE	Direction vector from object vertex to the light.
Atten	(0,0,0,0)	D3DCOLORVALUE	Light attenuation.
Spot	(0,0,0,0)	D3DVECTOR	Characteristics of the spotlight cone.

The parameters are defined in the following table.

The value for V_d is one of three values: one of the two possible vertex colors in a vertex declaration, or the material diffuse color. The value is:

• vertex color1, if DIFFUSEMATERIALSOURCE = D3DMCS_COLOR1, and the first vertex color is supplied in the vertex declaration.

- vertex color2, if DIFFUSEMATERIALSOURCE = D3DMCS_COLOR2, and the second vertex color is supplied in the vertex declaration.
- material diffuse color

Note: If either DIFFUSEMATERIALSOURCE option is used, and the vertex color is not provided, the material diffuse color is used.

To calculate the attenuation (Atten) or the spotlight characteristics (Spot), see <u>Attenuation and Spotlight Terms</u>

Diffuse components are clamped to be from 0 to 255, after all lights are processed and interpolated separately. The resulting diffuse lighting value is a combination of the ambient, diffuse and emissive light values.

Example

In this example, the object is colored using the light diffuse color and a material diffuse color. The code is shown below.

```
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl;, sizeof(D3DMATERIAL8) );
D3DLIGHT8 light;
ZeroMemory( &light;, sizeof(D3DLIGHT8) );
light.Type = D3DLIGHT_DIRECTIONAL;
D3DXVECTOR3 vecDir;
vecDir = D3DXVECTOR3(0.5f, 0.0f, -0.5f);
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction;, &vecDir; );
// set directional light diffuse color
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
light.Diffuse.a = 1.0f;
m_pd3dDevice->SetLight( 0, &light; );
m_pd3dDevice->LightEnable( 0, TRUE );
// if a material is used, SetRenderState must be used
// vertex color = light diffuse color * material diffuse color
mtrl.Diffuse.r = 0.75f;
mtrl.Diffuse.g = 0.0f;
mtrl.Diffuse.b = 0.0f;
mtrl.Diffuse.a = 0.0f;
```

```
m_pd3dDevice->SetMaterial( &mtrl; );
m_pd3dDevice->SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_MAT
```

According to the equation, the resulting color for the object vertices is a combination of the material color and the light color.

These two images show the material color, which is gray, and the light color, which is bright red.

The resulting scene is shown below. The only object in the scene is a sphere. The diffuse lighting calculation takes the material and light diffuse color and modifies it by the angle between the light direction and the vertex normal using the dot product. As a result, the backside of the sphere gets darker as the surface of the sphere curves away from the light.

Combining the diffuse lighting with the ambient lighting from the previous example shades the entire surface of the object. The ambient light shades the entire surface and the diffuse light helps reveal the object's three-dimensional (3-D) shape.

Diffuse lighting is more intensive to calculate than ambient lighting. Because it depends on the vertex normals and light direction, you can see the objects geometry in 3-D space, which produces a more realistic lighting than ambient lighting. You can use specular highlights to achieve a more realistic look.

Specular Lighting

Modeling specular reflection requires that the system not only know the direction that light is traveling, but also the direction to the viewer's eye. The system uses a simplified version of the Phong specular-reflection model, which employs a halfway vector to approximate the intensity of specular reflection.

The default lighting state does not calculate specular highlights. To enable specular lighting, be sure to set the **D3DRS_SPECULARENABLE** to TRUE.

Specular Lighting is described by the following equation.

Specular Lighting = $V_s^* sum[L_s^*(N \cdot H)^{P*}Atten^*Spot]$

The parameters are defined in the following table.

Parameter Default value Type	Description
---------------------------------	-------------

		Summatio
	Vertex	of each
Vs	(0,0,0,0) D3DCOLORVALUE specular	sum N/A N/A light's
	color.	specular
		componen

The value for V_c is one of three values: one of the two possible vertex colors in a vertex declaration, or the material specular color. The value is:

- vertex color1, if SPECULARMATERIALSOURCE = D3DMCS_COLOR1, and the first vertex color is supplied in the vertex declaration.
- vertex color2, if SPECULARMATERIALSOURCE = D3DMCS_COLOR2, and the second vertex color is supplied in the vertex declaration.
- material specular color

Note: If either SPECULARMATERIALSOURCE option is used, and the vertex color is not provided, then the material specular color is used.

To calculate the attenuation (Atten) or the spotlight characteristics (Spot), see <u>Attenuation and Spotlight Terms</u>

The halfway vector (H) exists midway between the vector from an object vertex to the light source and the vector from an object vertex and the camera position. Microsoft® Direct3D® provides two ways to compute the halfway vector. When D3DRS_LOCALVIEWER is set to TRUE, the system calculates the halfway vector using the position of the camera and the position of the vertex, along with the light's direction vector. The following formula illustrates this.

H = norm(norm($C_p - V_p$) + L_{dir}) where the parameters are defined in the following table:

Parameter Default Type Description

 C_p (0,0,0,0) D3DVECTOR $\frac{\text{Camera}}{\text{position.}}$ V_p (0,0,0,0) D3DVECTOR $\frac{\text{Vert}}{\text{position}}$

When **D3DRS_LOCALVIEWER** is set to TRUE, Direct3D determines the halfway vector by the following formula.

```
H = norm(norm(- V_p) + L_{dir})
```

Determining the halfway vector in this manner can be computationally intensive. As an alternative, setting **D3DRS_LOCALVIEWER** to FALSE instructs the system to act as though the viewpoint is infinitely distant on the z-axis. This setting is less computationally intensive, but much less accurate, so it is best used by applications that use orthogonal projection.

Specular components are clamped to be from 0 to 255, after all lights are processed and interpolated separately.

Example

In this example, the object is colored using the scene specular light color and a material specular color. The code is shown below.

```
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl;, sizeof(D3DMATERIAL8) );
D3DLIGHT8 light;
ZeroMemory( &light;, sizeof(D3DLIGHT8) );
light.Type = D3DLIGHT_DIRECTIONAL;
D3DXVECTOR3 vecDir;
vecDir = D3DXVECTOR3(0.5f, 0.0f, -0.5f);
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction;, &vecDir; );
light.Specular.r = 1.0f;
light.Specular.g = 1.0f;
light.Specular.b = 1.0f;
light.Specular.a = 1.0f;
light.Range = 1000;
light.Falloff = 0;
light.Attenuation0 = 1;
light.Attenuation1 = 0;
light.Attenuation2 = 0;
m_pd3dDevice->SetLight( 0, &light; );
m_pd3dDevice->LightEnable( 0, TRUE );
m_pd3dDevice->SetRenderState( D3DRS_SPECULARENABLE, TRUE );
mtrl.Specular.r = 1.0f;
mtrl.Specular.g = 1.0f;
mtrl.Specular.b = 1.0f;
mtrl.Specular.a = 1.0f;
mtrl.Power = 20;
m_pd3dDevice->SetMaterial( &mtrl; );
m_pd3dDevice->SetRenderState(D3DRS_SPECULARMATERIALSOURCE, D3DMCS_MA
```

According to the equation, the resulting color for the object vertices is a combination of the material color and the light color.

