Open CASCADE Technology 7.2.0

MFC samples
1. Contents

The directory `samples/mfc/standard` contains the following packages and files:

- Numbered packages: **01_Geometry**, **02_Modeling**, etc. provide projects and sources of samples;
- Files **All-vc(number).sln** are auxiliary utility projects depending on all other sample projects. When such project is rebuilt, all samples and **mfcsample** library are also rebuilt.
- **Common** directory provides common source and header files for samples and dynamic-link library **mfcsample.dll**.
- **Data** directory stores data files.
- **mfcsample** directory contains project for **mfcsample.dll** library providing basic functionality used by all OCC samples.
- File **env.bat** is called from **msvc.bat**.
2. Launching Open CASCADE Technology samples:

To run the Open CASCADE Technology samples, use command:

```
execute run.bat [vc10|vc11|vc12|vc14] [win32|win64] [Release|Debug] [SampleName]
```

To run the **Animation** sample, use command:

```
execute run.bat vc10 win64 Debug Animation
```
3. Modifying and rebuilding samples:

You can modify, compile and launch all sample projects in MS Visual C++ at once with command:

```
execute msvc.bat [vc10|vc11|vc12|vc14] [win32|win64] [Release|Debug]
```

To run all sample projects in MS Visual C++ at once, use command:

```
execute msvc.bat vc10 win64 Debug
```

Note: make sure that your PATH environment variable contains a directory, where msdev.exe is located.
OCCT CSharp sample

This sample demonstrates how to use OCCT libraries in .Net application written using CSharp and Windows Forms or Windows Presentation Foundation (WPF).

The connection between .Net and OCCT (C++) level is provided by proxy library OCCProxy, written in C++/CLI. The proxy library contains a single ref class encapsulating OCCT viewer and providing the functionality to manipulate this viewer and to import / export OCCT shapes from / to several supported CAD file formats (IGES, STEP, BREP).

The sample implements two approaches to the development of a user interface with C#. Both applications provide the same functionality as the standard OCCT Import/Export sample. The first project is called IE_WinForms and uses Windows Forms for GUI. The second application is called IE_WPF_WinForms and uses Windows Presentation Foundation.

Note a few important details:

- OCCT template class NCollection_Haft is used to encapsulate C++ class into a field of ref class;
- It is necessary to explicitly set the target platform for C# assemblies to x86 in project Properties - Build to work consistently on 64-bit systems with OCCT libraries built in 32-bit mode;
- this sample demonstrates indirect method of wrapping C++ to C# using a manually created proxy library. There is an alternative method of wrapping individual OCCT classes to C# equivalents to make their full API available to a C# user and to let the code be programmed on C# level similarly to C++ one. See the description of OCCT C# Wrapper in Advanced Samples and Tools at http://www.opencascade.org/support/products/advsamples
- in WPF sample, WinForms control is used to encapsulate OCC
viewer since WPF does not provide the necessary interface to embed OpenGL view. Other possible solution could be to render OpenGL scene in an off-screen buffer and to map it to WPF control as an image. That approach would allow using all WPF features to control the OCCT viewer.

Run `msvc.bat` to start MS Visual Studio for building the sample. Note that project files are provided only for VS 2010, you can open them in newer versions of Visual Studio the using automatic converter.

After conversion check option **Target framework** in the properties of C# projects (tab **Application**) to make sure that it corresponds to the version set in the properties of C++ projects (e.g. **.Net Framework 4.0** for VS 2010).

Run `run_winforms.bat` or `run_wpf.bat` to launch the corresponding sample.

Note that all batch scripts use the configuration defined in OCCT `custom.bat` file as default; you can provide arguments specifying VS version, bitness, and mode to override these settings, e.g.:

```
> msvc.bat vc10 win64 Debug
```
Direct3D CSharp sample

This sample demonstrates how to use OCCT and DirectX libraries in .Net application written using CSharp and Windows Presentation Foundation (WPF).

The connection between .Net, OCCT (C++) and DirectX level is provided by proxy libraries, OCCProxy and D3DProxy, written in C++/CLI. The proxy library OCCProxy contains a single ref class encapsulating OCCT viewer and providing the functionality to manipulate this viewer and to import / export OCCT shapes from / to several supported CAD file formats (IGES, STEP, BREP). The proxy library D3DProxy contains helper methods for rendering via DirectX.

The user interface in this sample is based on Windows Presentation Foundation (WPF). It has the same functionality as the standard OCCT Import/Export sample. The project is called IE_WPF_D3D.

Note a few important details:

- to build this sample you should to download and install DirectX SDK http://www.microsoft.com/en-us/download/details.aspx?id=6812
- OCCT template class NCollection_Haft is used to encapsulate C++ class into a field of ref class;
- It is necessary to explicitly set the target platform for C# assemblies to x86 in project Properties - Build to work consistently on 64-bit systems with OCCT libraries built in 32-bit mode;
- this sample demonstrates indirect method of wrapping C++ to C# using a manually created proxy library. There is an alternative method of wrapping individual OCCT classes to C# equivalents to make their full API available to a C# user and to let the code be programmed on C# level similarly to C++ one. See the description of OCCT C# Wrapper in Advanced Samples and Tools at http://www.opencascade.org/support/products/advsamples
• in WPF sample, **WinForms** control is used to encapsulate OCC viewer since WPF does not provide the necessary interface to embed OpenGl view. Other possible solution could be to render OpenGl scene in an off-screen buffer and to map it to WPF control as an image. That approach would allow using all WPF features to control the OCCT viewer.

Run `msvc.bat` to start MS Visual Studio for building the sample. Note that project files are provided only for VS 2010, you can open them in newer versions of Visual Studio using an automatic converter.

After conversion check option **Target framework** in the properties of C# projects (tab Application) to make sure that it corresponds to the version set in the properties of C++ projects (e.g. **.Net Framework 4.0** for VS 2010).

Run `run_wpf-D3D.bat` to launch the corresponding sample.

Note that all batch scripts use the configuration defined in OCCT custom.bat file as default; you can provide arguments specifying VS version, bitness, and mode to override these settings, e.g.:

```
> msvc.bat vc10 win64 Debug
```
OCCT JniViewer sample for Android

This sample demonstrates simple way of using OCCT libraries in Android application written using Java.

The connection between Java and OCCT (C++) level is provided by proxy library, libTKJniSample.so, written in C++ with exported JNI methods of Java class OcctJniRenderer. The proxy library contains single C++ class OcctJni_Vi ewer encapsulating OCCT viewer and providing functionality to manipulate this viewer and to import OCCT shapes from several supported formats of CAD files (IGES, STEP, BREP).

This sample demonstrates indirect method of wrapping C++ to Java using manually created proxy library. Alternative method is available, wrapping individual OCCT classes to Java equivalents so that their full API is available to Java user and the code can be programmed on Java level similarly to C++ one. See description of OCCT Java Wrapper in Advanced Samples and Tools on OCCT web site at http://www.opencascade.org/support/products/advsamples

Run Eclipse from ADT (Android Developer Tools) for building the sample. To import sample project perform

File -> Import... -> Android -> Existing Android code into Workspace

and specify this directory. The project re-build will be started immediately right after importation if "Build automatically" option is turned on (default in Eclipse). Proxy library compilation and packaging is performed by NDK build script, called by "C++ Builder" configured within Eclipse project. The path to "ndk-build" tool from Android NDK (Native Development Kit) should be specified in Eclipse project properties:

Project -> Properties -> Builders -> C++ Builder ->
Now paths to OCCT C++ libraries and additional components should be specified in "jni/Android.mk" file:

```
OCCT_ROOT := $(LOCAL_PATH)/../../..
FREETYPE_INC := $(OCCT_ROOT)/../freetype/include/freetype2
FREETYPE_LIBS := $(OCCT_ROOT)/../freetype/libs
FREEIMAGE_INC := $(OCCT_ROOT)/../FreeImage/include
FREEIMAGE_LIBS := $(OCCT_ROOT)/../FreeImage/libs
OCCT_INC := $(OCCT_ROOT)/inc
OCCT_LIBS := $(OCCT_ROOT)/and/libs
```

The list of extra components (Freetype, FreeImage) depends on OCCT configuration. Variable is used within this script to refer to active architecture. E.g. for 32-bit ARM build (see variable APP_ABI in "jni/Application.mk") the folder OCCT_LIBS should contain sub-folder "armeabi-v7a" with OCCT libraries.

FreeImage is optional and does not required for this sample, however you should include all extra libraries used for OCCT building and load the explicitly from Java code within OcctJniActivity::loadNatives() method, including toolkits from OCCT itself in proper order:

```
if (!loadLibVerbose ("TKernel", aLoaded, aFailed)
|| !loadLibVerbose ("TKMath", aLoaded, aFailed)
|| !loadLibVerbose ("TKG2d", aLoaded, aFailed)
```

Note that C++ STL library is not part of Android system. Thus application must package this library as well as extra component. "gnumstl_shared" STL implementation is expected within this sample.

After successful build, the application can be packaged to Android:

- Deploy and run application on connected device or emulator directly from Eclipse using adb interface by menu items "Run" and "Debug".
This would sign package with debug certificate.
• Prepare signed end-user package using wizard File -> Export -> Android -> Export Android Application.
OCCT AndroidQt sample for Android

This sample demonstrates a simple way of using OCCT libraries in an Android application written using Qt/Qml.

The connection between Qt/Qml and OCCT (C++) level is provided by a proxy library, libAndroidQt.so, written in C++. The proxy library contains a single C++ class AndroidQt encapsulating OCCT viewer and providing functionality to manipulate this viewer and to import OCCT shapes from supported format of CAD file (BREP).

Requirements for building sample:

- Java Development Kit 1.7 or higher
- Qt 5.3 or higher
- Android SDK from 2014.07.02 or newer
- Android NDK r9d or newer
- Apache Ant 1.9.4 or higher

Configure project for building sample:

In QtCreator, open AndroidQt.pro project-file:

File -> Open file or Project...

Specify Android configurations:

Tools->Options->Android

- In JDK location specify path to Java Development Kit
- In Android SDK location specify path to Android SDK
- In Android NDK location specify path to Android NDK
- In Ant executable specify path to ant.bat file located in Apache Ant bin directory
Make sure that "Android for armeabi-v7a" kit has been detected

Tools->Options->Build & Run

The paths to OCCT and 3rdparty libraries are specified in "OCCT.pri" file:

the paths to the headers:

```
INCLUDEPATH += /occt/inc /3rdparty/include
DEPENDPATH += /occt/inc /3rdparty/include
```

the libraries location:

```
LIBS += -L/occt/libs/armeabi-v7a
```

OCCT resources (Shaders, SHMessage, StdResource, TObj, UnitsAPI and XSMMessage folder) should be copied to androidqt_dir/android/assets/opencascade/shared/ directory. Current sample requires at least Shaders folder.

Select build configuration: Debug or Release and click Build->Build Project "AndroidQt" or (Ctrl + B). After successful build the application can be deployed to device or emulator.
Open CASCADE Technology 7.2.0

Tutorial

Table of Contents

- Overview
  - Prerequisites
  - The Model
  - Model Specifications
- Building the Profile
  - Defining Support Points
  - Profile: Defining the Geometry
  - Profile: Defining the Topology
  - Profile: Completing the Profile
- Building the Body
  - Prism the Profile
  - Applying Fillets
  - Adding the Neck
  - Creating a Hollowed Solid
- Building the Threading
  - Creating Surfaces
  - Defining 2D Curves
  - Building Edges and Wires
  - Creating Threading
- Building the Resulting Compound
- Appendix
Overview

This tutorial will teach you how to use Open CASCADE Technology services to model a 3D object. The purpose of this tutorial is not to describe all Open CASCADE Technology classes but to help you start thinking in terms of Open CASCADE Technology as a tool.
Prerequisites

This tutorial assumes that you have experience in using and setting up C++. From a programming standpoint, Open CASCADE Technology is designed to enhance your C++ tools with 3D modeling classes, methods and functions. The combination of all these resources will allow you to create substantial applications.
The Model

To illustrate the use of classes provided in the 3D geometric modeling toolkits, you will create a bottle as shown:

In the tutorial we will create, step-by-step, a function that will model a bottle as shown above. You will find the complete source code of this tutorial, including the very function `MakeBottle` in the distribution of Open CASCADE Technology. The function body is provided in the file `samples/qt/Tutorial/src/MakeBottle.cxx`. 
Model Specifications

We first define the bottle specifications as follows:

<table>
<thead>
<tr>
<th>Object Parameter</th>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottle height</td>
<td>MyHeight</td>
<td>70mm</td>
</tr>
<tr>
<td>Bottle width</td>
<td>MyWidth</td>
<td>50mm</td>
</tr>
<tr>
<td>Bottle thickness</td>
<td>MyThickness</td>
<td>30mm</td>
</tr>
</tbody>
</table>

In addition, we decide that the bottle's profile (base) will be centered on the origin of the global Cartesian coordinate system.

This modeling requires four steps:

- build the bottle's Profile
- build the bottle's Body
- build the Threading on the bottle's neck
- build the result compound
Building the Profile

Defining Support Points

To create the bottle's profile, you first create characteristic points with their coordinates as shown below in the (XOY) plane. These points will be the supports that define the geometry of the profile.

\begin{center}
\begin{tikzpicture}
\draw[->] (-myWidth/2,0) -- (myWidth/2,0);
\draw[->] (0,-myThickness/2) -- (0,myThickness/2);
\fill (-myWidth/2,0,0) circle (1pt) node[below] {(-myWidth / 2., 0, 0)};
\fill (myWidth/2,0,0) circle (1pt) node[below] {(myWidth / 2., 0, 0)};
\fill (-myWidth/2,-myThickness/4,0) circle (1pt) node[below] {(-myWidth / 2., -myThickness / 4., 0)};
\fill (myWidth/2,-myThickness/4,0) circle (1pt) node[below] {(myWidth / 2., myThickness / 4., 0)};
\fill (0,-myThickness/2,0) circle (1pt) node[below] {(0, -myThickness / 2., 0)};
\end{tikzpicture}
\end{center}

There are two classes to describe a 3D Cartesian point from its X, Y and Z coordinates in Open CASCADE Technology:

- the primitive geometric `gp_Pnt` class
- the transient `Geom_CartesianPoint` class manipulated by handle

A handle is a type of smart pointer that provides automatic memory management. To choose the best class for this application, consider the following:

- `gp_Pnt` is manipulated by value. Like all objects of its kind, it will have a limited lifetime.
- `Geom_CartesianPoint` is manipulated by handle and may have multiple references and a long lifetime.

Since all the points you will define are only used to create the profile's curves, an object with a limited lifetime will do. Choose the `gp_Pnt` class. To instantiate a `gp_Pnt` object, just specify the X, Y, and Z coordinates of the points in the global Cartesian coordinate system:

\begin{verbatim}
gp_Pnt aPnt1(-myWidth / 2., 0, 0);
\end{verbatim}
Once your objects are instantiated, you can use methods provided by the class to access and modify its data. For example, to get the X coordinate of a point:

```
Standard_Real xValue1 = aPnt1.X();
```
Profile: Defining the Geometry

With the help of the previously defined points, you can compute a part of the bottle's profile geometry. As shown in the figure below, it will consist of two segments and one arc.

To create such entities, you need a specific data structure, which implements 3D geometric objects. This can be found in the Geom package of Open CASCADE Technology. In Open CASCADE Technology a package is a group of classes providing related functionality. The classes have names that start with the name of a package they belong to. For example, Geom_Line and Geom_Circle classes belong to the Geom package. The Geom package implements 3D geometric objects: elementary curves and surfaces are provided as well as more complex ones (such as Bezier and BSpline). However, the Geom package provides only the data structure of geometric entities. You can directly instantiate classes belonging to Geom, but it is easier to compute elementary curves and surfaces by using the GC package. This is because the GC provides two algorithm classes which are exactly what is required for our profile:

- Class GC_MakeSegment to create a segment. One of its constructors allows you to define a segment by two end points P1 and P2
- Class GC_MakeArcOfCircle to create an arc of a circle. A useful constructor creates an arc from two end points P1 and P3 and going through P2.