These two images show the material color, which is gray, and the light color, which is white.

The resulting specular highlight is shown below.

Combining the specular highlight with the ambient and diffuse lighting produces the following image. With all three types of lighting applied, this more clearly resembles a realistic object.

Specular lighting is more intensive to calculate than diffuse lighting. It is typically used to provide visual clues about the surface material. The specular highlight varies in size and color with the material of the surface.

Emissive Lighting

Emissive lighting is described by a single term.

Emissive Lighting = M_e

The parameter is defined in the following table.

Parameter	Default value	Туре	Description
M _e	(0,0,0,0)	D3DCOLORVALUE	Material emissive color.

The value for M_e is one of three values: one of the two possible vertex colors in a vertex declaration, or the material emissive color. The value is:

- vertex color1, if EMISSIVEMATERIALSOURCE = D3DMCS_COLOR1, and the first vertex color is supplied in the vertex declaration.
- vertex color2, if EMISSIVEMATERIALSOURCE = D3DMCS_COLOR2, and the second vertex color is supplied in the vertex declaration.
- material emissive color

Note If either EMISSIVEMATERIALSOURCE option is used, and the vertex color is not provided, the material emissive color is used.

Example

In this example, the object is colored using the scene ambient light and a material ambient color. The code is shown below.

```
// create material
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl;, sizeof(D3DMATERIAL8) );
mtrl.Emissive.r = 0.0f;
mtrl.Emissive.g = 0.75f;
mtrl.Emissive.b = 0.0f;
mtrl.Emissive.a = 0.0f;
m_pd3dDevice->SetMaterial( &mtrl; );
m_pd3dDevice->SetRenderState(D3DRS_EMISSIVEMATERIALSOURCE, D3DMCS_MA
```

According to the equation, the resulting color for the object vertices is the material color.

The image below shows the material color, which is green. Emissive light lights all object vertices with the same color. It is not dependent on the vertex normal or the light direction. As a result, the sphere looks like a 2-D circle because there is no difference in shading around the surface of the object.

This image shows how the emissive light blends with the other three types of lights, from the previous examples. On the right side of the sphere, there is a blend of the green emissive and the red ambient light. On the left side of the sphere, the green emissive light blends with red ambient and diffuse light producing a red gradient. The specular highlight is white in the center and creates a yellow ring as the specular light value falls off sharply leaving the ambient, diffuse and emissive light values which blend together to make yellow.

Camera Space Transformations

Vertices in the camera space are computed by transforming the object vertices with the world view matrix.

V = V * wvMatrix

Vertex normals, in camera space, are computed by transforming the object normals with the inverse transpose of the world view matrix. The world view matrix may or may not be symmetrical.

 $N = N * (wvMatrix^{-1})^{T}$

The matrix inversion and matrix transpose operate on a 4×4 matrix. The multiply combines the normal with the 3×3 portion of the resulting 4×4 matrix.

If the render state, *D3DRENDERSTATE_NORMALIZENORMALS* is set to TRUE, vertex normal vectors are normalized after transformation to camera space as follows:

N = norm(N)

Light position in camera space is computed by transforming the light source position with the view matrix.

Lp = Lp * vMatrix

The direction to the light in camera space for a directional light is computed by multiplying the light source direction by the view matrix, normalizing, and negating the result.

L_{dir} = -norm(L_{dir} * vMatrix)

For the D3DLIGHT_POINT and D3DLIGHT_SPOT the direction to light is computed as follows:

 L_{dir} = norm(L_{dir}), where the parameters are defined in the following table.

Parameter	Default value	Туре	Description
L _{dir}	(0,0,0,0)	D3DCOLORVALUE	Direction vector from object vertex to the light
V	(0,0,0,0)	D3DVECTOR	Vertex position
wvMatrix	Identity	D3DMATRIX	Composite matrix containing the world and view transforms
Ν	(0,0,0,0)	D3DVECTOR	Vertex normal
Lp	(0,0,0,0)	D3DVECTOR	Light position
vMatrix	Identity	D3DMATRIX	Matrix containing the view transform

Attenuation and Spotlight Terms

The diffuse and specular lighting components of the global illumination equation contain terms that describe light attenuation and the spotlight cone. These terms are described below.

Attenuation Term

The attenuation of a light depends on the type of light and the distance between the light and the vertex position. To calculate attenuation, use one of the following three equations.

- Atten = 1, if the light is a directional light.
- Atten = 0, if the distance between the light and the vertex exceeds the light's range.
- Atten = $1/(att0_i + att1_i*d + att2_i*di^2)$.

Parameter	Default value	Туре	Description
att0i	(0,0,0,0)	FLOAT	Linear attenuation factor
att1i	(0,0,0,0)	FLOAT	Squared attenuation factor
att2i	(0,0,0,0)	FLOAT	Exponential attenuation factor
di	(0,0,0,0)	FLOAT	Distance from vertex position to light position

The att0, att1, att2 values are specified by the **Attenuation0**, **Attenuation1**, and **Attenuation2** members of the **D3DLIGHT8** structure.

The distance between the light and the vertex position is always pos di = $||L_{dir}||$ where:

Parameter	Default value	Туре	Description
			Direction vector from
L _{dir}	0.0	D3DVECTOR	vertex position to the

light position

If *di* is greater than the light's range, that is, the **Range** member of a **D3DLIGHT8** structure, Direct3D makes no further attenuation calculations and applies no effects from the light to the vertex. The *dvAttenuation0*, *dvAttenuation1*, and *dvAttenuation2* values are the light's attenuation constants as specified by the members of a light object's **D3DLIGHT8** structure. The corresponding structure members are **Attenuation0**, **Attenuation1**, and **Attenuation2**.

The attenuation constants act as coefficients in the formula—you can produce a variety of attenuation curves by making simple adjustments to them. You can set **Attenuation0** to 1.0 to create a light that doesn't attenuate but is still limited by range, or you can experiment with different values to achieve various attenuation effects.

The attenuation at the maximum range of the light is not 0.0. To prevent lights from suddenly appearing when they are at the light range, an application can increase the light range. Or, the application can set up attenuation constants so that the attenuation factor is close to 0.0 at the light range. The attenuation value is multiplied by the red, green, and blue components of the light's color to scale the light's intensity as a factor of the distance light travels to a vertex.

Spotlight Term

Parameter	Default value	Туре	Description
rho	0.0	N/A	Angle
phi	0.0	FLOAT	Penumbra angle of spotlight in radians. Range: [thetaI, p)
theta	0.0	FLOAT	Umbra angle of spotlight in radians. Range: [0, p)
falloff	0.0	FLOAT	Falloff factor. Range: (- infinity, +infinity)

where:

 $rho = norm(L_{dcs}) \cdot norm(L_{dir})$

Parameter	Default value	Туре	Description
L _{dcs}	0.0	D3DVECTOR	Direction vector from origin to the light position in camera space
L _{dir}	0.0	D3DVECTOR	Direction vector from vertex position to the light position

After computing the light attenuation, Direct3D also considers: spotlight effects if applicable, the angle that the light reflects from a surface, and the reflectance of the current material to calculate the diffuse and specular components for that vertex. For more information, see <u>Spotlight Model</u>.

```
Microsoft DirectX 8.1 (C++)
```

Preparing the Action Map

The action map is a **DIACTIONFORMAT** structure containing information about application actions and their mapping to virtual controls or device objects. The structure is passed back and forth between the application and Microsoft® DirectInput® to establish the final mapping. This section explains how to initialize the map.

1. Define Application Actions

The first step in implementing DirectInput action mapping is to determine what input-driven actions in your application need to be mapped to device objects. For actions that can be performed either by an axis or by a button, you must define separate actions for both input types. It is recommended that you define button actions for all important functions, in case the device does not have the appropriate axes.

The following sample enumeration of action values might be defined by a carracing game. Axis actions begin with "eA" and button actions with "eB".

```
enum eGameActions
{
    eA_STEER, // Steering
    eB_STEER_LEFT, // Steer left
    eB_STEER_RIGHT, // Steer right
    eA_ACCELERATE, // Change speed
    eB_ACCELERATE, // Speed up
    eB_DECELERATE, // Slow down
    eA_BRAKE, // Brake
    eB_BRAKE, // Brake
    eB_UPSHIFT, // Shift to higher gear
    eB_DOWNSHIFT, // Shift to lower gear
    eB_CYCLEVIEW, // Cycle to next view
    eB_COURSEVIEW, // Toggle course view
    eB_DRIVERVIEW, // View from driver's seat
    eB_BRAKEBIAS, // Brake bias
    eA_VOLUME, // Sound volume
    eB_MUTE // Toggle sound
}
```

#define NUM_MY_ACTIONS 16

In the example, actions are defined as enumerated values. However, they could be other 32-bit data types, such as pointers to functions. When you retrieve device data, you get whatever action value you have defined, and you can handle it in any way you like.

2. Define the Genre

The next step is to decide what genre your application belongs to. A genre defines a set of virtual controls. By selecting the proper genre, you can obtain the best possible fit of virtual controls to application actions. Manufacturers who choose to supply default mappings for their devices must support one or more of the genres defined by DirectInput. See <u>Action Mapping Constants</u> for a list of these genres.

For the game in the example, the obvious choice is the DIVIRTUAL_DRIVING_RACE genre, which contains the following virtual controls.

Priority 1 Controls

DIAXIS_DRIVINGR_STEER DIAXIS_DRIVINGR_ACCELERATE DIAXIS_DRIVINGR_BRAKE DIBUTTON_DRIVINGR_SHIFTUP DIBUTTON_DRIVINGR_SHIFTDOWN DIBUTTON_DRIVINGR_VIEW DIBUTTON_DRIVINGR_MENU

Priority 2 Controls

DIAXIS_DRIVINGR_ACCEL_AND_BRAKE DIHATSWITCH_DRIVINGR_GLANCE DIBUTTON_DRIVINGR_ACCELERATE_LINK DIBUTTON_DRIVINGR_AIDS DIBUTTON_DRIVINGR_BOOST DIBUTTON_DRIVINGR_BRAKE DIBUTTON_DRIVINGR_DASHBOARD DIBUTTON_DRIVINGR_DEVICE DIBUTTON_DRIVINGR_GLANCE_LEFT_LINK DIBUTTON_DRIVINGR_GLANCE_RIGHT_LINK DIBUTTON_DRIVINGR_MAP DIBUTTON_DRIVINGR_PAUSE DIBUTTON_DRIVINGR_PIT DIBUTTON_DRIVINGR_STEER_LEFT_LINK DIBUTTON_DRIVINGR_STEER_RIGHT_LINK

There is no difference in functionality between Priority 1 and Priority 2 controls. Priority 1 controls are those most likely to be supported by device manufacturers in their default mappings. However, there is no guarantee that any virtual control will be supported by a device.

3. Assign Actions to Controls or Device Objects

The next step in creating the action map is to associate each application action with one or more of the virtual controls defined for the genre. You do this by declaring and initializing an array of **DIACTION** structures. Each structure in the array specifies the action value, the virtual control to associate with it, and a friendly name that describes the action. Leave other members as zero; they will be filled in later by DirectInput.

You can also use elements of the **DIACTION** array to map actions to particular keys or buttons on the keyboard or mouse or to channels on a Microsoft DirectPlay® voice device. By doing so, you can take advantage of the simplified input loop for all input, not just that from virtual controls. For example, suppose you map the application-defined action eB_UPSHIFT to both the DIBUTTON_DRIVINGR_SHIFTUP virtual control and to the Page Up key. When retrieving data, you get back eB_UPSHIFT whether the input came from a joystick button or the keyboard.

The following example declares an action map for the car-racing game.

```
DIACTION rgActions[]=
{
//Genre-defined virtual axes
   {eA_STEER, DIAXIS_DRIVINGR_STEER, 0, "Steer", },
   {eA_ACCELERATE, DIAXIS_DRIVINGR_ACCELERATE, 0, "Accelerate", },
```

{eA_BRAKE, DIAXIS_DRIVINGR_BRAKE, 0, "Brake", },

//Genre-defined virtual buttons

<pre>{eB_UPSHIFT,</pre>	DIBUTTON_DRIVINGR_SHIFTUP,	Θ,	"Upshift",	},
<pre>{eB_DOWNSHIFT,</pre>	DIBUTTON_DRIVINGR_SHIFTDOWN,	Θ,	"DownShift",	},
{eB_CYCLEVIEW,	DIBUTTON_DRIVINGR_VIEW,	Θ,	"Change View"	,},

// Actions not defined in the genre that can be assigned to any
// button or axis

{eA_VOLUME,	DIAXIS_ANY_1,	0,	"Volume",	}
{eB_MUTE,	<pre>DIBUTTON_ANY(0),</pre>	0,	"Toggle Sound",	, }

// Actions not defined in the genre that must be assigned to
// particular keys

<pre>{eB_DRIVERVIEW,</pre>	DIKEYBOARD_1,	0, "Driver View",},
<pre>{eB_COURSEVIEW,</pre>	DIKEYBOARD_C,	0, "Course View",},
<pre>{eB_BRAKEBIAS,</pre>	DIKEYBOARD_B,	0, "Brake Bias", },

// Actions mapped to keys as well as to virtual controls

{eB_UPSHIFT,	DIKEYBOARD_PRIOR,	0, "Upshift", }		
{eB_DOWNSHIFT,	DIKEYBOARD_NEXT,	0, "Downshift", }		
{eB_STEER_LEFT,	DIKEYBOARD_LEFT,	0, "Steer Left", }		
{eB_STEER_RIGHT,	DIKEYBOARD_RIGHT,	0, "Steer Right",}		
{eB_ACCELERATE,	DIKEYBOARD_UP,	0, "Accelerate", }		
{eB_DECELERATE,	DIKEYBOARD_DOWN,	0, "Decelerate", }		
{eB_BRAKE,	DIKEYBOARD_END,	0, "Brake", }		
// Actions mapped to buttons as well as to virtual controls and keys				
{eB_UPSHIFT,	DIMOUSE_BUTTON0,	0, "Upshift", },		
{eB_DOWNSHIFT,	DIMOUSE_BUTTON1,	0 , "Downshift", $\frac{1}{2}$,		
};	_ ,	. , , , , , , , , , , , , , , , , , , ,		

In the example, some actions are mapped to actual keys by using <u>Keyboard</u> <u>Mapping Constants</u>. Similar mappings to the mouse buttons and axes can be made by using <u>Mouse Mapping Constants</u>.