Both of these classes return a Geom_TrimmedCurve manipulated by handle. This entity represents a base curve (line or circle, in our case), limited between two of its parameter values. For example, circle C is parameterized between 0 and 2PI. If you need to create a quarter of a circle, you create a Geom_TrimmedCurve on C limited between 0 and
M_PI/2.

```cpp
Handle(Geom_TrimmedCurve) aArcOfCircle = 
    GC_MakeArcOfCircle(aPnt2,aPnt3,aPnt4);
Handle(Geom_TrimmedCurve) aSegment1	=
    GC_MakeSegment(aPnt1, aPnt2);
Handle(Geom_TrimmedCurve) aSegment2	=
    GC_MakeSegment(aPnt4, aPnt5);
```

All GC classes provide a casting method to obtain a result automatically with a function-like call. Note that this method will raise an exception if construction has failed. To handle possible errors more explicitly, you may use the `IsDone` and `Value` methods. For example:

```cpp
GC_MakeSegment mkSeg (aPnt1, aPnt2);
Handle(Geom_TrimmedCurve) aSegment1;
if(mkSegment.IsDone()){
    aSegment1 = mkSeg.Value();
}
else {
    // handle error
}
```
Profile: Defining the Topology

You have created the support geometry of one part of the profile but these curves are independent with no relations between each other. To simplify the modeling, it would be right to manipulate these three curves as a single entity. This can be done by using the topological data structure of Open CASCADE Technology defined in the TopoDS package: it defines relationships between geometric entities which can be linked together to represent complex shapes. Each object of the TopoDS package, inheriting from the TopoDS_Shape class, describes a topological shape as described below:

<table>
<thead>
<tr>
<th>Shape</th>
<th>Open CASCADE Technology Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>TopoDS_Vertex</td>
<td>Zero dimensional shape corresponding to a point in geometry.</td>
</tr>
<tr>
<td>Edge</td>
<td>TopoDS_Edge</td>
<td>One-dimensional shape corresponding to a curve and bounded by a vertex at each extremity.</td>
</tr>
<tr>
<td>Wire</td>
<td>TopoDS_Wire</td>
<td>Sequence of edges connected by vertices.</td>
</tr>
<tr>
<td>Face</td>
<td>TopoDS_Face</td>
<td>Part of a surface bounded by a closed wire(s).</td>
</tr>
<tr>
<td>Shell</td>
<td>TopoDS_Shell</td>
<td>Set of faces connected by edges.</td>
</tr>
<tr>
<td>Solid</td>
<td>TopoDS_Solid</td>
<td>Part of 3D space bounded by Shells.</td>
</tr>
<tr>
<td>CompSolid</td>
<td>TopoDS_CompSolid</td>
<td>Set of solids connected by their faces.</td>
</tr>
<tr>
<td>Compound</td>
<td>TopoDS_Compound</td>
<td>Set of any other shapes described above.</td>
</tr>
</tbody>
</table>

Referring to the previous table, to build the profile, you will create:

- Three edges out of the previously computed curves.
One wire with these edges.

However, the TopoDS package provides only the data structure of the topological entities. Algorithm classes available to compute standard topological objects can be found in the BRepBuilderAPI package. To create an edge, you use the BRepBuilderAPI_MakeEdge class with the previously computed curves:

```cpp
TopoDS_Edge aEdge1 = BRepBuilderAPI_MakeEdge(aSegment1);
TopoDS_Edge aEdge2 = BRepBuilderAPI_MakeEdge(aArcOfCircle);
TopoDS_Edge aEdge3 = BRepBuilderAPI_MakeEdge(aSegment2);
```

In Open CASCADE Technology, you can create edges in several ways. One possibility is to create an edge directly from two points, in which case the underlying geometry of this edge is a line, bounded by two vertices being automatically computed from the two input points. For example, aEdge1 and aEdge3 could have been computed in a simpler way:

```cpp
TopoDS_Edge aEdge1 = BRepBuilderAPI_MakeEdge(aPnt1, aPnt3);
TopoDS_Edge aEdge2 = BRepBuilderAPI_MakeEdge(aPnt4, aPnt5);
```

To connect the edges, you need to create a wire with the BRepBuilderAPI_MakeWire class. There are two ways of building a wire with this class:

- directly from one to four edges
- by adding other wire(s) or edge(s) to an existing wire (this is explained later in this tutorial)
When building a wire from less than four edges, as in the present case, you can use the constructor directly as follows:

```
TopoDS_Wire aWire = BRepBuilderAPI_MakeWire(aEdge1, aEdge2, aEdge3);
```
Profile: Completing the Profile

Once the first part of your wire is created you need to compute the complete profile. A simple way to do this is to:

- compute a new wire by reflecting the existing one.
- add the reflected wire to the initial one.

To apply a transformation on shapes (including wires), you first need to define the properties of a 3D geometric transformation by using the gp_Trsf class. This transformation can be a translation, a rotation, a scale, a reflection, or a combination of these. In our case, we need to define a reflection with respect to the X axis of the global coordinate system. An axis, defined with the gp_Ax1 class, is built out of a point and has a direction (3D unitary vector). There are two ways to define this axis. The first way is to define it from scratch, using its geometric definition:

- X axis is located at (0, 0, 0) - use the gp_Pnt class.
- X axis direction is (1, 0, 0) - use the gp_Dir class. A gp_Dir instance is created out of its X, Y and Z coordinates.

```cpp
gp_Pnt aOrigin(0, 0, 0);
gp_Dir xDir(1, 0, 0);
gp_Ax1 xAxis(aOrigin, xDir);
```

The second and simplest way is to use the geometric constants defined in the gp package (origin, main directions and axis of the global coordinate system). To get the X axis, just call the gp::OX method:
As previously explained, the 3D geometric transformation is defined with the `gp_Trsf` class. There are two different ways to use this class:

- by defining a transformation matrix by all its values
- by using the appropriate methods corresponding to the required transformation (SetTranslation for a translation, SetMirror for a reflection, etc.): the matrix is automatically computed.

Since the simplest approach is always the best one, you should use the SetMirror method with the axis as the center of symmetry.

```cpp
gp_Trsf aTrsf;
aTrsf.SetMirror(xAxis);
```

You now have all necessary data to apply the transformation with the BRepBuilderAPI_Transform class by specifying:

- the shape on which the transformation must be applied.
- the geometric transformation

```cpp
BRepBuilderAPI_Transform aBRepTrsf(aWire, aTrsf);
```

`BRepBuilderAPI_Transform` does not modify the nature of the shape: the result of the reflected wire remains a wire. But the function-like call or the `BRepBuilderAPI_Transform::Shape` method returns a TopoDS_Shape object:

```cpp
TopoDS_Shape aMirroredShape = aBRepTrsf.Shape();
```

What you need is a method to consider the resulting reflected shape as a wire. The TopoDS global functions provide this kind of service by casting a shape into its real type. To cast the transformed wire, use the TopoDS::Wire method.

```cpp
TopoDS_Wire aMirroredWire = TopoDS::Wire(aMirroredShape);
```

The bottle's profile is almost finished. You have created two wires: `aWire`
and \textit{aMirroredWire}. You need to concatenate them to compute a single shape. To do this, you use the \textit{BRepBuilderAPI\_MakeWire} class as follows:

- create an instance of \textit{BRepBuilderAPI\_MakeWire}.
- add all edges of the two wires by using the \textit{Add} method on this object.

```cpp
BRepBuilderAPI\_MakeWire mkWire;
mkWire.Add(aWire);
mkWire.Add(aMirroredWire);
TopoDS\_Wire myWireProfile = mkWire.Wire();
```
Building the Body

Prism the Profile

To compute the main body of the bottle, you need to create a solid shape. The simplest way is to use the previously created profile and to sweep it along a direction. The *Prism* functionality of Open CASCADE Technology is the most appropriate for that task. It accepts a shape and a direction as input and generates a new shape according to the following rules:

<table>
<thead>
<tr>
<th>Shape</th>
<th>Generates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>Edge</td>
</tr>
<tr>
<td>Edge</td>
<td>Face</td>
</tr>
<tr>
<td>Wire</td>
<td>Shell</td>
</tr>
<tr>
<td>Face</td>
<td>Solid</td>
</tr>
<tr>
<td>Shell</td>
<td>Compound of Solids</td>
</tr>
</tbody>
</table>

Your current profile is a wire. Referring to the Shape/Generates table, you need to compute a face out of its wire to generate a solid. To create a face, use the *BRepBuilderAPI_MakeFace* class. As previously explained, a face is a part of a surface bounded by a closed wire. Generally, *BRepBuilderAPI_MakeFace* computes a face out of a surface and one or more wires. When the wire lies on a plane, the surface is automatically computed.
TopoDS_Face myFaceProfile = 
    BRepBuilderAPI_MakeFace(myWireProfile);

The BRepPrimAPI package provides all the classes to create topological primitive constructions: boxes, cones, cylinders, spheres, etc. Among them is the BRepPrimAPI_MakePrism class. As specified above, the prism is defined by:

- the basis shape to sweep;
- a vector for a finite prism or a direction for finite and infinite prisms.

You want the solid to be finite, swept along the Z axis and to be myHeight height. The vector, defined with the gp_Vec class on its X, Y and Z coordinates, is:

gp_Vec aPrismVec(0, 0, myHeight);

All the necessary data to create the main body of your bottle is now available. Just apply the BRepPrimAPI_MakePrism class to compute the solid:

TopoDS_Shape myBody = 
    BRepPrimAPI_MakePrism(myFaceProfile, aPrismVec);
Applying Fillets

The edges of the bottle's body are very sharp. To replace them by rounded faces, you use the Fillet functionality of Open CASCADe Technology. For our purposes, we will specify that fillets must be:

- applied on all edges of the shape
- have a radius of $\text{myThickness} / 12$

To apply fillets on the edges of a shape, you use the `BRepFilletAPI_MakeFillet` class. This class is normally used as follows:

- Specify the shape to be filleted in the `BRepFilletAPI_MakeFillet` constructor.
- Add the fillet descriptions (an edge and a radius) using the `Add` method (you can add as many edges as you need).
- Ask for the resulting filleted shape with the `Shape` method.

```cpp
BRepFilletAPI_MakeFillet mkFillet(myBody);
```

To add the fillet description, you need to know the edges belonging to your shape. The best solution is to explore your solid to retrieve its edges. This kind of functionality is provided with the `TopExp_Explorer` class, which explores the data structure described in a `TopoDS_Shape` and extracts the sub-shapes you specifically need. Generally, this explorer is created by providing the following information:
• the shape to explore
• the type of sub-shapes to be found. This information is given with the `TopAbs_ShapeEnum` enumeration.

```c++
TopExp_Explorer anEdgeExplorer(myBody, TopAbs_EDGE);
```

An explorer is usually applied in a loop by using its three main methods:

• `More()` to know if there are more sub-shapes to explore.
• `Current()` to know which is the currently explored sub-shape (used only if the `More()` method returns true).
• `Next()` to move onto the next sub-shape to explore.

```c++
while(anEdgeExplorer.More()){  
    TopoDS_Edge anEdge = 
        TopoDS::Edge(anEdgeExplorer.Current());  
    //Add edge to fillet algorithm
    ...
    anEdgeExplorer.Next();
}
```

In the explorer loop, you have found all the edges of the bottle shape. Each one must then be added in the `BRepFilletAPI_MakeFillet` instance with the `Add()` method. Do not forget to specify the radius of the fillet along with it.

```c++
mkFillet.Add(myThickness / 12., anEdge);
```

Once this is done, you perform the last step of the procedure by asking for the filleted shape.

```c++
myBody = mkFillet.Shape();
```
Adding the Neck

To add a neck to the bottle, you will create a cylinder and fuse it to the body. The cylinder is to be positioned on the top face of the body with a radius of $myThickness / 4.$ and a height of $myHeight / 10.$

To position the cylinder, you need to define a coordinate system with the `gp_Ax2` class defining a right-handed coordinate system from a point and two directions - the main (Z) axis direction and the X direction (the Y direction is computed from these two). To align the neck with the center of the top face, being in the global coordinate system $(0, 0, myHeight),$ with its normal on the global Z axis, your local coordinate system can be defined as follows:

```plaintext
gp_Pnt neckLocation(0, 0, myHeight);
gp_Dir neckAxis = gp::DZ();
gp_Ax2 neckAx2(neckLocation, neckAxis);
```

To create a cylinder, use another class from the primitives construction package: the `BRepPrimAPI_MakeCylinder` class. The information you must provide is:

- the coordinate system where the cylinder will be located;
- the radius and height.

```plaintext
Standard_Real myNeckRadius = myThickness / 4.;
```
Standard_Real myNeckHeight = myHeight / 10;
BRepPrimAPI_MakeCylinder MKCylinder(neckAx2,  
    myNeckRadius, myNeckHeight);
TopoDS_Shape myNeck = MKCylinder.Shape();

You now have two separate parts: a main body and a neck that you need to fuse together. The \textit{BRepAlgoAPI} package provides services to perform Boolean operations between shapes, and especially: \textit{common} (Boolean intersection), \textit{cut} (Boolean subtraction) and \textit{fuse} (Boolean union). Use \textit{BRepAlgoAPI\_Fuse} to fuse the two shapes:

\begin{verbatim}
myBody = BRepAlgoAPI_Fuse(myBody, myNeck);
\end{verbatim}
Creating a Hollowed Solid

Since a real bottle is used to contain liquid material, you should now create a hollowed solid from the bottle’s top face. In Open CASCADE Technology, a hollowed solid is called a *Thick Solid* and is internally computed as follows:

- Remove one or more faces from the initial solid to obtain the first wall W1 of the hollowed solid.
- Create a parallel wall W2 from W1 at a distance D. If D is positive, W2 will be outside the initial solid, otherwise it will be inside.
- Compute a solid from the two walls W1 and W2.

To compute a thick solid, you create an instance of the `BRepOffsetAPI_MakeThickSolid` class by giving the following information:

- The shape, which must be hollowed.
- The tolerance used for the computation (tolerance criterion for coincidence in generated shapes).
- The thickness between the two walls W1 and W2 (distance D).
- The face(s) to be removed from the original solid to compute the first wall W1.

The challenging part in this procedure is to find the face to remove from your shape - the top face of the neck, which:

- has a plane (planar surface) as underlying geometry;
is the highest face (in Z coordinates) of the bottle.