The **DIACTION** array is contained within a **DIACTIONFORMAT** structure that also contains information about the genre, the application, and the desired scaling of axis data. Use the same instance of this structure throughout the action mapping process. Some members will not be used immediately, but you can fill in the entire structure before the next step, <u>Finding Matching Devices</u>.

Finding Matching Devices

After you define the application actions and the virtual controls or device objects to which these actions are to be mapped, the next step is to enumerate devices on the system to find those that best support the desired virtual controls.

To do so, pass the **DIACTIONFORMAT** structure to **IDirectInput8::EnumDevicesBySemantics**. This method works in much the same way as **IDirectInput8::EnumDevices** and takes a similar callback function.

Devices that have been configured by the user to match certain controls are always enumerated first. For example, if a user has configured a wheel as the primary steering device for driving games, then the wheel is enumerated first whenever devices that support DIAXIS_DRIVINGR_STEER are requested, taking precedence over other capable devices such as joysticks that have not been configured by the user. Otherwise, the order in which available devices are enumerated is determined by the degree to which they match the requested controls. However, the order in which devices are enumerated by Microsoft® DirectInput® is not guaranteed.

In the enumeration callback, you can retrieve the default action mapping for each device, change any mappings you don't like, give the user an opportunity to reconfigure the device, and apply the action map. These steps are covered in <u>Configuring the Action Map</u>. Flags returned in the <u>DIEnumDevicesBySemanticsCallback</u> will provide information about why a particular device was enumerated. These flags will indicate whether a device has been used recently, is newly installed, or will accept mappings of priority 1 or priority 2 controls.

Configuring the Action Map

As each device is enumerated, you can obtain a pointer to it, retrieve the default action map, make changes in the default map, and apply the final mappings.

1. Obtaining the Device

Obtain the **IDirectInputDevice8** interface pointer for each enumerated device from the *lpdid* parameter of the enumeration callback. See **DIEnumDevicesBySemanticsCallback**. If you want to save the device interface for use in your application, call **AddRef** on the pointer and assign it to a global variable.

2. Obtaining the Default Action Map

To obtain the default action map for the device, call

IDirectInputDevice8::BuildActionMap. Microsoft® DirectInput® takes the list of virtual controls specified in your **DIACTIONFORMAT** structure and attempts to map these to physical device objects, returning the results in the same structure. You should examine the **dwHow** member of each **DIACTION** element to determine whether the control was successfully mapped. If it was, you can also ascertain what criterion was used in choosing the object—for example, configuration by the user or by the device manufacturer.

3. Making Changes to the Action Map

You now have the option of changing the default mappings, although it is not recommended that you do so. After examining the **dwSemantic** member of the **DIACTION** structure to determine which device object was mapped to an action, you can change that value. For example, if an action is mapped to DIJOFS_BUTTON9, but you want that action to be mapped to the trigger button instead, change the value to DIJOFS_BUTTON0 before applying the action map.

4. Applying the Action Map

When you are satisfied that the **DIACTIONFORMAT** structure contains suitable mappings for the device, call **IDirectInputDevice8::SetActionMap**. The value you assigned to the **uAppData** member of each **DIACTION** structure now becomes bound to the control specified in the **dwSemantic** member, which in turn is bound to a particular device object.

5. Mapping More than One Device

Repeat steps 1 though 4 for each device you want to use in your application. Suppose you want to map actions to both a joystick and the keyboard. In the racing-game example, the action defined in the game as eB_DRIVERVIEW was mapped to a keyboard key in the following element of the DIACTION array.

{eB_DRIVERVIEW, DIKEYBOARD_1, "Driver View", },

In that example, when **BuildActionMap** is called on any device that is not a keyboard, the **IHow** member of the **DIACTION** structure for that element is set to DIAH_UNMAPPED. Continue examining the **IHow** member as each device in turn is enumerated, until a value other than DIAH_UNMAPPED is returned. This indicates that the device being currently mapped is a keyboard and the action has been successfully mapped to the requested key.

Even actions that have been successfully mapped can be mapped to another device. In the example, eB_UPSHIFT is given two **DIACTION** structures, as follows:

```
{eB_UPSHIFT, DIBUTTON_DRIVINGR_SHIFTUP, 0, "Upshift", },
...
{eB_UPSHIFT, DIKEYBOARD_PRIOR, 0, "Upshift", },
```

As devices are successively enumerated, the eB_UPSHIFT action is mapped to a suitable button on one or more joysticks or other game controllers, and then again to the keyboard.

6. Displaying the Configuration

To show the user how actions have been mapped to devices, pass the

DICD_DEFAULT flag to **IDirectInput8::ConfigureDevices**. The property sheet for the device, containing a graphical representation of mappings, is displayed in view-only mode as in the following diagram. For more information on the mechanics of displaying the image, refer to the <u>Using Action Mapping</u> tutorial.

If the device manufacturer has not provided a device image, the mapping will be presented in text mode as in the following diagram.

Note Even if the cooperative level for the application is disabling the Microsoft® Windows® logo key passively through an exclusive cooperative level or actively through use of the DISCL_NOWINKEY flag, that key will be active while the default action mapping UI is displayed.

For more information about this property sheet, see <u>User Configuration of the</u> <u>Device</u>.

User Configuration of the Device

Microsoft® DirectInput® provides a property sheet that can be called from an application, enabling the user to configure devices for the application and view the current configuration. This property sheet can display various views of the device as provided by the manufacturer.

To enable user configuration, pass a **DICONFIGUREDEVICESPARAMS** structure containing a pointer to the **DIACTIONFORMAT** structure describing the desired mapping, along with the DICD_EDIT flag, to the **IDirectInput8::ConfigureDevices** method. Normally you would do this after calling **IDirectInputDevice8::BuildActionMap** on all devices that will be used in the application.

The following illustration shows a typical property sheet in edit mode.

If the device manufacturer has not provided a device image, the mapping will be presented in text mode as in the following diagram.

Note Even if the cooperative level for the application is disabling the Microsoft® Windows® logo key passively through an exclusive cooperative level or actively through use of the DISCL_NOWINKEY flag, that key will be active while the default action mapping UI is displayed.

The property page for a device lists the friendly names that were provided by you in the **lptszActionName** member of each **DIACTION** structure. If you have already called **BuildActionMap** for a device, the page also shows these names as callouts on the image of the device, with lines pointing to the device objects to which the actions have been mapped.

The user now has the opportunity to reassign game actions by first choosing a control then choosing an action from the menu. When the user closes the

property sheet, the method returns and the modifications are stored in the **DIACTIONFORMAT** structure that you passed in. You can now pass the same structure to **IDirectInputDevice8::SetActionMap** in order to implement the new mapping scheme.

Retrieving Action Data

You retrieve buffered data from action-mapped devices just as you would from unmapped devices: by calling **IDirectInputDevice8::GetDeviceData**. However, instead of identifying device objects by examining the **dwOfs** member of the **DIDEVICEOBJECTDATA** structure, you obtain the action associated with the object from the **uAppData** member. This is the same value you passed to the device in the **DIACTION** structure. It can be a simple identifier or a pointer to a function designed to handle the action.