To find the face with such characteristics, you will once again use an explorer to iterate on all the bottle's faces to find the appropriate one.

```cpp
for(TopExp_Explorer aFaceExplorer(myBody,
    TopAbs_FACE); aFaceExplorer.More();
    aFaceExplorer.Next()){
    TopoDS_Face aFace =
    TopoDS::Face(aFaceExplorer.Current());
}
```

For each detected face, you need to access the geometric properties of the shape: use the `BRep_Tool` class for that. The most commonly used methods of this class are:

- `Surface` to access the surface of a face;
- `Curve` to access the 3D curve of an edge;
- `Point` to access the 3D point of a vertex.

```cpp
Handle(Geom_Surface) aSurface =
    BRep_Tool::Surface(aFace);
```

As you can see, the `BRep_Tool::Surface` method returns an instance of the `Geom_Surface` class manipulated by handle. However, the `Geom_Surface` class does not provide information about the real type of the object `aSurface`, which could be an instance of `Geom_Plane`, `Geom_CylindricalSurface`, etc. All objects manipulated by handle, like `Geom_Surface`, inherit from the `Standard_Transient` class which provides two very useful methods concerning types:

- `DynamicType` to know the real type of the object
- `IsKind` to know if the object inherits from one particular type

`DynamicType` returns the real type of the object, but you need to compare it with the existing known types to determine whether `aSurface` is a plane, a cylindrical surface or some other type. To compare a given type with the type you seek, use the `STANDARD_TYPE` macro, which returns the type of a class:
If this comparison is true, you know that the `aSurface` real type is `Geom_Plane`. You can then convert it from `Geom_Surface` to `Geom_Plane` by using the `DownCast()` method provided by each class inheriting `Standard_Transient`. As its name implies, this static method is used to downcast objects to a given type with the following syntax:

```cpp
Handle(Geom_Plane) aPlane = Handle(Geom_Plane)::DownCast(aSurface);
```

Remember that the goal of all these conversions is to find the highest face of the bottle lying on a plane. Suppose that you have these two global variables:

```cpp
TopoDS_Face faceToRemove;
Standard_Real zMax = -1;
```

You can easily find the plane whose origin is the biggest in Z knowing that the location of the plane is given with the `Geom_Plane::Location` method. For example:

```cpp
gp_Pnt aPnt = aPlane->Location();
Standard_Real aZ = aPnt.Z();
if (aZ > zMax){
    zMax = aZ;
    faceToRemove = aFace;
}
```

You have now found the top face of the neck. Your final step before creating the hollowed solid is to put this face in a list. Since more than one face can be removed from the initial solid, the `BRepOffsetAPI_MakeThickSolid` constructor takes a list of faces as arguments. Open CASCADE Technology provides many collections for different kinds of objects: see `TColGeom` package for collections of objects from `Geom` package, `TColgp` package for collections of objects
from gp package, etc. The collection for shapes can be found in the *TopTools* package. As *BRepOffsetAPI_MakeThickSolid* requires a list, use the *TopTools_ListOfShape* class.

```plaintext
TopTools_ListOfShape facesToRemove;
facesToRemove.Append(faceToRemove);
```

All the necessary data are now available so you can create your hollowed solid by calling the *BRepOffsetAPI_MakeThickSolid* MakeThickSolidByJoin method:

```plaintext
BRepOffsetAPI_MakeThickSolid BodyMaker;
BodyMaker.MakeThickSolidByJoin(myBody, facesToRemove, -myThickness / 50, 1.e-3);
myBody = BodyMaker.Shape();
```
Building the Threading

Creating Surfaces

Up to now, you have learned how to create edges out of 3D curves. You will now learn how to create an edge out of a 2D curve and a surface. To learn this aspect of Open CASCADE Technology, you will build helicoidal profiles out of 2D curves on cylindrical surfaces. The theory is more complex than in previous steps, but applying it is very simple. As a first step, you compute these cylindrical surfaces. You are already familiar with curves of the Geom package. Now you can create a cylindrical surface (Geom_CylindricalSurface) using:

- a coordinate system;
- a radius.

Using the same coordinate system neckAx2 used to position the neck, you create two cylindrical surfaces Geom_CylindricalSurface with the following radii:

\[
\text{myNeckRadius} \times 0.99
\]

\[
\text{myNeckRadius} \times 1.05
\]

Notice that one of the cylindrical surfaces is smaller than the neck. There is a good reason for this: after the thread creation, you will fuse it with the neck. So, we must make sure that the two shapes remain in contact.

```cpp
Handle(Geom_CylindricalSurface) aCyl1 = new Geom_CylindricalSurface(neckAx2, myNeckRadius * 0.99);

Handle(Geom_CylindricalSurface) aCyl2 = new
```
Geom_CylindricalSurface(neckAx2, myNeckRadius * 1.05);
Defining 2D Curves

To create the neck of the bottle, you made a solid cylinder based on a cylindrical surface. You will create the profile of threading by creating 2D curves on such a surface. All geometries defined in the Geom package are parameterized. This means that each curve or surface from Geom is computed with a parametric equation. A Geom\_CylindricalSurface surface is defined with the following parametric equation:

\[ P(U, V) = O + R \times (\cos(U) \times xDir + \sin(U) \times yDir) + V \times zDir, \]

where:

- \( P \) is the point defined by parameters \((U, V)\).
- \( O, xDir, yDir \) and \( zDir \) are respectively the origin, the \( X \) direction, \( Y \) direction and \( Z \) direction of the cylindrical surface local coordinate system.
- \( R \) is the radius of the cylindrical surface.
- \( U \) range is \([0, 2\pi]\) and \( V \) is infinite.

The advantage of having such parameterized geometries is that you can compute, for any \((U, V)\) parameters of the surface:

- the 3D point;
- the derivative vectors of order 1, 2 to \( N \) at this point.

There is another advantage of these parametric equations: you can consider a surface as a 2D parametric space defined with a \((U, V)\) coordinate system. For example, consider the parametric ranges of the neck's surface:
Suppose that you create a 2D line on this parametric (U, V) space and compute its 3D parametric curve. Depending on the line definition, results are as follows:

<table>
<thead>
<tr>
<th>Case</th>
<th>Parametric Equation</th>
<th>Parametric Curve</th>
</tr>
</thead>
<tbody>
<tr>
<td>U = 0</td>
<td>P(V) = O + V * zDir</td>
<td>Line parallel to the Z direction</td>
</tr>
<tr>
<td>V = 0</td>
<td>P(U) = O + R * (cos(U) * xDir + sin(U) * yDir)</td>
<td>Circle parallel to the (O, X, Y) plane</td>
</tr>
<tr>
<td>U ≠ 0</td>
<td>P(U, V) = O + R * (cos(U) * xDir + sin(U) * yDir) + V * zDir</td>
<td>Helicoidal curve describing the evolution of height and angle on the cylinder</td>
</tr>
</tbody>
</table>

The helicoidal curve type is exactly what you need. On the neck's surface, the evolution laws of this curve will be:

- In V parameter: between 0 and myHeighNeck for the height description
- In U parameter: between 0 and 2PI for the angle description. But, since a cylindrical surface is U periodic, you can decide to extend this angle evolution to 4PI as shown in the following drawing:
In this (U, V) parametric space, you will create a local (X, Y) coordinate system to position the curves to be created. This coordinate system will be defined with:

- A center located in the middle of the neck's cylinder parametric space at (2*PI, myNeckHeight / 2) in U, V coordinates.
- A X direction defined with the (2*PI, myNeckHeight/4) vector in U, V coordinates, so that the curves occupy half of the neck's surfaces.

To use 2D primitive geometry types of Open CASCADE Technology for defining a point and a coordinate system, you will once again instantiate classes from gp:

- To define a 2D point from its X and Y coordinates, use the *gp_Pnt2d* class.
- To define a 2D direction (unit vector) from its X and Y coordinates, use the *gp_Dir2d* class. The coordinates will automatically be normalized.
- To define a 2D right-handed coordinate system, use the *gp_Ax2d* class, which is computed from a point (origin of the coordinate system) and a direction - the X direction of the coordinate system.
The Y direction will be automatically computed.

```cpp
gp_Pnt2d aPnt(2.*M_PI, myNeckHeight/2.);
gp_Dir2d aDir(2.*M_PI, myNeckHeight/4.);
gp_Ax2d anAx2d(aPnt, aDir);
```

You will now define the curves. As previously mentioned, these thread profiles are computed on two cylindrical surfaces. In the following figure, curves on the left define the base (on aCyl1 surface) and the curves on the right define the top of the thread's shape (on aCyl2 surface).

You have already used the `Geom` package to define 3D geometric entities. For 2D, you will use the `Geom2d` package. As for `Geom`, all geometries are parameterized. For example, a `Geom2d_Ellipse` ellipse is defined from:

- a coordinate system whose origin is the ellipse center;
- a major radius on the major axis defined by the X direction of the coordinate system;
- a minor radius on the minor axis defined by the Y direction of the coordinate system.

Supposing that:

- Both ellipses have the same major radius of `2*PI`,
- Minor radius of the first ellipse is `myNeckHeight/10`,
- And the minor radius value of the second ellipse is a fourth of the first one,

Your ellipses are defined as follows:

```cpp
Standard_Real aMajor = 2.*M_PI;
Standard_Real aMinor = myNeckHeight/10;
Handle(Geom2d_Ellipse) anEllipse1 = new Geom2d_Ellipse(anAx2d, aMajor, aMinor);
```
Handle(Geom2d_Ellipse) anEllipse2 = new Geom2d_Ellipse(anAx2d, aMajor, aMinor / 4);

To describe portions of curves for the arcs drawn above, you define `Geom2d_TrimmedCurve` trimmed curves out of the created ellipses and two parameters to limit them. As the parametric equation of an ellipse is $P(U) = O + (\text{MajorRadius} \times \cos(U) \times \text{XDirection}) + (\text{MinorRadius} \times \sin(U) \times \text{YDirection})$, the ellipses need to be limited between 0 and M_PI.

Handle(Geom2d_TrimmedCurve) anArc1 = new Geom2d_TrimmedCurve(anEllipse1, 0, M_PI);
Handle(Geom2d_TrimmedCurve) anArc2 = new Geom2d_TrimmedCurve(anEllipse2, 0, M_PI);

The last step consists in defining the segment, which is the same for the two profiles: a line limited by the first and the last point of one of the arcs. To access the point corresponding to the parameter of a curve or a surface, you use the Value or D0 method (meaning 0th derivative), D1 method is for first derivative, D2 for the second one.

gp_Pnt2d anEllipsePnt1 = anEllipse1->Value(0);
gp_Pnt2d anEllipsePnt2;
anEllipse1->D0(M_PI, anEllipsePnt2);

When creating the bottle's profile, you used classes from the GC package, providing algorithms to create elementary geometries. In 2D geometry, this kind of algorithms is found in the GCE2d package. Class names and behaviors are similar to those in GC. For example, to create a 2D segment out of two points:

Handle(Geom2d_TrimmedCurve) aSegment =
    GCE2d_MakeSegment(anEllipsePnt1, anEllipsePnt2);
Building Edges and Wires

As you did when creating the base profile of the bottle, you can now:

- compute the edges of the neck's threading.
- compute two wires out of these edges.

Previously, you have built:

- two cylindrical surfaces of the threading
- three 2D curves defining the base geometry of the threading

To compute the edges out of these curves, once again use the `BRepBuilderAPI_MakeEdge` class. One of its constructors allows you to build an edge out of a curve described in the 2D parametric space of a surface.

```cpp
TopoDS_Edge anEdge1OnSurf1 = BRepBuilderAPI_MakeEdge(anArc1, aCyl1);
TopoDS_Edge anEdge2OnSurf1 = BRepBuilderAPI_MakeEdge(aSegment, aCyl1);
TopoDS_Edge anEdge1OnSurf2 = BRepBuilderAPI_MakeEdge(anArc2, aCyl2);
TopoDS_Edge anEdge2OnSurf2 = BRepBuilderAPI_MakeEdge(aSegment, aCyl2);
```

Now, you can create the two profiles of the threading, lying on each surface.

```cpp
TopoDS_Wire threadingWire1 = BRepBuilderAPI_MakeWire(anEdge1OnSurf1,
```
Remember that these wires were built out of a surface and 2D curves. One important data item is missing as far as these wires are concerned: there is no information on the 3D curves. Fortunately, you do not need to compute this yourself, which can be a difficult task since the mathematics can be quite complex. When a shape contains all the necessary information except 3D curves, Open CASCADE Technology provides a tool to build them automatically. In the BRepLib tool package, you can use the `BuildCurves3d` method to compute 3D curves for all the edges of a shape.

```cpp
BRepLib::BuildCurves3d(threadingWire1);
BRepLib::BuildCurves3d(threadingWire2);
```
Creating Threading

You have computed the wires of the threading. The threading will be a solid shape, so you must now compute the faces of the wires, the faces allowing you to join the wires, the shell out of these faces and then the solid itself. This can be a lengthy operation. There are always faster ways to build a solid when the base topology is defined. You would like to create a solid out of two wires. Open CASCADE Technology provides a quick way to do this by building a loft: a shell or a solid passing through a set of wires in a given sequence. The loft function is implemented in the BRepOffsetAPI_ThruSections class, which you use as follows:

- Initialize the algorithm by creating an instance of the class. The first parameter of this constructor must be specified if you want to create a solid. By default, BRepOffsetAPI_ThruSections builds a shell.
- Add the successive wires using the AddWire method.
- Use the CheckCompatibility method to activate (or deactivate) the option that checks whether the wires have the same number of edges. In this case, wires have two edges each, so you can deactivate this option.
- Ask for the resulting loft shape with the Shape method.

```plaintext
BRepOffsetAPI_ThruSections aTool(Standard_True);
aTool.AddWire(threadingWire1);
    aTool.AddWire(threadingWire2);
aTool.CheckCompatibility(Standard_False);
TopoDS_Shape myThreading = aTool.Shape();
```
Building the Resulting Compound

You are almost done building the bottle. Use the `TopoDS_Compound` and `BRep_Builder` classes to build single shape from `myBody` and `myThreading`:

```cpp
TopoDS_Compound aRes;
BRep_Builder aBuilder;
aBuilder.MakeCompound (aRes);
aBuilder.Add (aRes, myBody);
aBuilder.Add (aRes, myThreading);
```

Congratulations! Your bottle is complete. Here is the result snapshot of the Tutorial application:

We hope that this tutorial has provided you with a feel for the industrial strength power of Open CASCADE Technology. If you want to know more and develop major projects using Open CASCADE Technology, we invite you to study our training, support, and consulting services on our site at [http://www.opencascade.com/content/technology-support](http://www.opencascade.com/content/technology-support). Our
professional services can maximize the power of your Open CASCADE Technology applications.
Appendix

Complete definition of MakeBottle function (defined in the file src/MakeBottle.cxx of the Tutorial):

```cpp
TopoDS_Shape MakeBottle(const Standard_Real myWidth, const Standard_Real myHeight, const Standard_Real myThickness)
{
    // Profile : Define Support Points
    gp_Pnt aPnt1(-myWidth / 2., 0, 0);
    gp_Pnt aPnt2(-myWidth / 2., -myThickness / 4., 0);
    gp_Pnt aPnt3(0, -myThickness / 2., 0);
    gp_Pnt aPnt4(myWidth / 2., -myThickness / 4., 0);
    gp_Pnt aPnt5(myWidth / 2., 0, 0);

    // Profile : Define the Geometry
    Handle(Geom_TrimmedCurve) anArcOfCircle = GC_MakeArcOfCircle(aPnt2, aPnt3, aPnt4);
    Handle(Geom_TrimmedCurve) aSegment1 = GC_MakeSegment(aPnt1, aPnt2);
    Handle(Geom_TrimmedCurve) aSegment2 = GC_MakeSegment(aPnt4, aPnt5);

    // Profile : Define the Topology
    TopoDS_Edge anEdge1 = BRepBuilderAPI_MakeEdge(aSegment1);
    TopoDS_Edge anEdge2 = BRepBuilderAPI_MakeEdge(anArcOfCircle);
    TopoDS_Edge anEdge3 = BRepBuilderAPI_MakeEdge(aSegment2);
    TopoDS_Wire aWire = BRepBuilderAPI_MakeWire(anEdge1, anEdge2, anEdge3);
```
// Complete Profile
    gp_Ax1 xAxis = gp::OX();
    gp_Trsf aTrsf;

    aTrsf.SetMirror(xAxis);
    BRepBuilderAPI_Transform aBRepTrsf(aWire, aTrsf);
    TopoDS_Shape aMirroredShape = aBRepTrsf.Shape();
    TopoDS_Wire aMirroredWire =
        TopoDS::Wire(aMirroredShape);

    BRepBuilderAPI_MakeWire mkWire;
    mkWire.Add(aWire);
    mkWire.Add(aMirroredWire);
    TopoDS_Wire myWireProfile = mkWire.Wire();

// Body : Prism the Profile
    TopoDS_Face myFaceProfile =
        BRepBuilderAPI_MakeFace(myWireProfile);
    gp_Vec aPrismVec(0, 0, myHeight);
    TopoDS_Shape myBody =
        BRepPrimAPI_MakePrism(myFaceProfile, aPrismVec);

// Body : Apply Fillets
    BRepFilletAPI_MakeFillet mkFillet(myBody);
    TopExp_Explorer anEdgeExplorer(myBody,
        TopAbs_EDGE);
    while(anEdgeExplorer.More()){
        TopoDS_Edge anEdge =
            TopoDS::Edge(anEdgeExplorer.Current());
        //Add edge to fillet algorithm
            mkFillet.Add(myThickness / 12., anEdge);
        anEdgeExplorer.Next();
    }
    myBody = mkFillet.Shape();
// Body : Add the Neck
gp_Pnt neckLocation(0, 0, myHeight);
gp_Dir neckAxis = gp::DZ();
gp_Ax2 neckAx2(neckLocation, neckAxis);

Standard_Real myNeckRadius = myThickness / 4.;
Standard_Real myNeckHeight = myHeight / 10.;

BRepPrimAPI_MakeCylinder MKCylinder(neckAx2,
myNeckRadius, myNeckHeight);
TopoDS_Shape myNeck = MKCylinder.Shape();

myBody = BRepAlgoAPI_Fuse(myBody, myNeck);

// Body : Create a Hollowed Solid
TopoDS_Face faceToRemove;
Standard_Real zMax = -1;

for(TopExp_Explorer aFaceExplorer(myBody,
TopAbs_FACE); aFaceExplorer.More();
aFaceExplorer.Next()){
    TopoDS_Face aFace =
    TopoDS::Face(aFaceExplorer.Current());
// Check if <aFace> is the top face of the bottle's neck
    Handle(Geom_Surface) aSurface =
    BRep_Tool::Surface(aFace);
    if(aSurface->DynamicType() ==
    STANDARD_TYPE(Geom_Plane)){
        Handle(Geom_Plane) aPlane =
        Handle(Geom_Plane)::DownCast(aSurface);
        gp_Pnt aPnt = aPlane->Location();
        Standard_Real aZ = aPnt.Z();
        if(aZ > zMax){
            zMax = aZ;
            faceToRemove = aFace;
        }
    }
TopTools_ListOfShape facesToRemove;
facesToRemove.Append(faceToRemove);
BRepOffsetAPI_MakeThickSolid BodyMaker;
BodyMaker.MakeThickSolidByJoin(myBody,
facesToRemove, -myThickness / 50, 1.e-3);
myBody = BodyMaker.Shape();

// Threading : Create Surfaces
Handle(Geom_CylindricalSurface) aCyl1 = new
Geom_CylindricalSurface(neckAx2, myNeckRadius * 0.99);
Handle(Geom_CylindricalSurface) aCyl2 = new
Geom_CylindricalSurface(neckAx2, myNeckRadius * 1.05);

// Threading : Define 2D Curves
gp_Pnt2d aPnt(2. * M_PI, myNeckHeight / 2.);
gp_Dir2d aDir(2. * M_PI, myNeckHeight / 4.);
gp_Ax2d anAx2d(aPnt, aDir);

Standard_Real aMajor = 2. * M_PI;
Standard_Real aMinor = myNeckHeight / 10;

Handle(Geom2d_Ellipse) anEllipse1 = new
Geom2d_Ellipse(anAx2d, aMajor, aMinor);
Handle(Geom2d_Ellipse) anEllipse2 = new
Geom2d_Ellipse(anAx2d, aMajor, aMinor / 4);
Handle(Geom2d_TrimmedCurve) anArc1 = new
Geom2d_TrimmedCurve(anEllipse1, θ, M_PI);
Handle(Geom2d_TrimmedCurve) anArc2 = new
Geom2d_TrimmedCurve(anEllipse2, θ, M_PI);
gp_Pnt2d anEllipsePnt1 = anEllipse1->Value(0);
gp_Pnt2d anEllipsePnt2 = anEllipse1->Value(M_PI);

Handle(Geom2d_TrimmedCurve) aSegment =
GCE2d_MakeSegment(anEllipsePnt1, anEllipsePnt2);

// Threading : Build Edges and Wires
TopoDS_Edge anEdge1OnSurf1 =
BRepBuilderAPI_MakeEdge(anArc1, aCyl1);
TopoDS_Edge anEdge2OnSurf1 =
BRepBuilderAPI_MakeEdge(aSegment, aCyl1);
TopoDS_Edge anEdge1OnSurf2 =
BRepBuilderAPI_MakeEdge(anArc2, aCyl2);
TopoDS_Edge anEdge2OnSurf2 =
BRepBuilderAPI_MakeEdge(aSegment, aCyl2);
TopoDS_Wire threadingWire1 =
BRepBuilderAPI_MakeWire(anEdge1OnSurf1, anEdge2OnSurf1);
TopoDS_Wire threadingWire2 =
BRepBuilderAPI_MakeWire(anEdge1OnSurf2, anEdge2OnSurf2);
BRepLib::BuildCurves3d(threadingWire1);
BRepLib::BuildCurves3d(threadingWire2);

// Create Threading
BRepOffsetAPI_ThruSections aTool(Standard_True);
aTool.AddWire(threadingWire1);
aTool.AddWire(threadingWire2);
aTool.CheckCompatibility(Standard_False);

TopoDS_Shape myThreading = aTool.Shape();

// Building the Resulting Compound
TopoDS_Compound aRes;
BRep_Builder aBuilder;
aBuilder.MakeCompound (aRes);
aBuilder.Add (aRes, myBody);
aBuilder.Add (aRes, myThreading);

return aRes;
Open CASCADE Technology 7.2.0

Technical Overview

Open CASCADE Technology (OCCT) is an object-oriented C++ class library designed for rapid production of sophisticated domain-specific CAD/CAM/CAE applications.

A typical application developed using OCCT deals with two or three-dimensional (2D or 3D) geometric modeling in general-purpose or specialized Computer Aided Design (CAD) systems, manufacturing or analysis applications, simulation applications, or even illustration tools.

OCCT library is designed to be truly modular and extensible, providing C++ classes for:

- Basic data structures (geometric modeling, visualization, interactive selection and application specific services);
- Modeling algorithms;
- Working with mesh (faceted) data;
- Data interoperability with neutral formats (IGES, STEP);

The C++ classes and other types are grouped into packages. Packages are organized into toolkits (libraries), to which you can link your application. Finally, toolkits are grouped into seven modules.
This modular structure is illustrated in the diagram below.

- **Foundation Classes** module underlies all other OCCT classes;
- **Modeling Data** module supplies data structures to represent 2D and 3D geometric primitives and their compositions into CAD models;
- **Modeling Algorithms** module contains a vast range of geometrical and topological algorithms;
- **Mesh** module implements tessellated representations of objects;
- **Visualization** module provides complex mechanisms for graphical data representation;
- **Data Exchange** module inter-operates with popular data formats and relies on **Shape Healing** to improve compatibility between CAD software of different vendors;
- **Application Framework** module offers ready-to-use solutions for handling application-specific data (user attributes) and commonly used functionality (save/restore, undo/redo, copy/paste, tracking CAD modifications, etc).

In addition, **Open CASCADE Test Harness**, also called Draw, provides an entry point to the library and can be used as a testing tool for its modules.
Foundation Classes

**Foundation Classes** module contains data structures and services used by higher-level Open CASCADE Technology classes:

- Primitive types, such as Boolean, Character, Integer or Real;
- String classes that handle ASCII and Unicode strings;
- Collection classes that handle statically or dynamically sized aggregates of data, such as arrays, lists, queues, sets and hash tables (data maps);
- Classes providing commonly used numerical algorithms and basic linear algebra calculations (addition, multiplication, transposition of vectors and matrices, solving linear systems etc);
- Fundamental types representing physical quantities and supporting date and time information;
- Primitive geometry types providing implementation of basic geometric and algebraic entities that define and manipulate elementary data structures.
- Exception classes that describe situations, when the normal execution of program is abandoned;

This module also provides a variety of general-purpose services, such as:

- Safe handling of dynamically created objects, ensuring automatic deletion of unreferenced objects (smart pointers);
- Configurable optimized memory manager increasing the performance of applications that intensively use dynamically created objects;
- Extended run-time type information (RTTI) mechanism maintaining a full type hierarchy and providing means to iterate over it;
- Encapsulation of C++ streams;
- Automated management of heap memory by means of specific allocators;
- Basic interpreter of expressions facilitating the creation of customized scripting tools, generic definition of expressions, etc.;
- Tools for dealing with configuration resource files and customizable message files facilitating multi-language support in applications;
• Progress indication and user break interfaces, giving a possibility even for low-level algorithms to communicate with the user in a universal and convenient way;
• and many others...

Please, see the details in Foundation Classes User's Guide

See also: our E-learning & Training offerings.
Modeling Data

Modeling Data supplies data structures to implement boundary representation (BRep) of objects in 3D. In BRep the shape is represented as an aggregation of geometry within topology. The geometry is understood as a mathematical description of a shape, e.g. as curves and surfaces (simple or canonical, Bezier, NURBS, etc). The topology is a data structure binding geometrical objects together.

Geometry types and utilities provide geometric data structures and services for:

- Description of points, vectors, curves and surfaces:
  - their positioning in 3D space using axis or coordinate systems, and
  - their geometric transformation, by applying translations, rotations, symmetries, scaling transformations and combinations thereof.
- Creation of parametric curves and surfaces by interpolation and approximation;
- Algorithms of direct construction;
- Conversion of curves and surfaces to NURBS form;
- Computation of point coordinates on 2D and 3D curves;
- Calculation of extrema between geometric objects.

Topology defines relationships between simple geometric entities. A shape, which is a basic topological entity, can be divided into components (sub-shapes):

- Vertex – a zero-dimensional shape corresponding to a point;
- Edge – a shape corresponding to a curve and bounded by a vertex at each extremity;
- Wire – a sequence of edges connected by their vertices;
- Face – a part of a plane (in 2D) or a surface (in 3D) bounded by wires;
- Shell – a collection of faces connected by edges of their wire boundaries;
- Solid – a finite closed part of 3D space bounded by shells;
- Compound solid – a collection of solids connected by faces of their
shell boundaries.

Complex shapes can be defined as assemblies of simpler entities.

Please, see the details in *Modeling Data User's Guide*

3D geometric models can be stored in OCCT native BREP format. See *BREP Format Description White Paper* for details on the format.

See also: our *E-learning & Training* offerings.
Modeling Algorithms

Modeling Algorithms module groups a wide range of topological and geometric algorithms used in geometric modeling. Basically, there are two groups of algorithms in Open CASCADE Technology:

- High-level modeling routines used in the real design;
- Low-level mathematical support functions used as a groundwork for the modeling API;
- Low-level geometric tools provide the algorithms, which:
  - Calculate the intersection of two curves, surfaces, or a curve and a surface;
  - Project points onto 2D and 3D curves, points onto surfaces and 3D curves onto surfaces;
  - Construct lines and circles from constraints;
  - Construct free-form curves and surfaces from constraints (interpolation, approximation, skinning, gap filling, etc);
- Low-level topological tools provide the algorithms, which:
  - Tessellate shapes;
  - Check correct definition of shapes;
  - Determine the local and global properties of shapes (derivatives, mass-inertia properties, etc);
  - Perform affine transformations;
  - Find planes in which edges are located;
  - Convert shapes to NURBS geometry;
  - Sew connected topologies (shells and wires) from separate topological elements (faces and edges).

Top-level API provides the following functionality:

- Construction of Primitives:
  - Boxes;
  - Prisms;
  - Cylinders;
  - Cones;
  - Spheres;
  - Toruses.
- Kinematic Modeling:
• Prisms – linear sweeps;
• Revolutions – rotational sweeps;
• Pipes – general-form sweeps;
• Lofting.

Shapes containing pipes with variable radius produced by sweeping

• Boolean Operations, which allow creating new shapes from the combinations of source shapes. For two shapes $S_1$ and $S_2$:
  • $Common$ contains all points that are in $S_1$ and $S_2$;
  • $Fuse$ contains all points that are in $S_1$ or $S_2$;
  • $Cut$ contains all points in that are in $S_1$ and not in $S_2$


• Algorithms for local modifications such as:
  • Hollowing;
  • Shelling;
  • Creation of tapered shapes using draft angles;
  • Algorithms to make fillets and chamfers on shape edges, including those with variable radius (chord).
• Algorithms for creation of mechanical features, i.e. depressions, protrusions, ribs and grooves or slots along planar or revolution surfaces.
Please, see the details in *Modeling Algorithms User's Guide*.

See also: our *E-learning & Training* offerings.
Mesh

**Mesh** module provides the functionality to work with tessellated representations of objects in form of triangular facets. This module contains:

- data structures to store surface mesh data associated to shapes and basic algorithms to handle them;
- data structures and algorithms to build triangular surface mesh from *BRep* objects (shapes);
- tools for displaying meshes with associated pre- and post-processor data (scalars or vectors).

Open CASCADE Technology includes two mesh converters:

- VRML converter translates Open CASCADE shapes to VRML 1.0 files (Virtual Reality Modeling Language). Two representation modes are possible: shaded, which presents shapes as sets of triangles computed by the mesh algorithm, or wireframe, which presents shapes as sets of curves.
- STL converter translates Open CASCADE shapes to STL files. STL (STtereoLithography) format is widely used for rapid prototyping (3D printing).

Open CASCADE SAS also offers Advanced Mesh Products:

- **Open CASCADE Mesh Framework (OMF)**
- **Express Mesh**
Visualization

Visualization module provides ready-to-use algorithms to create graphic presentations from various objects: shapes, meshes, etc.

In Open CASCADE Technology visualization is based on the separation of CAD data and its graphical presentation. The presentations can be customized to take the specificity of your application into account.

The module also supports a fast and powerful interactive selection mechanism.

The view facilities provided by OCCT range from low-level tools working with basic geometry and topology (such as NURBS visualization with control points and nodes, rendering of isolines to estimate speed and quality of parameterization, or rendering of a parametric profile of edges) to high-level tools for real time quality rendering of models using ray tracing: shades, reflections, transparency, anti-aliasing, etc.

Here are just a few examples:

- Camera-driven view projection and orientation. It is possible to choose between perspective, orthographic and stereographic projection.
- Real-time ray tracing technique using recursive Whitted's algorithm and Bounded Volume Hierarchy effective optimization structure.
Real time visualization by ray tracing method

- Support of GLSL shaders. The shader management is fully automatic, like with any other OpenGL resource.
Fragment shader implementing custom clipping surface

- Support of standard and custom materials, defined by transparency, diffuse, ambient and specular reflection and refraction index. The latter allows implementing transparent materials, such as glass, diamond and water.

Simulation of a glass cover

- Optimization of rendering performance through the algorithms of:
  - View frustum culling, which skips the presentation outside camera at the rendering stage and
  - Back face culling, which reduces the rendered number of triangles and eliminates artifacts at shape boundaries.
- Definition of clipping planes through the plane equation coefficients. Ability to define visual attributes for cross-section at the level or individual clipping planes. In the image below different parts of the rocket are clipped with different planes and hatched.
- Possibility to flexibly adjust appearance of dimensions in a 3D view. The 3D text object represents a given text string as a true 3D object in the model space.
Display of shape cross-section and dimensions

For more details see Visualization User's Guide.

The visualization of OCCT topological shapes by means of VTK library provided by VIS component is described in a separate VTK Integration Services User's Guide.

See also: our E-learning & Training offerings.
Data Exchange allows developing OCCT-based applications that can interact with other CAD systems by writing and reading CAD models to and from external data. The exchanges run smoothly regardless of the quality of external data or requirements to its internal representation, for example, to the data types, accepted geometric inaccuracies, etc.

Data Exchange is organized in a modular way as a set of interfaces that comply with various CAD formats: IGES, STEP, STL, VRML, etc. The interfaces allow software based on OCCT to exchange data with various
CAD/PDM software packages, maintaining a good level of interoperability.

- **Standardized Data Exchange** interfaces allow querying and examining the input file, converting its contents to a CAD model and running validity checks on a fully translated shape. The following formats are currently supported:
  - **STEP** (AP203: Mechanical Design, this covers General 3D CAD; AP214: Automotive Design)
  - **IGES** (up to 5.3)
  - VRML and STL meshes.
- **Extended data exchange** (XDE) allows translating additional attributes attached to geometric data (colors, layers, names, materials etc).
- **Advanced Data Exchange Components** are available in addition to standard Data Exchange interfaces to support interoperability and data adaptation (also using **Shape Healing**) with CAD software using the following proprietary formats:
  - ACIS SAT
  - Parasolid
  - DXF

These components are based on the same architecture as interfaces with STEP and IGES.
Shape Healing

**Shape Healing** library provides algorithms to correct and adapt the geometry and topology of shapes imported to OCCT from other CAD systems.