Remember that an action can be associated with more than one device. You still have to obtain data from both devices independently, but you can use the same routine to handle the data regardless of where it comes from.

The following sample code, which might be part of the game loop in a driving simulation, retrieves data from all devices in the $g_lpIdiDevices$ array. This array contains $g_nDevices$ elements.

```
for (int iDevice = 0x0; iDevice < q_nDevices; iDevice++)</pre>
ł
    DIDEVICEOBJECTDATA didod;
    DWORD dwObjCount = 1;
    // Poll the device for data.
    g_lpDiDevices[iDevice]->Poll();
    // Retrieve the data.
    g_lpDiDevices[iDevice]->GetDeviceData( sizeof(didod),
                                             &didod,
                                             &dwObjCount, 0 );
    // Handle the actions regardless of what device returned them.
    switch(didod.uAppData)
    {
        case eA_STEER:
            SteerCar(didod.dwData);
            break;
        case eB_UPSHIFT
            if (didod.dwData & 0x80) ShiftGears(UPSHIFT);
            break;
```

```
:
.
default:
break;
}
}
```

Note Axis constants for specific genres, such as DIAXIS_DRIVINGR_STEER or DIAXIS_SPACESIM_LATERAL, are used for absolute joystick data. The action mapper attempts to map this virtual control to a device object that returns absolute data. The data returned from that device should be processed accordingly in the application. Device constants such as DIMOUSE_XAXIS, however, are expected to return relative data.

When retrieving data, each potential source of data should be processed separately to keep one device object from possibly overwriting the data from another. For instance, the following **DIACTION** structures are used in an action map to control direction.

<pre>{INPUT_LEFTRIGHT_ABS_AXIS,</pre>	DIAXIS_SPACESIM_LATERA	۹L,	0, _T("Turn"),
<pre>{INPUT_LEFTRIGHT_REL_AXIS,</pre>	DIMOUSE_XAXIS,		0, _T("Turn"),
<pre>{INPUT_TURNLEFT,</pre>	DIKEYBOARD_LEFT, 0	Э,	_T("Turn left"),
<pre>{INPUT_TURNRIGHT,</pre>	DIKEYBOARD_RIGHT, 0	9, _	_T("Turn right"),

The application's input loop processes data from these actions in the following case statement.

```
switch (adod[j].uAppData)
{
    case INPUT_LEFTRIGHT_ABS_AXIS:
        g_dwAbsLR = adod[j].dwData
        break;
    case INPUT_LEFTRIGHT_REL_AXIS:
        g_dwRelLR = adod[j].dwData;
        break;
    case INPUT_TURNLEFT:
        g_bLeft = (adod[j].dwData != 0);
        break;
    case INPUT_TURNRIGHT:
        g_bRight = (adod[j].dwData != 0)
        break;
}
```

Note that each data source is assigned to a separate variable rather than all data sources being assigned a generic "turn" variable. If they were to share a generic variable, holding down the LEFT ARROW key and then moving the joystick would cause the keyboard information to be lost. This is because the joystick data would overwrite the variable.

In addition to individual variables, there are many ways to process the data. Whatever method is used, care should be taken in the processing of data to avoid unexpectedly lost information.

Maintaining Files During Development

During a development cycle, unused and out of date .ini files may accumulate due to frequent action map changes, Microsoft® DirectX® reinstallations, multiple users, and other normal development situations. These files could possibly cause unexpected mappings or reports of "recent" (DIEDBS_RECENTDEVICE) for devices that would not be expected to return that value. For this reason, it is good practice to occasionally delete any unused .ini files. These files can be found in C:\Program Files\Common Files\DirectX\DirectInput\User Maps.

Note The procedure suggested above is meant to be performed only manually during a development cycle to ensure that the development environment is in a cleaner state. A shipping application should never delete user maps as this could result in the loss of a user's preferred settings.

IDirectInput8::ConfigureDevices

Displays property pages for connected input devices and enables the user to map actions to device controls.

```
HRESULT IDirectInput8::ConfigureDevices(
   LPDICONFIGUREDEVICESCALLBACK lpdiCallback,
   LPDICONFIGUREDEVICESPARAMS lpdiCDParams,
   DWORD dwFlags,
   LPVOID pvRefData
);
```

Parameters

lpdiCallback

Address of a callback function to be called each time the contents of the surface change. See **DIConfigureDevicesCallback**. Pass NULL if the application does not handle the display of the property sheet. In this case, Microsoft® DirectInput® displays the property sheet and returns control to the application when the user closes the property sheet. If you supply a callback pointer, you must also supply a valid surface pointer in the **IpUnkDDSTarget** member of the **DICONFIGUREDEVICESPARAMS** structure.

lpdiCDParams

Address of a **DICONFIGUREDEVICESPARAMS** structure that contains information about users and genres for the game, as well as information about how the user interface is displayed.

dwFlags

DWORD value that specifies the mode in which the control panel should be invoked. *DwFlags* must be one of the following values.

DICD_DEFAULT

Open the property sheet in view-only mode.

DICD_EDIT

Open the property sheet in edit mode. This mode enables the user to change action-to-control mappings. After the call returns, the application should assume current devices are no longer valid, release all device interfaces, and reinitialize them by calling **IDirectInput8::EnumDevicesBySemantics**.

pvRefData

Application-defined value to pass to the callback function.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following.

DIERR_INVALIDPARAM DIERR_OUTOFMEMORY

Remarks

Hardware vendors provide bitmaps and other display information for their device.

Before calling the method, an application can modify the text labels associated with each action by changing the value in the **lptszActionName** member of the **DIACTION** structure.

Configuration is stored for each user of each device for each game. The information can be retrieved by the **IDirectInputDevice8::BuildActionMap** method.

By default, acceleration is supported for these pixel formats:

A1R5G5B5

16-bit pixel format with 5 bits reserved for each color and 1 bit reserved for alpha (transparent texel).

A8R8G8B8

32-bit ARGB pixel format with alpha.

R9G8B8

24-bit RGB pixel format.

X1R5G5B5

16-bit pixel format with 5 bits reserved for each color. X8R8G8B8

32-bit RGB pixel format with 8 bits reserved for each color.

Other formats will result in color conversion and dramatically slow the frame rate.

Note Even if the cooperative level for the application is disabling the Microsoft Windows® logo key passively through an exclusive cooperative level or actively through use of the DISCL_NOWINKEY flag, that key will be active while the default action mapping UI is displayed.

Requirements

Windows NT/2000/XP: Requires Windows® 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

IDirectInput8::CreateDevice

Creates and initializes an instance of a device based on a given GUID, and obtains an **IDirectInputDevice8** interface.

```
HRESULT CreateDevice(
    REFGUID rguid,
    LPDIRECTINPUTDEVICE *lplpDirectInputDevice,
    LPUNKNOWN pUnkOuter
);
```

Parameters

rguid

Reference to (C++) or address of (C) the instance GUID for the desired input device (see Remarks). The GUID is retrieved through the **IDirectInput8::EnumDevices** method, or it can be one of the following predefined GUIDs:

GUID_SysKeyboard

The default system keyboard.