Shape Healing algorithms include, but are not limited to, the following operations:

- analyze shape characteristics and, in particular, identify the shapes that do not comply with OCCT geometry and topology validity rules by analyzing geometrical objects and topology:
  - check edge and wire consistency;
  - check edge order in a wire;
  - check the orientation of face boundaries;
  - analyze shape tolerances;
  - identify closed and open wires in a boundary.
- fix incorrect or incomplete shapes:
  - provide consistency between a 3D curve and its corresponding parametric curve;
  - repair defective wires;
  - fit the shapes to a user-defined tolerance value;
  - fill gaps between patches and edges.
- upgrade and change shape characteristics:
  - reduce curve and surface degree;
  - split shapes to obtain C1 continuity;
  - convert any types of curves or surfaces to Bezier or B-Spline curves or surfaces and back;
  - split closed surfaces and revolution surfaces.

Each sub-domain of Shape Healing has its own scope of functionality:

<table>
<thead>
<tr>
<th>Sub-domain</th>
<th>Description</th>
<th>Impact on the shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>Explores shape properties, computes shape features, detects</td>
<td>The shape itself is not</td>
</tr>
<tr>
<td>Action</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Fixing</td>
<td>Fixes the shape to meet the OCCT requirements.</td>
<td>The shape may change its original form: modification, removal or creation of sub-shapes, etc.)</td>
</tr>
<tr>
<td>Upgrade</td>
<td>Improves the shape to fit some particular algorithms.</td>
<td>The shape is replaced with a new one, but geometrically they are the same.</td>
</tr>
<tr>
<td>Customization</td>
<td>Modifies the shape representation to fit specific needs.</td>
<td>The shape is not modified, only the mathematical form of its internal representation is changed.</td>
</tr>
<tr>
<td>Processing</td>
<td>Mechanism of shape modification via a user-editable resource file.</td>
<td></td>
</tr>
</tbody>
</table>

For more details refer to **Shape Healing User's guide**.

See also: our **E-learning & Training** offerings.
Application Framework

**Open CASCADE Application Framework** (OCAF) handles Application Data basing on the Application/Document paradigm. It uses an associativity engine to simplify the development of a CAD application thanks to the following ready-to-use features and services:

- Data attributes managing the application data, which can be organized according to the development needs;
- Data storage and persistence (open/save);
- Possibility to modify and recompute attributes in documents. With OCAF it is easy to represent the history of modification and parametric dependencies within your model;
- Possibility to manage multiple documents;
- Predefined attributes common to CAD/CAM/CAE applications (e.g. to store dimensions);
- Undo-Redo and Copy-Paste functions.

Since OCAF handles the application structure, the only development task is the creation of application-specific data and GUIs.

OCAF differs from any other CAD framework in the organization of application data, as there the data structures are based on reference keys rather than on shapes. In a model, such attributes as shape data, color and material are attached to an invariant structure, which is deeper than the shapes. A shape object becomes the value of *Shape* attribute, in the same way as an integer number is the value of *Integer* attribute and a string is the value of *Name* attribute.

OCAF organizes and embeds these attributes in a document. OCAF documents, in their turn, are managed by an OCAF application.

For more details see **OCAF User's Guide**.

See also: our **E-learning & Training** offerings.
Draw Test Harness

Test Harness or Draw is a convenient testing tool for OCCT libraries. It can be used to test and prototype various algorithms before building an entire application. It includes:

- A command interpreter based on the TCL language;
- A number of 2D and 3D viewers;
- A set of predefined commands.

The viewers support operations such as zoom, pan, rotation and full-screen views.

The basic commands provide general-purpose services such as:

- Getting help;
- Evaluating a script from a file;
- Capturing commands in a file;
- Managing views;
- Displaying objects.

In addition, Test Harness provides commands to create and manipulate curves and surfaces (geometry) and shapes, access visualization services, work with OCAF documents, perform data exchange, etc.

You can add custom commands to test or demonstrate any new functionalities, which you develop.

For more details see Draw Test Harness Manual.

Generated on Wed Aug 30 2017 17:04:20 for Open CASCADE Technology by doxygen 1.8.13
User Guides

OCCT User Guides are organized by OCCT modules:

- **Foundation Classes**
- **Modeling Data**
  - BREP format description
- **Modeling Algorithms**
  - Boolean Operations
  - Shape Healing
- **Visualization**
  - VTK Integration Services
- **Data Exchange**
  - IGES translator
  - STEP translator
  - Extended Data Exchange (XDE)
- **Open CASCADE Application Framework (OCAF)**
  - TObj package
- **DRAW Test Harness**
- **Inspector**
Introduction
Basics
- Library organization
  - Modules and toolkits
  - Packages
  - Classes
  - Inheritance
Data Types
- Primitive Types
- Types manipulated by value
- Types manipulated by reference (handle)
- When is it necessary to use a handle?
Programming with Handles
- Handle Definition
- Type Management
- Using Handles to Create Objects
- Invoking
Introduction

This manual explains how to use Open CASCADE Technology (OCCT) Foundation Classes. It provides basic documentation on foundation classes. For advanced information on foundation classes and their applications, see our E-learning & Training offerings.

Foundation Classes provide a variety of general-purpose services such as automated dynamic memory management (manipulation of objects by handle), collections, exception handling, genericity by down-casting and plug-in creation.

Foundation Classes include the following:

Root Classes

Root classes are the basic data types and classes on which all the other classes are built. They provide:

- fundamental types such as Boolean, Character, Integer or Real,
- safe handling of dynamically created objects, ensuring automatic deletion of unreferenced objects (see Standard_Transient class),
- configurable optimized memory manager increasing the performance of applications that intensively use dynamically created objects,
- extended run-time type information (RTTI) mechanism facilitating the creation of complex programs,
- management of exceptions,
- encapsulation of C++ streams. Root classes are mainly implemented in Standard and MMgt packages.

Strings

Strings are classes that handle dynamically sized sequences of characters based on both ASCII (normal 8-bit character type) and Unicode (16-bit character type). Strings may also be manipulated by handles, and consequently be shared. Strings are implemented in the TCollection package.
Collections

Collections are the classes that handle dynamically sized aggregates of data. Collection classes are *generic*, that is, they define a structure and algorithms allowing to hold a variety of objects which do not necessarily inherit from a unique root class (similarly to C++ templates). When you need to use a collection of a given type of object, you must *instantiate* it for this specific type of element. Once this declaration is compiled, all functions available on the generic collection are available on your instantiated class.

Collections include a wide range of generic classes such as run-time sized arrays, lists, stacks, queues, sets and hash maps. Collections are implemented in the *TCollection* and *NCollection* packages.

Collections of Standard Objects

The *TColStd* package provides frequently used instantiations of generic classes from the *TCollection* package with objects from the *Standard* package or strings from the *TCollection* package.

Vectors and Matrices

These classes provide commonly used mathematical algorithms and basic calculations (addition, multiplication, transposition, inversion, etc.) involving vectors and matrices.

Primitive Geometric Types

Open CASCADE Technology primitive geometric types are a STEP-compliant implementation of basic geometric and algebraic entities. They provide:

- Descriptions of elementary geometric shapes:
  - Points,
  - Vectors,
  - Lines,
  - Circles and conics,
  - Planes and elementary surfaces,
  - Positioning of these shapes in space or in a plane by means of an
axis or a coordinate system,
- Definition and application of geometric transformations to these shapes:
  - Translations
  - Rotations
  - Symmetries
  - Scaling transformations
  - Composed transformations
- Tools (coordinates and matrices) for algebraic computation.

**Common Math Algorithms**

Open CASCADE Technology common math algorithms provide a C++ implementation of the most frequently used mathematical algorithms. These include:

- Algorithms to solve a set of linear algebraic equations,
- Algorithms to find the minimum of a function of one or more independent variables,
- Algorithms to find roots of one, or of a set, of non-linear equations,
- Algorithms to find the eigen-values and eigen-vectors of a square matrix.

**Exceptions**

A hierarchy of commonly used exception classes is provided, all based on class Failure, the root of exceptions. Exceptions describe exceptional situations, which can arise during the execution of a function. With the raising of an exception, the normal course of program execution is abandoned. The execution of actions in response to this situation is called the treatment of the exception.

**Quantities**

These are various classes supporting date and time information and fundamental types representing most physical quantities such as length, area, volume, mass, density, weight, temperature, pressure etc.

**Application services**
Foundation Classes also include implementation of several low-level services that facilitate the creation of customizable and user-friendly applications with Open CASCADE Technology. These include:

- Unit conversion tools, providing a uniform mechanism for dealing with quantities and associated physical units: check unit compatibility, perform conversions of values between different units and so on (see package UnitsAPI);
- Basic interpreter of expressions that facilitates the creation of customized scripting tools, generic definition of expressions and so on (see package ExprIntrp);
- Tools for dealing with configuration resource files (see package Resource) and customizable message files (see package Message), making it easy to provide a multi-language support in applications;
- Progress indication and user break interfaces, giving a possibility even for low-level algorithms to communicate with the user in a universal and convenient way.
Basics

This chapter deals with basic services such as library organization, persistence, data types, memory management, programming with handles, exception handling, genericity by downcasting and plug-in creation.
Library organization

This chapter introduces some basic concepts, which are used not only in Foundation Classes, but throughout the whole OCCT library.

Modules and toolkits

The whole OCCT library is organized in a set of modules. The first module, providing most basic services and used by all other modules, is called Foundation Classes and described by this manual.

Every module consists primarily of one or several toolkits (though it can also contain executables, resource units etc.). Physically a toolkit is represented by a shared library (e.g. .so or .dll). The toolkit is built from one or several packages.

Packages

A package groups together a number of classes which have semantic links. For example, a geometry package would contain Point, Line, and Circle classes. A package can also contain enumerations, exceptions and package methods (functions). In practice, a class name is prefixed with the name of its package e.g. Geom_Circle. Data types described in a package may include one or more of the following data types:

- Enumerations
- Object classes
- Exceptions
- Pointers to other object classes

Inside a package, two data types cannot bear the same name.
Methods are either functions or procedures. Functions return an object, whereas procedures only communicate by passing arguments. In both cases, when the transmitted object is an instance manipulated by a handle, its identifier is passed. There are three categories of methods:

- **Object constructor** Creates an instance of the described class. A class will have one or more object constructors with various different arguments or none.
- **Instance method** Operates on the instance which owns it.
- **Class method** Does not work on individual instances, only on the class itself.

**Classes**

The fundamental software component in object-oriented software development is the class. A class is the implementation of a data type. It
defines its **behavior** (the services offered by its functions) and its **representation** (the data structure of the class – the fields, which store its data).

Classes fall into three categories:

- Ordinary classes.
- Abstract classes. An **abstract class** cannot be instantiated. The purpose of having such classes is to have a given behavior shared by a hierarchy of classes and dependent on the implementation of the descendants. This is a way of guaranteeing a certain base of inherited behavior common to all the classes based on a particular deferred class.
- Template classes. A **template class** offers a set of functional behaviors to manipulate other data types. Instantiation of a template class requires that a data type is given for its argument(s).

**Inheritance**

The purpose of inheritance is to reduce the development workload. The inheritance mechanism allows a new class to be declared already containing the characteristics of an existing class. This new class can then be rapidly specialized for the task in hand. This avoids the necessity of developing each component “from scratch”. For example, having already developed a class `BankAccount` you could quickly specialize new classes: `SavingsAccount`, `LongTermDepositAccount`, `MoneyMarketAccount`, `RevolvingCreditAccount`, etc....

The corollary of this is that when two or more classes inherit from a parent (or ancestor) class, all these classes guarantee as a minimum the behavior of their parent (or ancestor). For example, if the parent class `BankAccount` contains the method `Print` which tells it to print itself out, then all its descendant classes guarantee to offer the same service.

One way of ensuring the use of inheritance is to declare classes at the top of a hierarchy as being **abstract**. In such classes, the methods are not implemented. This forces the user to create a new class which redefines the methods. This is a way of guaranteeing a certain minimum of behavior among descendant classes.
Data Types

An object-oriented language structures a system around data types rather than around the actions carried out on this data. In this context, an object is an instance of a data type and its definition determines how it can be used. Each data type is implemented by one or more classes, which make up the basic elements of the system.

The data types in Open CASCADE Technology fall into two categories:

- Data types manipulated by handle (or reference)
- Data types manipulated by value

A data type is implemented as a class. The class not only defines its data representation and the methods available on instances, but it also suggests how the instance will be manipulated.

- A variable of a type manipulated by value contains the instance itself.
- A variable of a type manipulated by handle contains a reference to the instance. The first examples of types manipulated by values are the predefined primitive types: `Boolean`, `Character`, `Integer`, `Real`, etc.
A variable of a type manipulated by handle which is not attached to an object is said to be **null**. To reference an object, we instantiate the class with one of its constructors. For example, in C++:

```cpp
Handle(myClass) m = new myClass;
```

In Open CASCADE Technology, the Handles are specific classes that are used to safely manipulate objects allocated in the dynamic memory by reference, providing reference counting mechanism and automatic destruction of the object when it is not referenced.

### Primitive Types

The primitive types are predefined in the language and they are **manipulated by value**.

- **Boolean** is used to represent logical data. It may have only two values: `Standard_True` and `Standard_False`.
- **Character** designates any ASCII character.
- **ExtCharacter** is an extended character.
- **Integer** is a whole number.
- **Real** denotes a real number (i.e. one with whole and a fractional part, either of which may be null).
- **ShortReal** is a real with a smaller choice of values and memory size.
- **CString** is used for literal constants.
- **ExtString** is an extended string.
- **Address** represents a byte address of undetermined size.

The services offered by each of these types are described in the **Standard** Package. The table below presents the equivalence existing between C++ fundamental types and OCCT primitive types.

### Table 1: Equivalence between C++ Types and OCCT Primitive Types

<table>
<thead>
<tr>
<th>C++ Types</th>
<th>OCCT Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Standard_Integer</td>
</tr>
<tr>
<td>double</td>
<td>Standard_Real</td>
</tr>
<tr>
<td>float</td>
<td>Standard_ShortReal</td>
</tr>
<tr>
<td>unsigned int</td>
<td>Standard_Boolean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>char</th>
<th>Standard_Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>Standard_ExtCharacter</td>
</tr>
<tr>
<td>char*</td>
<td>Standard_CString</td>
</tr>
<tr>
<td>void*</td>
<td>Standard_Address</td>
</tr>
<tr>
<td>short*</td>
<td>Standard_ExtString</td>
</tr>
</tbody>
</table>

- The types with asterisk are pointers.

**Reminder of the classes listed above:**

- **Standard_Integer**: fundamental type representing 32-bit integers yielding negative, positive or null values. *Integer* is implemented as a *typedef* of the C++ *int* fundamental type. As such, the algebraic operations +, -, *, / as well as the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on it.
- **Standard_Real**: fundamental type representing real numbers with finite precision and finite size. *Real* is implemented as a *typedef* of the C++ *double* (double precision) fundamental type. As such, the algebraic operations +, -, *, /, unary- and the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on reals.
- **Standard_ShortReal**: fundamental type representing real numbers with finite precision and finite size. *ShortReal* is implemented as a *typedef* of the C++ *float* (simple precision) fundamental type. As such, the algebraic operations +, -, *, /, unary- and the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on reals.
- **Standard_Boolean**: fundamental type representing logical expressions. It has two values: false and true. *Boolean* is implemented as a *typedef* of the C++ *unsigned int* fundamental type. As such, the algebraic operations and, or, xor and not as well as equivalence relations == and != are defined on Booleans.
- **Standard_Character**: fundamental type representing the normalized ASCII character set. It may be assigned the values of the 128 ASCII characters. *Character* is implemented as a *typedef* of the C++ *char* fundamental type. As such, the ordering and equivalence relations <, <=, ==, !=, >=, > are defined on characters using the order of the ASCII chart (ex: A B).
- **Standard_ExtCharacter**: fundamental type representing the Unicode character set. It is a 16-bit character type. *ExtCharacter* is implemented as a *typedef* of the C++ *short* fundamental type. As
such, the ordering and equivalence relations <, <=, ==, !=, >=, > are
defined on extended characters using the order of the UNICODE
cart (ex: A B).
- **Standard_CString** : fundamental type representing string literals. A
  string literal is a sequence of ASCII (8 bits) characters enclosed in
double quotes. CString is implemented as a **typedef** of the C++ **char**
fundamental type.
- **Standard_Address** : fundamental type representing a generic
  pointer. Address is implemented as a **typedef** of the C++ **void**
fundamental type.
- **Standard_ExtString** is a fundamental type representing string
  literals as sequences of Unicode (16 bits) characters. ExtString is
  implemented as a **typedef** of the C++ **short** fundamental type.

**Types manipulated by value**

There are three categories of types which are manipulated by value:

- Primitive types
- Enumerated types
- Types defined by classes not inheriting from **Standard_Transient**,
  whether directly or not. Types which are manipulated by value
  behave in a more direct fashion than those manipulated by handle
  and thus can be expected to perform operations faster, but they
  cannot be stored independently in a file.
Types manipulated by reference (handle)

These are types defined by classes inheriting from the *Transient* class.

![Diagram of Manipulated by handle]

**An object manipulated by reference can be compounded of primitives and handles to other objects. It may or may not be persistent.**

**Manipulation of a data type by reference**

**When is it necessary to use a handle?**

When you design an object, it can be difficult to choose how to manipulate that object: by value or by handle. The following ideas can help you to make up your mind:

- If your object may have a long lifetime within the application and you want to make multiple references to it, it would be preferable to manipulate this object with a handle. The memory for the object will be allocated on the heap. The handle which points to that memory is a light object which can be rapidly passed in argument. This avoids the penalty of copying a large object.
- If your object will have a limited lifetime, for example, used within a single algorithm, it would be preferable to manipulate this object by value, non-regarding its size, because this object is allocated on the stack and the allocation and de-allocation of memory is extremely rapid, which avoids the implicit calls to *new* and *delete* occasioned by allocation on the heap.
- Finally, if an object will be created only once during, but will exist throughout the lifetime of the application, the best choice may be a class manipulated by handle or a value declared as a global.
variable.
Programming with Handles

Handle Definition

A handle is OCCT implementation of a smart pointer. Several handles can reference the same object. Also, a single handle may reference several objects, but only one at a time. To have access to the object it refers to, the handle must be de-referenced just as with a C++ pointer.

Organization of Classes

Class \textit{Standard\_Transient} is a root of a big hierarchy of OCCT classes that are said to be operable by handles. It provides a reference counter field, inherited by all its descendant classes, that is used by associated \textit{Handle()} classes to track a number of handles pointing to this instance of the object.

Objects of classes derived (directly or indirectly) from \textit{Transient}, are normally allocated in dynamic memory using operator \texttt{new}, and manipulated by handle. Handle is defined as template class \texttt{opencascade::handle<>.} Open CASCADE Technology provides preprocessor macro \textit{Handle()} that is historically used throughout OCCT code to name a handle:

\begin{verbatim}
Handle(Geom_Line) aLine; // "Handle(Geom_Line)" is expanded to "opencascade::handleL<Geom_Line>"
\end{verbatim}

In addition, for standard OCCT classes additional \texttt{typedef} is defined for a handle, as the name of a class prefixed by \texttt{Handle\_}. For instance, the above example can be also coded as:

\begin{verbatim}
Handle_Geom_Line aLine; // "Handle_Geom_Line" is typedef to "opencascade::handleL<Geom_Line>"
\end{verbatim}

Using a Handle

A handle is characterized by the object it references.
Before performing any operation on a transient object, you must declare the handle. For example, if Point and Line are two transient classes from the Geom package, you would write:

```
Handle(Geom_Point) p1, p2;
```

Declaring a handle creates a null handle that does not refer to any object. The handle may be checked to be null by its method `IsNull()`. To nullify a handle, use method `Nullify()`.

To initialize a handle, either a new object should be created or the value of another handle can be assigned to it, on condition that their types are compatible.

**Note** that handles should only be used for object sharing. For all local operations, it is advisable to use classes manipulated by values.

### Type Management

Open CASCADE Technology provides a means to describe the hierarchy of data types in a generic way, with a possibility to check the exact type of the given object at run-time (similarly to C++ RTTI).

To enable this feature, a class declaration should include the declaration of OCCT RTTI. Header `Standard_Type.hxx` provides two variants of preprocessor macros facilitating this:

- **Inline variant**, which declares and defines RTTI methods by a single line of code:
  ```
  #include <Geom_Surface.hxx>
  class Appli_ExtSurface : public Geom_Surface
  {
    ...
    public:
      DEFINE_STANDARD_RTTIEXT(Appli_ExtSurface, Geom_Surface)
  };
  ```
- **Out-of line variant**, which uses one macro in the declaration
(normally in the header file), and another in the implementation (in C++ source):

In *Appli_ExtSurface.hxx* file:

```c++
#include <Geom_Surface.hxx>

class Appli_ExtSurface : public Geom_Surface
{
    ...

public:

    DEFINE_STANDARD_RTTIEXT(Appli_ExtSurface, Geom_Surface)
};
```

In *Appli_ExtSurface.cxx* file:

```c++
#include <Appli_ExtSurface.hxx>

IMPLEMENT_STANDARD_RTTIEXT(Appli_ExtSurface, Geom_Surface)
```

These macros define method *DynamicType()* that returns a type descriptor - handle to singleton instance of the class *Standard_Type* describing the class. The type descriptor stores the name of the class and the descriptor of its parent class.

Note that while inline version is easier to use, for widely used classes this method may lead to bloating of binary code of dependent libraries, due to multiple instantiations of inline method.

To get the type descriptor for a given class type, use macro *STANDARD_TYPE()* with the name of the class as argument.

Example of usage:

```c++
if (aCurve->IsKind(STANDARD_TYPE(Geom_Line))) // equivalent to "if (dynamic_cast<Geom_Line>(aCurve.get()) != 0)"
{
    ...
```
Type Conformity

The type used in the declaration of a handle is the static type of the object, the type seen by the compiler. A handle can reference an object instantiated from a subclass of its static type. Thus, the dynamic type of an object (also called the actual type of an object) can be a descendant of the type which appears in the handle declaration through which it is manipulated.

Consider the class `CartesianPoint`, a sub-class of `Point`; the rule of type conformity can be illustrated as follows:

```cpp
Handle (Geom_Point) p1;
Handle (Geom_CartesianPoint) p2;
p2 = new Geom_CartesianPoint;
p1 = p2; // OK, the types are compatible
```

The compiler sees `p1` as a handle to `Point` though the actual object referenced by `p1` is of the `CartesianPoint` type.

Explicit Type Conversion

According to the rule of type conformity, it is always possible to go up the class hierarchy through successive assignments of handles. On the other hand, assignment does not authorize you to go down the hierarchy. Consequently, an explicit type conversion of handles is required.

A handle can be converted explicitly into one of its sub-types if the actual type of the referenced object is a descendant of the object used to cast the handle. If this is not the case, the handle is nullified (explicit type conversion is sometimes called a “safe cast”). Consider the example below.

```cpp
Handle (Geom_Point) p1;
Handle (Geom_CartesianPoint) p2, p3;
p2 = new Geom_CartesianPoint;
p1 = p2; // OK, standard assignment
```
p3 = Handle (Geom_CartesianPoint)::DownCast (p1);
// OK, the actual type of p1 is CartesianPoint,
   although the static type of the handle is Point

If conversion is not compatible with the actual type of the referenced object, the handle which was “cast” becomes null (and no exception is raised). So, if you require reliable services defined in a sub-class of the type seen by the handle (static type), write as follows:

```cpp
void MyFunction (const Handle(A) & a)
{
    Handle(B) b = Handle(B)::DownCast(a);
    if (! b.IsNull()) {
        // we can use “b” if class B inherits from A
    } else {
        // the types are incompatible
    }
}
```

Downcasting is used particularly with collections of objects of different types; however, these objects should inherit from the same root class.

For example, with a sequence of transient objects `SequenceOfTransient` and two classes A and B that both inherit from `Standard_Transient`, you get the following syntax:

```cpp
Handle (A) a;
Handle (B) b;
Handle (Standard_Transient) t;
SequenceOfTransient s;
a = new A;
s.Append (a);
b = new B;
s.Append (b);
t = s.Value (1);
// here, you cannot write:
// a = t; // ERROR !
```


// so you downcast:
a = Handle (A)::Downcast (t)
if (! a.IsNull()) {
    // types are compatible, you can use a
}
else {
    // the types are incompatible
}

**Using Handles to Create Objects**

To create an object which is manipulated by handle, declare the handle and initialize it with the standard C++ **new** operator, immediately followed by a call to the constructor. The constructor can be any of those specified in the source of the class from which the object is instanced.

```cpp
Handle (Geom_CartesianPoint) p;
p = new Geom_CartesianPoint (0, 0, 0);
```

Unlike for a pointer, the **delete** operator does not work on a handle; the referenced object is automatically destroyed when no longer in use.

**Invoking Methods**

Once you have a handle to an object, you can use it like a pointer in C++. To invoke a method which acts on the referenced object, you translate this method by the standard **arrow** operator, or alternatively, by function call syntax when this is available.

To test or to modify the state of the handle, the method is translated by the **dot** operator. The example below illustrates how to access the coordinates of an (optionally initialized) point object:

```cpp
Handle (Geom_CartesianPoint) centre;
Standard_Real x, y, z;
if (centre.IsNull()) {
    centre = new PGeom_CartesianPoint (0, 0, 0);
}
centre->Coord(x, y, z);
```
The example below illustrates how to access the type object of a Cartesian point:

```cpp
Handle(Standard_Transient) p = new
    Geom_CartesianPoint(0.,0.,0.);
if ( p->DynamicType() ==
    STANDARD_TYPE(Geom_CartesianPoint) )
    cout << ;Type check OK; << endl;
else
    cout << ;Type check FAILED; << endl;
```

*NullObject* exception will be raised if a field or a method of an object is accessed via a *Null* handle.

**Invoking Class Methods**

A class method is called like a static C++ function, i.e. it is called by the name of the class of which it is a member, followed by the “::*” operator and the name of the method.

For example, we can find the maximum degree of a Bezier curve:

```cpp
Standard_Integer n;
n = Geom_BezierCurve::MaxDegree();
```

**Handle deallocation**

Before you delete an object, you must ensure it is no longer referenced. To reduce the programming load related to this management of object life, the delete function in Open CASCADE Technology is secured by a *reference counter* of classes manipulated by handle. A handle automatically deletes an object when it is no longer referenced. Normally you never call the delete operator explicitly on instances of subclasses of *Standard_Transient*.

When a new handle to the same object is created, the reference counter is incremented. When the handle is destroyed, nullified, or reassigned to another object, that counter is decremented. The object is automatically deleted by the handle when reference counter becomes 0.
The principle of allocation can be seen in the example below.

```cpp
...
{
    Handle (TColStd_HSequenceOfInteger) H1 = new TColStd_HSequenceOfInteger;
    // H1 has one reference and corresponds to 48 bytes of memory
    
    Handle (TColStd_HSequenceOfInteger) H2;
    H2 = H1; // H1 has two references
    if (argc == 3) {
        Handle (TColStd_HSequenceOfInteger) H3;
        H3 = H1;
        // Here, H1 has three references
        ...
    }
    // Here, H1 has two references
}
// Here, H1 has 1 reference
}
// Here, H1 has no reference and the referred TColStd_HSequenceOfInteger object is deleted.
```

You can easily cast a reference to the handle object to `void*` by defining the following:

```cpp
void *pointer;
Handle(Some_class) aHandle;
// Here only a pointer will be copied
Pointer = &aHandle;
// Here the Handle object will be copied
aHandle = * (Handle(Some_Class) *)pointer;
```

**Cycles**

Cycles appear if two or more objects reference each other by handles (stored as fields). In this condition automatic destruction will not work.
Consider for example a graph, whose objects (primitives) have to know the graph object to which they belong, i.e. a primitive must have a reference to complete graph object. If both primitives and the graph are manipulated by handle and they refer to each other by keeping a handle as a field, the cycle appears.

The graph object will not be deleted when the last handle to it is destructed in the application, since there are handles to it stored inside its own data structure (primitives).

There are two approaches how to avoid such situation:

- Use C++ pointer for one kind of references, e.g. from a primitive to the graph
- Nullify one set of handles (e.g. handles to a graph in primitives) when a graph object needs to be destroyed
Memory Management

In a work session, geometric modeling applications create and delete a considerable number of C++ objects allocated in the dynamic memory (heap). In this context, performance of standard functions for allocating and deallocating memory may be not sufficient. For this reason, Open CASCADE Technology employs a specialized memory manager implemented in the Standard package.

The Memory Manager is based on the following principles:

- small memory arrays are grouped into clusters and then recycled (clusters are never released to the system),
- large arrays are allocated and de-allocated through the standard functions of the system (the arrays are released to system when they are no longer used).

As a general rule, it is advisable to allocate memory through significant blocks. In this way, the user can work with blocks of contiguous data and it facilitates memory page manager processing.

Usage of Memory Manager

To allocate memory in a C code with Open CASCADE Technology memory manager, simply use method Standard::Allocate() instead of malloc() and method Standard::Free() instead of free(). In addition, method Standard::Reallocate() is provided to replace C function realloc().

In C++, operators new() and delete() for a class may be defined so as to allocate memory using Standard::Allocate() and free it using Standard::Free(). In that case all objects of that class and all inherited classes will be allocated using the OCCT memory manager.

Preprocessor macro DEFINE_STANDARD_ALLOC provided by header Standard_DefineAlloc.hxx defines new() and delete() in this way. It is used for all OCCT classes (apart from a few exceptions) which thus are allocated using the OCCT memory manager. Since operators new() and delete() are inherited, this is also true for any class derived from an OCCT class, for instance, for all classes derived from
Note that it is possible (though not recommended unless really unavoidable) to redefine new() and delete() functions for a class inheriting Standard_Transient. If that is done, the method Delete() should be also redefined to apply operator delete to this pointer. This will ensure that appropriate delete() function will be called, even if the object is manipulated by a handle to a base class.

How to configure the Memory Manager

The OCCT memory manager may be configured to apply different optimization techniques to different memory blocks (depending on their size), or even to avoid any optimization and use C functions malloc() and free() directly. The configuration is defined by numeric values of the following environment variables:

- **MMGT_OPT**: if set to 0 (default) every memory block is allocated in C memory heap directly (via malloc() and free() functions). In this case, all other options except for MMGT_CLEAR are ignored; if set to 1 the memory manager performs optimizations as described below; if set to 2, Intel ® TBB optimized memory manager is used.
- **MMGT_CLEAR**: if set to 1 (default), every allocated memory block is cleared by zeros; if set to 0, memory block is returned as it is.
- **MMGT CELLSIZE**: defines the maximal size of blocks allocated in large pools of memory. Default is 200.
- **MMGT_NBPAGES**: defines the size of memory chunks allocated for small blocks in pages (operating-system dependent). Default is 1000.
- **MMGT_THRESHOLD**: defines the maximal size of blocks that are recycled internally instead of being returned to the heap. Default is 40000.
- **MMGT_MMAP**: when set to 1 (default), large memory blocks are allocated using memory mapping functions of the operating system; if set to 0, they will be allocated in the C heap by malloc().

Optimization Techniques

When MMGT_OPT is set to 1, the following optimization techniques are used:
• Small blocks with a size less than MMGT_CELLSIZE, are not allocated separately. Instead, a large pools of memory are allocated (the size of each pool is MMGT_NBPAGES pages). Every new memory block is arranged in a spare place of the current pool. When the current memory pool is completely occupied, the next one is allocated, and so on.

In the current version memory pools are never returned to the system (until the process finishes). However, memory blocks that are released by the method Standard::Free() are remembered in the free lists and later reused when the next block of the same size is allocated (recycling).

• Medium-sized blocks, with a size greater than MMGT_CELLSIZE but less than MMGT_THRESHOLD, are allocated directly in the C heap (using malloc() and free()). When such blocks are released by the method Standard::Free() they are recycled just like small blocks.