GUID_SysMouse

The default system mouse.

For the preceding GUID values to be valid, your application must define INITGUID before all other preprocessor directives at the beginning of the source file, or link to Dxguid.lib.

lplpDirectInputDevice

Address of a variable to receive the **IDirectInputDevice8** interface pointer if successful.

pUnkOuter

Address of the controlling object's **IUnknown** interface for COM aggregation, or NULL if the interface is not aggregated. Most callers pass NULL.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following:

DIERR DEVICENOTREG DIERR INVALIDPARAM DIERR NOINTERFACE DIERR NOTINITIALIZED DIERR_OUTOFMEMORY

Remarks

Calling this method with *pUnkOuter* = NULL is equivalent to creating the object by **CoCreateInstance**(&*CLSID_DirectInputDevice*, *NULL*, *CLSCTX_INPROC_SERVER*, *riid*, *lplpDirectInputDevice*) and then initializing it with **Initialize**.

Calling this method with *pUnkOuter* != NULL is equivalent to creating the object by **CoCreateInstance**(&CLSID_DirectInputDevice, punkOuter, CLSCTX_INPROC_SERVER, &IID_IUnknown, lplpDirectInputDevice). The aggregated object must be initialized manually.

Requirements

Windows NT/2000/XP: Requires Windows® 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

IDirectInput8::EnumDevices

Enumerates available devices.

```
HRESULT EnumDevices(
   DWORD dwDevType,
   LPDIENUMDEVICESCALLBACK lpCallback,
   LPVOID pvRef,
   DWORD dwFlags
);
```

Parameters

dwDevType Device type filter.

To restrict the enumeration to a particular type of device, set this parameter to a DI8DEVTYPE_* value. See **<u>DIDEVICEINSTANCE</u>**.

To enumerate a class of devices, use one of the following values.

DI8DEVCLASS_ALL All devices. DI8DEVCLASS_DEVICE All devices that do not fall into another class. DI8DEVCLASS_GAMECTRL All game controllers. DI8DEVCLASS_KEYBOARD All keyboards. Equivalent to DI8DEVTYPE_KEYBOARD. DI8DEVCLASS_POINTER All devices of type DI8DEVTYPE_MOUSE and DI8DEVTYPE_SCREENPOINTER.

lpCallback

Address of a callback function to be called once for each device enumerated. See **<u>DIEnumDevicesCallback</u>**.

pvRef

Application-defined 32-bit value to be passed to the enumeration callback

each time it is called.

dwFlags

Flag value that specifies the scope of the enumeration. This parameter can be one or more of the following values:

DIEDFL_ALLDEVICES

All installed devices are enumerated. This is the default behavior. DIEDFL ATTACHEDONLY

Only attached and installed devices.

DIEDFL_FORCEFEEDBACK

Only devices that support force feedback.

DIEDFL_INCLUDEALIASES

Include devices that are aliases for other devices.

DIEDFL_INCLUDEHIDDEN

Include hidden devices. For more information about hidden devices, see **<u>DIDEVCAPS</u>**.

DIEDFL_INCLUDEPHANTOMS Include phantom (placeholder) devices.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following error values:

DIERR_INVALIDPARAM DIERR_NOTINITIALIZED

Remarks

All installed devices can be enumerated, even if they are not present. For example, a flight stick might be installed on the system but not currently plugged into the computer. Set the *dwFlags* parameter to indicate whether only attached or all installed devices should be enumerated. If the DIEDFL_ATTACHEDONLY flag is not present, all installed devices are enumerated.

A preferred device type can be passed as a *dwDevType* filter so that only the devices of that type are enumerated.

Note The order in which devices are enumerated by DirectInput is not guaranteed.

Requirements

Windows NT/2000/XP: Requires Windows® 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

See Also

IDirectInput8::EnumDevicesBySemantics

IDirectInput8::EnumDevicesBySemar

Enumerates devices that most closely match the application-specified action map.

```
HRESULT EnumDevicesBySemantics(
  LPCTSTR ptszUserName,
  LPDIACTIONFORMAT lpdiActionFormat,
  LPDIENUMDEVICESBYSEMANTICSCB lpCallback,
  LPVOID pvRef,
  DWORD dwFlags
);
```

Parameters

ptszUserName

String identifying the current user, or NULL to specify the user logged onto the system. The user name is taken into account when enumerating devices. A device with user mappings is preferred to a device without any user mappings. By default, devices in use by other users are not enumerated for this user.

lpdiActionFormat

Address of a **<u>DIACTIONFORMAT</u>** structure that specifies the action map for which suitable devices are enumerated.

lpCallback

Address of a callback function to be called once for each device enumerated. See **<u>DIEnumDevicesBySemanticsCallback</u>**.

pvRef

Application-defined 32-bit value to pass to the enumeration callback each time it is called.

dwFlags

Flag value that specifies the scope of the enumeration. This parameter can be one or more of the following values.

DIEDBSFL_ATTACHEDONLY

Only attached and installed devices are enumerated.

DIEDBSFL_AVAILABLEDEVICES

Only unowned, installed devices are enumerated.

DIEDBSFL_FORCEFEEDBACK

Only devices that support force feedback are enumerated. DIEDBSFL_MULTIMICEKEYBOARDS

Only secondary (non-system) keyboard and mouse devices.

DIEDBSFL_NONGAMINGDEVICES

Only HID-compliant devices whose primary purpose is not as a gaming device. Devices such as USB speakers and multimedia buttons on some keyboards would fall within this value.

DIEDBSFL_THISUSER

All installed devices for the user identified by *ptszUserName*, and all unowned devices, are enumerated.

DIEDBSFL_VALID is also defined in Dinput.h, but is not used by applications.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following error values.

DIERR_INVALIDPARAM DIERR_NOTINITIALIZED

Remarks

The keyboard and mouse are enumerated last.

Note The order in which devices are enumerated by DirectInput is not guaranteed.

Requirements

Windows NT/2000/XP: Requires Windows® 2000.
Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.
Header: Declared in Dinput.h.

See Also

IDirectInput8::EnumDevices, Action Mapping.

IDirectInput8::FindDevice

Retrieves the instance GUID of a device that has been newly attached to the system. It is called in response to a Microsoft® Win32® device management notification.

```
HRESULT FindDevice(
    REFGUID rguidClass,
    LPCTSTR ptszName,
    LPGUID pguidInstance
);
```

Parameters

rguidClass

Unique identifier of the device class for the device that the application is to locate. The application obtains the class GUID from the device arrival notification. For more information, see the documentation on the DBT_DEVICEARRIVAL event in the Microsoft Platform Software Development Kit (SDK).

ptszName

Name of the device. The application obtains the name from the device arrival notification.

pguidInstance

Address of a variable to receive the instance GUID for the device, if the device is found. This value can be passed to <u>IDirectInput8::CreateDevice</u>.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be <u>DIERR_DEVICENOTREG</u>. Failure results if the GUID and name do not correspond to a device class that is registered with Microsoft® DirectInput®. For example, they might refer to a storage device, rather than an input device.