However, unlike small blocks, the recycled medium blocks contained in the free lists (i.e. released by the program but held by the memory manager) can be returned to the heap by method Standard::Purge().

• Large blocks with a size greater than MMGT_THRESHOLD, including memory pools used for small blocks, are allocated depending on the value of MMGT_MMAP: if it is 0, these blocks are allocated in the C heap; otherwise they are allocated using operating-system specific functions managing memory mapped files. Large blocks are returned to the system immediately when Standard::Free() is called.

Benefits and drawbacks

The major benefit of the OCCT memory manager is explained by its recycling of small and medium blocks that makes an application work much faster when it constantly allocates and frees multiple memory blocks of similar sizes. In practical situations, the real gain on the application performance may be up to 50%.

The associated drawback is that recycled memory is not returned to the operating system during program execution. This may lead to considerable memory consumption and even be misinterpreted as a memory leak. To minimize this effect it is necessary to call the method
Standard::Purge after the completion of memory-intensive operations.

The overhead expenses induced by the OCCT memory manager are:

- size of every allocated memory block is rounded up to 8 bytes (when MMGT_OPT is 0 (default), the rounding is defined by the CRT; the typical value for 32-bit platforms is 4 bytes)
- additional 4 bytes (or 8 on 64-bit platforms) are allocated in the beginning of every memory block to hold its size (or address of the next free memory block when recycled in free list) only when MMGT_OPT is 1.

Note that these overheads may be greater or less than overheads induced by the C heap memory manager, so overall memory consumption may be greater in either optimized or standard modes, depending on circumstances.

As a general rule, it is advisable to allocate memory through significant blocks. In this way, you can work with blocks of contiguous data, and processing is facilitated for the memory page manager.

OCCT memory manager uses mutex to lock access to free lists, therefore it may have less performance than non-optimized mode in situations when different threads often make simultaneous calls to the memory manager. The reason is that modern implementations of malloc() and free() employ several allocation arenas and thus avoid delays waiting mutex release, which are possible in such situations.
Exceptions

Introduction

The behavior of any object is implemented by the methods, which were defined in its class declaration. The definition of these methods includes not only their signature (their programming interface) but also their domain of validity.

This domain is expressed by **exceptions**. Exceptions are raised under various error conditions to protect software quality.

Exception handling provides a means of transferring control from a given point in a program being executed to an **exception handler** associated with another point previously executed.

A method may raise an exception which interrupts its normal execution and transfers control to the handler catching this exception.

A hierarchy of commonly used exception classes is provided. The root class is `Standard_Failure` from the `Standard` package. So each exception inherits from `Standard_Failure` either directly or by inheriting from another exception. Exception classes list all exceptions, which can be raised by any OCCT function.

Open CASCADE Technology also provides support for converting system signals (such as access violation or division by zero) to exceptions, so that such situations can be safely handled with the same uniform approach.

However, in order to support this functionality on various platforms, some special methods and workarounds are used. Though the implementation details are hidden and handling of OCCT exceptions is done basically in the same way as with C++, some peculiarities of this approach shall be taken into account and some rules must be respected.

The following paragraphs describe recommended approaches for using exceptions when working with Open CASCADE Technology.
Raising an Exception

“C++ like” Syntax

To raise an exception of a definite type method `Raise()` of the appropriate exception class shall be used.

```cpp
DomainError::Raise("Cannot cope with this condition");
```

raises an exception of `DomainError` type with the associated message “Cannot cope with this condition”, the message being optional. This exception may be caught by a handler of a `DomainError` type as follows:

```cpp
try {
    OCC_CATCH_SIGNALS
    // try block
} catch(DomainError) {
    // handle DomainError exceptions here
}
```

Regular usage

Exceptions should not be used as a programming technique, to replace a “goto” statement for example, but as a way to protect methods against misuse. The caller must make sure its condition is such that the method can cope with it.

Thus,

- No exception should be raised during normal execution of an application.
- A method which may raise an exception should be protected by other methods allowing the caller to check on the validity of the call.

For example, if you consider the `TCollection_Array1` class used with:

- `Value` function to extract an element
• Lower function to extract the lower bound of the array
• Upper function to extract the upper bound of the array.

then, the Value function may be implemented as follows:

```cpp
Item TCollection_Array1::Value (const Standard_Integer&index) const
{
    // where r1 and r2 are the lower and upper bounds of the array
    if(index < r1 || index > r2) {
       OutOfRange::Raise("Index out of range in Array1::Value");
    }
    return contents[index];
}
```

Here validity of the index is first verified using the Lower and Upper functions in order to protect the call. Normally the caller ensures the index being in the valid range before calling Value(). In this case the above implementation of Value is not optimal since the test done in Value is time-consuming and redundant.

It is a widely used practice to include that kind of protections in a debug build of the program and exclude in release (optimized) build. To support this practice, the macros Raise_if() are provided for every OCCT exception class:

```
<ErrorTypeName>_Raise_if(condition, “Error message”);
```

where ErrorTypeName is the exception type, condition is the logical expression leading to the raise of the exception, and Error message is the associated message.

The entire call may be removed by defining one of the preprocessor symbols No_Exception or No_<ErrorTypeName> at compile-time:

```
#define No_Exception /* remove all raises */
```
Using this syntax, the `Value` function becomes:

```c++
Item  TCollection_Array1::Value (const Standard_Integer&index) const
{
    OutOfRange_Raise_if(index < r1 || index > r2, 
                        "index out of range in Array1::Value");
    return contents[index];
}
```

**Handling an Exception**

When an exception is raised, control is transferred to the nearest handler of a given type in the call stack, that is:

- the handler whose try block was most recently entered and not yet exited,
- the handler whose type matches the raise expression.

A handler of `T` exception type is a match for a raise expression with an exception type of `E` if:

- `T` and `E` are of the same type, or
- `T` is a supertype of `E`.

In order to handle system signals as exceptions, make sure to insert macro `OCC_CATCH_SIGNALS` somewhere in the beginning of the relevant code. The recommended location for it is first statement after opening brace of `try {}` block.

As an example, consider the exceptions of type `NumericError`, `Overflow`, `Underflow` and `ZeroDivide`, where `NumericError` is the parent type of the three others.

```c++
void f(1)
{
    try {
        OCC_CATCH_SIGNALS
```
Here, the first handler will catch exceptions of *Overflow* type and the second one – exceptions of *NumericError* type and all exceptions derived from it, including *Underflow* and *ZeroDivide*.

The handlers are checked in order of appearance, from the nearest to the try block to the most distant from it, until one matches the raise expression. For a try block, it would be a mistake to place a handler for a base exception type ahead of a handler for its derived type since that would ensure that the handler for the derived exception would never be invoked.
The exceptions form a hierarchy tree completely separated from other user defined classes. One exception of type *Failure* is the root of the entire exception hierarchy. Thus, using a handler with *Failure* type catches any OCCT exception. It is recommended to set up such a handler in the main routine.

The main routine of a program would look like this:

```cpp
#include <Standard_ErrorHandler.hxx>
#include <Standard_Failure.hxx>
#include <iostream.h>
int main (int argc, char* argv[]) {
    try {
        OCC_CATCH_SIGNALS
        // main block
        return 0;
    }
    catch(Standard_Failure) {
        Handle(Standard_Failure) error = Standard_Failure::Caught ();
        cout error end1;
    }
    return 1;
}
```

In this example function *Caught* is a static member of *Failure* that returns an exception object containing the error message built in the raise expression. Note that this method of accessing a raised object is used in Open CASCADE Technology instead of usual C++ syntax (receiving the exception in catch argument).

Though standard C++ scoping rules and syntax apply to try block and handlers, note that on some platforms Open CASCADE Technology may be compiled in compatibility mode when exceptions are emulated by long jumps (see below). In this mode it is required that no statement precedes or follows any handler. Thus it is highly recommended to always include a
try block into additional {} braces. Also this mode requires that header file
*Standard_ErrorHandler.hxx* be included in your program before a try
block, otherwise it may fail to handle Open CASCADE Technology
exceptions; furthermore *catch()* statement does not allow passing
exception object as argument.

**Catching signals**

In order for the application to be able to catch system signals (access
violation, division by zero, etc.) in the same way as other exceptions, the
appropriate signal handler shall be installed in the runtime by the method
*OSD::SetSignal()*.

Normally this method is called in the beginning of the main() function. It
installs a handler that will convert system signals into OCCT exceptions.

In order to actually convert signals to exceptions, macro
*OCC_CATCH_SIGNALS* needs to be inserted in the source code. The
typical place where this macro is put is beginning of the *try{}* block which
catches such exceptions.

**Implementation on various platforms.**

The exception handling mechanism in Open CASCADE Technology is
implemented in different ways depending on the preprocessor macro
*OCC_CONVERT_SIGNALS*, which shall be consistently defined by
compilation procedures for both Open CASCADE Technology and user
applications:

1. On Windows, these macros are not defined by default, and normal
   C++ exceptions are used in all cases, including throwing from signal
   handler. Thus the behavior is as expected in C++.
2. On Linux, macro *OCC_CONVERT_SIGNALS* is defined by default.
   The C++ exception mechanism is used for catching exceptions and
   for throwing them from normal code. Since it is not possible to throw
   C++ exception from system signal handler function, that function
   makes a long jump to the nearest (in the execution stack) invocation
   of macro *OCC_CATCH_SIGNALS*, and only there the C++ exception
   gets actually thrown. The macro *OCC_CATCH_SIGNALS* is defined
   in the file *Standard_ErrorHandler.hxx*. Therefore, including this file is
necessary for successful compilation of a code containing this macro.

This mode differs from standard C++ exception handling only for signals:

- macro `OCC_CATCH_SIGNALS` is necessary (besides call to `OSD::SetSignal()` described above) for conversion of signals into exceptions;
- the destructors for automatic C++ objects created in the code after that macro and till the place where signal is raised will not be called in case of signal, since no C++ stack unwinding is performed by long jump.

In general, for writing platform-independent code it is recommended to insert macros `OCC_CATCH_SIGNALS` in try {} blocks or other code where signals may happen.
Plug-In Management

Distribution by Plug-Ins

A plug-in is a component that can be loaded dynamically into a client application, not requiring to be directly linked to it. The plug-in is not bound to its client, i.e. the plug-in knows only how its connection mechanism is defined and how to call the corresponding services.

A plug-in can be used to:

- implement the mechanism of a driver, i.e dynamically changing a driver implementation according to the current transactions (for example, retrieving a document stored in another version of an application),
- restrict processing resources to the minimum required (for example, it does not load any application services at run-time as long as the user does not need them),
- facilitate modular development (an application can be delivered with base functions while some advanced capabilities will be added as plug-ins when they are available).

The plug-in is identified with the help of the global universal identifier (GUID). The GUID includes lower case characters and cannot end with a blank space.

Once it has been loaded, the call to the services provided by the plug-in is direct (the client is implemented in the same language as the plug-in).

C++ Plug-In Implementation

The C++ plug-in implements a service as an object with functions defined in an abstract class (this abstract class and its parent classes with the GUID are the only information about the plug-in implemented in the client application). The plug-in consists of a sharable library including a method named Factory which creates the C++ object (the client cannot instantiate this object because the plug-in implementation is not visible). Foundation classes provide in the package Plugin a method named
Load(), which enables the client to access the required service through a library.

That method reads the information regarding available plug-ins and their locations from the resource file Plugin found by environment variable CSF_PluginDefaults:

\$CSF_PluginDefaults/Plugin

The Load method looks for the library name in the resource file or registry through its GUID, for example, on UNIX:

! METADATADRIVER whose value must be OS or DM.

! FW
a148e300-5740-11d1-a904-080036aaa103.Location:
   libFWOSPlugin.so

Then the Load method loads the library according to the rules of the operating system of the host machine (for example, by using environment variables such as LD_LIBRARY_PATH with Unix and PATH with Windows). After that it invokes the PLUGINFACTORY method to return the object, which supports the required service. The client may then call the functions supported by this object.

**C++ Client Plug-In Implementation**

To invoke one of the services provided by the plug-in, you may call the Plugin::Load() global function with the Standard_GUID of the requested service as follows:

```
Handle(FADriver_PartStorer)::DownCast(PlugIn::Load (yourStandardGUID));
```

Let us take FAFactory.hxx and FAFactory.cxx as an example:

```
#include <Standard_Macro.hxx>
#include <Standard_GUID.hxx>
#include <Standard_Transient.hxx>
```
class FAFactory
{
public:
    Standard_EXPORT static Handle(Standard_Transient) Factory (const Standard_GUID& theGUID);
};

#include <FAFactory.hxx>
#include <FADriver_PartRetriever.hxx>
#include <FADriver_PartStorer.hxx>
#include <FirstAppSchema.hxx>
#include <Standard_Failure.hxx>
#include <FACDM_Application.hxx>
#include <Plugin_Macro.hxx>

static Standard_GUID StorageDriver ("45b3c690-22f3-11d2-b09e-0000f8791463");
static Standard_GUID RetrievalDriver("45b3c69c-22f3-11d2-b09e-0000f8791463");
static Standard_GUID Schema ("45b3c6a2-22f3-11d2-b09e-0000f8791463");

Handle(Standard_Transient) FAFactory::Factory (const Standard_GUID& theGUID)
{
    if (theGUID == StorageDriver)
    {
        std::cout << "FAFactory : Create store driver\n";
        static Handle(FADriver_PartStorer) sd = new
FADriver_PartStorer();
return sd;
}

if (theGUID == RetrievalDriver)
{
    std::cout << "FAFactory : Create retrieve
driver\n";
    static Handle(FADriver_PartRetriever) rd = new
    FADriver_PartRetriever();
    return rd;
}

if (theGUID == Schema)
{
    std::cout << "FAFactory : Create schema\n";
    static Handle(FirstAppSchema) s = new
    FirstAppSchema();
    return s;
}

Standard_Failure::Raise("FAFactory: unknown
GUID");
return Handle(Standard_Transient());
}

// export plugin function "PLUGINFAC TORY"
PLUGIN(FAFactory)

Application might also instantiate a factory by linking to the library and
calling FAFactory::Factory() directly.
Collections, Strings, Quantities and Unit Conversion

Collections

Overview

The Collections component contains the classes that handle dynamically sized aggregates of data. They include a wide range of collections such as arrays, lists and maps.

Collections classes are generic (C++ template-like), that is, they define a structure and algorithms allowing to hold a variety of objects which do not necessarily inherit from a unique root class (similarly to C++ templates).

When you need to use a collection of a given type of object you must instantiate it for this specific type of element. Once this declaration is compiled, all the functions available on the generic collection are available on your instantiated class.

However, note that:

- Each collection directly used as an argument in OCCT public syntax is instantiated in an OCCT component.
- The TColStd package (Collections of Standard Objects component) provides numerous instantiations of these generic collections with objects from the Standard package or from the Strings component. The Collections component provides a wide range of generic collections:
  - Arrays are generally used for a quick access to the item, however an array is a fixed sized aggregate.
  - Sequences are variable-sized structures, they avoid the use of large and quasi-empty arrays. A sequence item is longer to access than an array item: only an exploration in sequence is effective (but sequences are not adapted for numerous explorations). Arrays and sequences are commonly used as data structures for more complex objects.
Maps are dynamic structures, where the size is constantly adapted to the number of inserted items and access to an item is the fastest. Maps structures are commonly used in cases of numerous explorations: they are typically internal data structures for complex algorithms.

Lists are similar to sequences but have different algorithms to explore them.

Specific iterators for sequences and maps.

**Generic general-purpose Aggregates**

**TCollection_Array1**

These are unidimensional arrays similar to C arrays, i.e. of fixed size but dynamically dimensioned at construction time. As with a C array, the access time for an Array1 indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects.

Array1 is a generic class which depends on Item, the type of element in the array.

Array1 indexes start and end at a user-defined position. Thus, when accessing an item, you must base the index on the lower and upper bounds of the array.

**TCollection_Array2**

These are bi-dimensional arrays of fixed size but dynamically dimensioned at construction time.

As with a C array, the access time for an Array2 indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects.

Array2 is a generic class which depends on Item, the type of element in the array.