Requirements

Windows NT/2000/XP: Requires Windows® 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

IDirectInput8::GetDeviceStatus

Retrieves the status of a specified device.

```
HRESULT GetDeviceStatus(
    REFGUID rguidInstance
);
```

Parameters

rguidInstance

Reference to (C++) or address of (C) the GUID identifying the instance of the device whose status is being checked.

Return Values

If the method succeeds, the return value is DI_OK if the device is <u>attached</u> to the system, or <u>DI_NOTATTACHED</u> otherwise.

If the method fails, the return value can be one of the following error values:

DIERR_GENERIC DIERR_INVALIDPARAM DIERR_NOTINITIALIZED

Requirements

Windows NT/2000/XP: Requires Windows @ 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

IDirectInput8::Initialize

Initializes a Microsoft® DirectInput® object. Applications normally do not need to call this method. The <u>DirectInput8Create</u> function automatically initializes the DirectInput object after creating it.

```
HRESULT Initialize(
   HINSTANCE hinst,
   DWORD dwVersion
);
```

Parameters

hinst

Instance handle to the application or dynamic-link library (DLL) that is creating the DirectInput object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that might be necessary for backwards compatibility.

It is an error for a DLL to pass the handle of the parent application. For example, a Microsoft ActiveX® control embedded in a Web page that uses DirectInput must pass its own instance handle, and not the handle of the Web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that might be necessary.

dwVersion

Version number of DirectInput for which the application is designed. This value is normally DIRECTINPUT_VERSION. Passing the version number of a previous version causes DirectInput to emulate that version.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following error values:

DIERR_BETADIRECTINPUTVERSION

DIERR_OLDDIRECTINPUTVERSION

Requirements

Windows NT/2000/XP: Requires Windows® 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

IDirectInput8::RunControlPanel

Runs Control Panel to enable the user to install a new input device or modify configurations.

```
HRESULT RunControlPanel(
   HWND hwndOwner,
   DWORD dwFlags
);
```

Parameters

hwndOwner

Handle of the window to be used as the parent window for the subsequent user interface. If this parameter is NULL, no parent window is used.

dwFlags

Currently not used and must be set to 0.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following error values:

DIERR_INVALIDPARAM DIERR_NOTINITIALIZED

Requirements

Windows NT/2000/XP: Requires Windows® 2000. Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

See Also

IDirectInputDevice8::RunControlPanel

IUnknown::AddRef

Increases the reference count of the object by 1.

ULONG AddRef();

Parameters

There are no parameters.

Return Values

Returns the new reference count. This value is for diagnostic and testing purposes only.

Remarks

When the object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the **Release** method to decrease the object's reference count by 1.

Requirements

Windows NT/2000/XP: Requires Windows NT 3.1 or later. Windows 98/Me: Requires Windows 98 or later. Header: Declared in Unknwn.h.

IUnknown::QueryInterface

Determines whether the object supports a particular COM interface. If it does, the system increases the object's reference count, and the application can use that interface immediately.

```
HRESULT QueryInterface(
    REFIID riid,
    LPVOID* ppvObj
);
```

Parameters

riid

Reference identifier of the interface being requested.

ppvObj

Address of a pointer to fill with the interface pointer if the query succeeds.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value may be E_NOINTERFACE or E_POINTER. Some components also have their own definitions of these error values in their header files. In Microsoft® DirectInput®, for example, DIERR_NOINTERFACE is equivalent to E_NOINTERFACE.

Remarks

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **QueryInterface** method enables Microsoft and third parties to extend objects without interfering with existing or evolving functionality.

Requirements

Windows NT/2000/XP: Requires Windows NT 3.1 or later. Windows 98/Me: Requires Windows 98 or later. Header: Declared in Unknwn.h.

IUnknown::Release

Decreases the reference count of the object by 1.

ULONG Release();

Parameters

There are no parameters.

Return Values

Returns the new reference count. This value is for diagnostic and testing purposes only.

Remarks

The object deallocates itself when its reference count reaches 0. Use the **AddRef** method to increase the object's reference count by 1.

Applications must call this method to release only interfaces that the method explicitly created in a previous call to **IUnknown::AddRef**, **IUnknown::QueryInterface**, or a creation function such as **Direct3DCreate8**.

Requirements

Windows NT/2000/XP: Requires Windows NT 3.1 or later. Windows 98/Me: Requires Windows 98 or later. Header: Declared in Unknwn.h.

Return Values

The list below contains the **HRESULT** values that can be returned by Microsoft® DirectInput® methods and functions. Errors are represented by negative values and cannot be combined.

For a list of the error values each method or function can return, see the individual descriptions. Lists of error codes in the documentation are necessarily incomplete. For example, any DirectInput method can return DIERR_OUTOFMEMORY even though the error code is not explicitly listed as a possible return value in the documentation for that method.

DI_BUFFEROVERFLOW

The device buffer overflowed and some input was lost. This value is equal to the S_FALSE standard COM return value.

DI_DOWNLOADSKIPPED

The parameters of the effect were successfully updated, but the effect could not be downloaded because the associated device was not acquired in exclusive mode.

DI_EFFECTRESTARTED

The effect was stopped, the parameters were updated, and the effect was restarted.

DI_NOEFFECT

The operation had no effect. This value is equal to the S_FALSE standard COM return value.

DI_NOTATTACHED

The device exists but is not currently <u>attached</u>. This value is equal to the S_FALSE standard COM return value.

DI_OK

The operation completed successfully. This value is equal to the S_OK standard COM return value.

DI_POLLEDDEVICE

The device is a polled device. As a result, device buffering does not collect any data and event notifications is not signaled until the **IDirectInputDevice8::Poll** method is called.

DI_PROPNOEFFECT

The change in device properties had no effect. This value is equal to the S_FALSE standard COM return value.

DI_SETTINGSNOTSAVED

The action map was applied to the device, but the settings could not be saved.

DI_TRUNCATED

The parameters of the effect were successfully updated, but some of them were beyond the capabilities of the device and were truncated to the nearest supported value.

DI_TRUNCATEDANDRESTARTED

Equal to DI_EFFECTRESTARTED | DI_TRUNCATED.

DI_WRITEPROTECT

A SUCCESS code indicating that settings cannot be modified.

DIERR_ACQUIRED

The operation cannot be performed while the device is acquired.

DIERR_ALREADYINITIALIZED

This object is already initialized

DIERR_BADDRIVERVER

The object could not be created due to an incompatible driver version or mismatched or incomplete driver components.

DIERR_BETADIRECTINPUTVERSION

The application was written for an unsupported prerelease version of DirectInput.

DIERR_DEVICEFULL

The device is full.

DIERR_DEVICENOTREG

The device or device instance is not registered with DirectInput. This value is equal to the REGDB_E_CLASSNOTREG standard COM return value.

DIERR_EFFECTPLAYING

The parameters were updated in memory but were not downloaded to the device because the device does not support updating an effect while it is still playing.

DIERR_GENERIC

An undetermined error occurred inside the DirectInput subsystem. This value is equal to the E_FAIL standard COM return value.

DIERR_HANDLEEXISTS

The device already has an event notification associated with it. This value is equal to the E_ACCESSDENIED standard COM return value.

DIERR_HASEFFECTS

The device cannot be reinitialized because effects are attached to it. DIERR_INCOMPLETEEFFECT

The effect could not be downloaded because essential information is missing. For example, no axes have been associated with the effect, or no type-specific information has been supplied.

DIERR_INPUTLOST

Access to the input device has been lost. It must be reacquired.

DIERR_INVALIDPARAM

An invalid parameter was passed to the returning function, or the object was not in a state that permitted the function to be called. This value is equal to the E_INVALIDARG standard COM return value.

DIERR_MAPFILEFAIL

An error has occurred either reading the vendor-supplied action-mapping file for the device or reading or writing the user configuration mapping file for the device.

DIERR_MOREDATA

Not all the requested information fit into the buffer.

DIERR_NOAGGREGATION

This object does not support aggregation.

DIERR_NOINTERFACE

The object does not support the specified interface. This value is equal to the E_NOINTERFACE standard COM return value.

DIERR_NOTACQUIRED

The operation cannot be performed unless the device is acquired.

DIERR_NOTBUFFERED

The device is not buffered. Set the DIPROP_BUFFERSIZE property to enable buffering.

DIERR_NOTDOWNLOADED

The effect is not downloaded.

DIERR_NOTEXCLUSIVEACQUIRED

The operation cannot be performed unless the device is acquired in

DISCL_EXCLUSIVE mode.

DIERR_NOTFOUND

The requested object does not exist.

DIERR_NOTINITIALIZED

This object has not been initialized.

DIERR_OBJECTNOTFOUND

The requested object does not exist.

DIERR_OLDDIRECTINPUTVERSION

The application requires a newer version of DirectInput.

DIERR_OTHERAPPHASPRIO

Another application has a higher priority level, preventing this call from succeeding. This value is equal to the E_ACCESSDENIED standard COM return value. This error can be returned when an application has only foreground access to a device but is attempting to acquire the device while in the background.

DIERR_OUTOFMEMORY

The DirectInput subsystem could not allocate sufficient memory to complete the call. This value is equal to the E_OUTOFMEMORY standard COM return value.

DIERR_READONLY

The specified property cannot be changed. This value is equal to the E_ACCESSDENIED standard COM return value.

DIERR_REPORTFULL

More information was requested to be sent than can be sent to the device. DIERR_UNPLUGGED

The operation could not be completed because the device is not plugged in. DIERR_UNSUPPORTED

The function called is not supported at this time. This value is equal to the E_NOTIMPL standard COM return value.

E_HANDLE

The HWND parameter is not a valid top-level window that belongs to the process.

E_PENDING

Data is not yet available.

E_POINTER

An invalid pointer, usually NULL, was passed as a parameter.

IDirectInputDevice8::GetDeviceState

Retrieves immediate data from the device.

```
HRESULT GetDeviceState(
   DWORD cbData,
   LPVOID lpvData
);
```

Parameters

cbData

Size of the buffer in the *lpvData* parameter, in bytes.

lpvData

Address of a structure that receives the current state of the device. The format of the data is established by a prior call to the **IDirectInputDevice8::SetDataFormat** method.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following error values:

DIERR_INPUTLOST DIERR_INVALIDPARAM DIERR_NOTACQUIRED DIERR_NOTINITIALIZED E PENDING

Remarks

Before device data can be obtained, set the cooperative level by using the **IDirectInputDevice8::SetCooperativeLevel** method, then set the data format by using **IDirectInputDevice8::SetDataFormat**, and acquire the device by using the **IDirectInputDevice8::Acquire** method.

The five predefined data formats require corresponding device state structures according to the following table:

Data format	State structure
c_dfDIMouse	DIMOUSESTATE
<i>c_dfDIMouse2</i>	DIMOUSESTATE2
c_dfDIKeyboard	array of 256 bytes
c_dfDIJoystick	DIJOYSTATE
c_dfDIJoystick2	DIJOYSTATE2

For example, if you passed the *c_dfDIMouse* format to the **IDirectInputDevice8::SetDataFormat** method, you must pass a **DIMOUSESTATE** structure to the **IDirectInputDevice8::GetDeviceState** method.

Requirements

Windows NT/2000/XP: Requires Windows® 2000. **Windows 98/Me:** Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

See Also

IDirectInputDevice8::Poll, Polling and Event Notification, Buffered and Immediate Data

IDirectInputDevice8::GetObjectInfo

Retrieves information about a device object, such as a button or axis.

```
HRESULT GetObjectInfo(
   LPDIDEVICEOBJECTINSTANCE pdidoi,
   DWORD dwObj,
   DWORD dwHow
);
```

Parameters

pdidoi

Address of a **DIDEVICEOBJECTINSTANCE** structure to be filled with information about the object. The structure's **dwSize** member must be initialized before this method is called.

dwObj

Value that identifies the object whose information is to be retrieved. The value set for this parameter depends on the value specified in the *dwHow* parameter.

dwHow

Value that specifies how the *dwObj* parameter should be interpreted. This value can be one of the following:

Value	Meaning
DIPH_BYOFFSET	The <i>dwObj</i> parameter is the offset into the current data format of the object whose information is being accessed.
DIPH_BYID	The <i>dwObj</i> parameter is the object type/instance identifier. This identifier is returned in the dwType member of the DIDEVICEOBJECTINSTANCE structure returned from a previous call to the IDirectInputDevice8::EnumObjects method.
DIPH_BYUSAGE	The <i>dwObj</i> parameter contains the HID Usage Page and Usage values of the object, combined by the DIMAKEUSAGEDWORD macro.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following error values:

DIERR_INVALIDPARAM DIERR_NOTINITIALIZED DIERR_OBJECTNOTFOUND E_POINTER

Remarks

For compatibility with Microsoft® DirectX® 3, it is also valid to pass a **DIDEVICEOBJECTINSTANCE_DX3** structure with the **dwSize** member initialized to **sizeof(DIDEVICEOBJECTINSTANCE_DX3)**.

Requirements

Windows NT/2000/XP: Requires Windows® 2000. Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

IDirectInputDevice8::SetDataFormat

Sets the data format for the Microsoft® DirectInput® device.

```
HRESULT SetDataFormat(
   LPCDIDATAFORMAT lpdf
);
```

Parameters

lpdf

Address of a structure that describes the format of the data that the DirectInputDevice should return. An application can define its own **DIDATAFORMAT** structure or use one of the following predefined global variables:

- c_dfDIKeyboard
- *c_dfDIMouse*
- *c_dfDIMouse2*
- *c_dfDIJoystick*
- c_dfDIJoystick2

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value can be one of the following error values:

DIERR_ACQUIRED DIERR_INVALIDPARAM DIERR_NOTINITIALIZED

Remarks

The data format must be set before the device can be acquired by using the **IDirectInputDevice8::Acquire** method. It is necessary to set the data format only once. The data format cannot be changed while the device is acquired.

If the application is using action mapping, the data format is set instead by the call to **IDirectInputDevice8::SetActionMap**.

Requirements

Windows NT/2000/XP: Requires Windows® 2000.

Windows 98/Me: Requires Windows 98 or later. Available as a redistributable for Windows 98.

Header: Declared in Dinput.h.

See Also

IDirectInputDevice8::GetDeviceState