Array2 indexes start and end at a user-defined position. Thus, when accessing an item, you must base the index on the lower and upper
bounds of the array.

**TCollection_HArray1**

These are unidimensional arrays similar to C arrays, i.e. of fixed size but dynamically dimensioned at construction time. As with a C array, the access time for an HArray1 or HArray2 indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects.

HArray1 objects are **handles** to arrays.

- HArray1 arrays may be shared by several objects.
- You may use a TCollection_Array1 structure to have the actual array.

HArray1 is a generic class which depends on two parameters:

- **Item**, the type of element in the array,
- **Array**, the actual type of array handled by HArray1. This is an instantiation with Item of the TCollection_Array1 generic class.

HArray1 indexes start and end at a user-defined position. Thus, when accessing an item, you must base the index on the lower and upper bounds of the array.

**TCollection_HArray2**

These are bi-dimensional arrays of fixed size but dynamically dimensioned at construction time.

As with a C array, the access time for an HArray2 indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects.

HArray2 objects are **handles** to arrays.

- HArray2 arrays may be shared by several objects.
- You may use a TCollection_Array2 structure to have the actual array.

HArray2 is a generic class which depends on two parameters:
- **Item**, the type of element in the array,
- **Array**, the actual type of array handled by `HArray2`. This is an instantiation with `Item` of the `TCollection_Array2` generic class.

**TCollection_HSequence**

This is a sequence of items indexed by an integer.

Sequences have about the same goal as unidimensional arrays `TCollection_HArray1`: they are commonly used as elementary data structures for more complex objects. But a sequence is a structure of *variable size*: sequences avoid the use of large and quasi-empty arrays. Exploring a sequence data structure is effective when the exploration is done in sequence; elsewhere a sequence item is longer to read than an array item. Note also that sequences are not effective when they have to support numerous algorithmic explorations: a map is better for that.

*HSequence* objects are **handles** to sequences.

- *HSequence* sequences may be shared by several objects.
- You may use a `TCollection_Sequence` structure to have the actual sequence.

*HSequence* is a generic class which depends on two parameters:

- **Item**, the type of element in the sequence,
- **Seq**, the actual type of sequence handled by `HSequence`. This is an instantiation with `Item` of the `TCollection_Sequence` generic class.

**TCollection_List**

These are ordered lists of non-unique objects which can be accessed sequentially using an iterator. Item insertion in a list is very fast at any position. But searching for items by value may be slow if the list is long, because it requires a sequential search.

*List* is a generic class, which depends on `Item`, the type of element in the structure. Use a `ListIterator` iterator to explore a `List` structure.

An iterator class is automatically instantiated from the
TCollection_ListIterator class at the time of instantiation of a List structure.

A sequence is a better structure when searching for items by value.

Queues and stacks are other kinds of list with a different access to data.

TCollection_Sequence

This is a sequence of items indexed by an integer. Sequences have about the same goal as unidimensional arrays (TCollection_Array1): they are commonly used as elementary data structures for more complex objects. But a sequence is a structure of variable size: sequences avoid the use of large and quasi-empty arrays. Exploring a sequence data structure is effective when the exploration is done in sequence; elsewhere a sequence item is longer to read than an array item. Note also that sequences are not effective when they have to support numerous algorithmic explorations: a map is better for that.

Sequence is a generic class which depends on Item, the type of element in the sequence.

Generic Maps

Maps are dynamically extended data structures where data is quickly accessed with a key. TCollection_BasicMap is a root class for maps.

General properties of maps

Map items may contain complex non-unitary data, thus it can be difficult to manage them with an array. The map allows a data structure to be indexed by complex data.

The size of a map is dynamically extended. So a map may be first dimensioned for a little number of items. Maps avoid the use of large and quasi-empty arrays.

The access time for a map item is much better than the one for a sequence, list, queue or stack item. It is comparable with the access time for an array item. It depends on the size of the map and on the quality of
the user redefinable function (the *hashing function*) to find quickly where is the item.

The performance of a map exploration may be better of an array exploration because the size of the map is adapted to the number of inserted items.

That is why maps are commonly used as internal data structures for algorithms.

**Definitions**

A map is a data structure for which data are addressed by *keys*.

Once inserted in the map, a map item is referenced as an *entry* of the map.

Each entry of the map is addressed by a key. Two different keys address two different entries of the map. The position of an entry in the map is called a *bucket*.

A map is dimensioned by its number of buckets, i.e. the maximum number of entries in the map. The performance of a map is conditioned by the number of buckets.

The *hashing function* transforms a key into a bucket index. The number of values that can be computed by the hashing function is equal to the number of buckets of the map.

Both the hashing function and the equality test between two keys are provided by a *hasher* object.

A map may be explored by a *map iterator*. This exploration provides only inserted entries in the map (i.e. non empty buckets).

**Collections of generic maps**

The *Collections* component provides numerous generic derived maps.

These maps include automatic management of the number of *buckets*: 
they are automatically resized when the number of keys exceeds the number of buckets. If you have a fair idea of the number of items in your map, you can save on automatic resizing by specifying a number of buckets at the time of construction, or by using a resizing function. This may be considered for crucial optimization issues.

**Keys, items and hashers** are parameters of these generic derived maps.

TCollection_MapHasher class describes the functions required by any hasher, which is to be used with a map instantiated from the **Collections** component.

An iterator class is automatically instantiated at the time of instantiation of a map provided by the **Collections** component if this map is to be explored with an iterator. Note that some provided generic maps are not to be explored with an iterator but with indexes (**indexed maps**).

TCollection_DataMap

This is a map used to store keys with associated items. An entry of **DataMap** is composed of both the key and the item. The **DataMap** can be seen as an extended array where the keys are the indexes.

**DataMap** is a generic class which depends on three parameters:

- **Key** is the type of key for an entry in the map,
- **Item** is the type of element associated with a key in the map,
- **Hasher** is the type of hasher on keys.

Use a **DataMapIterator** iterator to explore a **DataMap** map.

An iterator class is automatically instantiated from the TCollection_DataMapIterator generic class at the time of instantiation of a **DataMap** map.

TCollection_MapHasher class describes the functions required for a **Hasher** object.

TCollection_DoubleMap

This is a map used to bind pairs of keys (Key1,Key2) and retrieve them in
linear time.

Key1 is referenced as the first key of the DoubleMap and Key2 as the second key.

An entry of a DoubleMap is composed of a pair of two keys: the first key and the second key.

DoubleMap is a generic class which depends on four parameters:

- Key1 is the type of the first key for an entry in the map,
- Key2 is the type of the second key for an entry in the map,
- Hasher1 is the type of hasher on first keys,
- Hasher2 is the type of hasher on second keys.

Use DoubleMapIterator to explore a DoubleMap map.

An iterator class is automatically instantiated from the TCollection_DoubleMapIterator class at the time of instantiation of a DoubleMap map.

TCollection_MapHasher class describes the functions required for a Hasher1 or a Hasher2 object.

TCollection_IndexedDataMap

This is map to store keys with associated items and to bind an index to them.

Each new key stored in the map is assigned an index. Indexes are incremented as keys (and items) stored in the map. A key can be found by the index, and an index can be found by the key. No key but the last can be removed, so the indexes are in the range 1...Upper, where Upper is the number of keys stored in the map. An item is stored with each key.

An entry of an IndexedDataMap is composed of both the key, the item and the index. An IndexedDataMap is an ordered map, which allows a linear iteration on its contents. It combines the interest:

- of an array because data may be accessed with an index,
- and of a map because data may also be accessed with a key.
IndexedDataMap is a generic class which depends on three parameters:

- *Key* is the type of key for an entry in the map,
- *Item* is the type of element associated with a key in the map,
- *Hasher* is the type of hasher on keys.

TCollection_IndexedMap

This is map used to store keys and to bind an index to them.

Each new key stored in the map is assigned an index. Indexes are incremented as keys stored in the map. A key can be found by the index, and an index by the key. No key but the last can be removed, so the indexes are in the range 1...Upper where Upper is the number of keys stored in the map.

An entry of an IndexedMap is composed of both the key and the index. An IndexedMap is an ordered map, which allows a linear iteration on its contents. But no data is attached to the key. An IndexedMap is typically used by an algorithm to know if some action is still performed on components of a complex data structure.

IndexedMap is a generic class which depends on two parameters:

- *Key* is the type of key for an entry in the map,
- *Hasher* is the type of hasher on keys.

TCollection_Map

This is a basic hashed map, used to store and retrieve keys in linear time.

An entry of a Map is composed of the key only. No data is attached to the key. A Map is typically used by an algorithm to know if some action is still performed on components of a complex data structure.

Map is a generic class which depends on two parameters:

- *Key* is the type of key in the map,
- *Hasher* is the type of hasher on keys.
Use a *MapIterator* iterator to explore a *Map* map.

**TCollection**\_MapHasher

This is a hasher on the *keys* of a map instantiated from the *Collections* component.

A hasher provides two functions:

- *HashCode()* function transforms a key into a bucket index in the map. The number of values that can be computed by the hashing function is equal to the number of buckets in the map.
- *isEqual* is the equality test between two keys. Hashers are used as parameters in generic maps provided by the *Collections* component.

*MapHasher* is a generic class which depends on the type of keys, providing that *Key* is a type from the *Standard* package. In such cases *MapHasher* may be directly instantiated with *Key*. Note that the package *TColStd* provides some of these instantiations.

Elsewhere, if *Key* is not a type from the *Standard* package you must consider *MapHasher* as a template and build a class which includes its functions, in order to use it as a hasher in a map instantiated from the *Collections* component.

Note that *TCollection**\_AsciiString* and *TCollection**\_ExtendedString* classes correspond to these specifications, in consequence they may be used as hashers: when *Key* is one of these two types you may just define the hasher as the same type at the time of instantiation of your map.

**Iterators**

**TCollection**\_BasicMapIterator

This is a root class for map iterators. A map iterator provides a step by step exploration of all the entries of a map.

**TCollection**\_DataMapIterator
These are functions used for iterating the contents of a \textit{DataMap} map.

A map is a non-ordered data structure. The order in which entries of a map are explored by the iterator depends on its contents and change when the map is edited. It is not recommended to modify the contents of a map during the iteration: the result is unpredictable.

\textbf{TCollection\_DoubleMapIterator}

These are functions used for iterating the contents of a \textit{DoubleMap} map.

\textbf{TCollection\_ListIterator}

These are functions used for iterating the contents of a \textit{List} data structure.

A \textit{ListIterator} object can be used to go through a list sequentially, and as a bookmark to hold a position in a list. It is not an index, however. Each step of the iteration gives the current position of the iterator, to which corresponds the current item in the list. The current position is not defined if the list is empty, or when the exploration is finished.

An iterator class is automatically instantiated from this generic class at the time of instantiation of a \textit{List} data structure.

\textbf{TCollection\_MapIterator}

These are functions used for iterating the contents of a \textit{Map} map. An iterator class is automatically instantiated from this generic class at the time of instantiation of a \textit{Map} map.

\textbf{TCollection\_SetIterator}

These are functions used for iterating the contents of a \textit{Set} data structure. An iterator class is automatically instantiated from this generic class at the time of instantiation of a \textit{Set} structure.

\textbf{TCollection\_StackIterator}

These are functions used for iterating the contents of a \textit{Stack} data
structure.

An iterator class is automatically instantiated from this generic class at the time of instantiation of a Stack structure.
Collections of Standard Objects

Overview

While generic classes of the TCollection package are the root classes that describe the generic purpose of every type of collection, classes effectively used are extracted from the TColStd package. The TColStd and TShort packages provide frequently used instantiations of generic classes with objects from the Standard package or strings from the TCollection package.

Description

These instantiations are the following:

- Unidimensional arrays: instantiations of the TCollection_Array1 generic class with Standard Objects and TCollection strings.
- Bidimensional arrays: instantiations of the TCollection_Array2 generic class with Standard Objects.
- Unidimensional arrays manipulated by handles: instantiations of the TCollection_HArray1 generic class with Standard Objects and TCollection strings.
- Bidimensional arrays manipulated by handles: instantiations of the TCollection_HArray2 generic class with Standard Objects.
- Sequences: instantiations of the TCollection_Sequence generic class with Standard objects and TCollection strings.
- Sequences manipulated by handles: instantiations of the TCollection_HSequence generic class with Standard objects and TCollection strings.
- Lists: instantiations of the TCollection_List generic class with Standard objects.
- Queues: instantiations of the TCollection_Queue generic class with Standard objects.
- Sets: instantiations of the TCollection_Set generic class with Standard objects.
- Sets manipulated by handles: instantiations of the TCollection_HSet generic class with Standard objects.
- Stacks: instantiations of the TCollection_Stack generic class with Standard objects.
Standard objects.

- Hashers on map keys: instantiations of the `TCollection_MapHasher` generic class with Standard objects.
- Basic hashed maps: instantiations of the `TCollection_Map` generic class with Standard objects.
- Hashed maps with an additional item: instantiations of the `TCollection_DataMap` generic class with Standard objects.
- Basic indexed maps: instantiations of the `TCollection_IndexedMap` generic class with Standard objects.
- Indexed maps with an additional item: instantiations of the `TCollection_IndexedDataMap` generic class with Standard_Transient objects.
- Class `TColStd_PackedMapOfInteger` provides alternative implementation of map of integer numbers, optimized for both performance and memory usage (it uses bit flags to encode integers, which results in spending only 24 bytes per 32 integers stored in optimal case). This class also provides Boolean operations with maps as sets of integers (union, intersection, subtraction, difference, checks for equality and containment).
NCollections

Overview

The *NCollection* package provides a set of template collection classes used throughout OCCT.

Macro definitions of these classes are stored in *NCollection_Define*.hxx files. These definitions are now obsolete though still can be used, particularly for compatibility with the existing code.

Instantiation of collection classes

Now we are going to implement the definitions from *NCollection* in the code, taking as an example a sequence of points (analogue of *TColgp_SequenceOfPnt*).

Definition of a new collection class

Let the header file be *MyPackage_SequenceOfPnt.hxx*:

Template class instantiation

```cpp
#include <NCollection_Sequence.hxx>
#include <gp_Pnt.hxx>
typedef NCollection_Sequence<gp_Pnt> MyPackage_SequenceOfPnt;
```

Macro instantiation

```cpp
#include <NCollection_DefineSequence.hxx>
#include <gp_Pnt.hxx>

The following line defines the class "base collection of points"

```
DEFINE_BASECOLLECTION(MyPackage_BaseCollPnt, gp_Pnt)
```

The following line defines the class *MyPackage_SequenceOfPnt*
DEFINE_SEQUENCE (MyPackage_SequenceOfPnt, 
    MyPackage_BaseCollPnt, gp_Pnt)

**Definition of a new collection class managed by Handle**

It is necessary to provide relevant statements both in the header (.hxx file) and the C++ source (.cxx file).

Header file MyPackage_HSequenceOfPnt.hxx:

```cpp
#include <NCollection_DefineHSequence.hxx>
#include <gp_Pnt.hxx>
```

The following line defines the class "base collection of points"

```cpp
DEFINE_BASECOLLECTION(MyPackage_BaseCollPnt, gp_Pnt)
```

The following line defines the class *MyPackage_SequenceOfPnt*

```cpp
DEFINE_SEQUENCE (MyPackage_SequenceOfPnt, 
    MyPackage_BaseCollPnt, gp_Pnt)
```

The following line defines the classes *MyPackage_HSequenceOfPnt* and *Handle(MyPackage_HSequenceOfPnt)*

```cpp
DEFINE_HSEQUENCE (MyPackage_HSequenceOfPnt, 
    MyPackage_SequenceOfPnt)
```

Source code file will be *MyPackage_HSequenceOfPnt.cxx* or any other .cxx file (once in the whole project):

```cpp
IMPLEMENT_HSEQUENCE (MyPackage_HSequenceOfPnt)
```

**Arrays and sequences**

Standard collections provided by OCCT are:

- *NCollection_Array1* – fixed-size (at initialization) one-dimensional array; note that the index can start at any value, usually 1;
- *NCollection_Array2* – fixed-size (at initialization) two-dimensional array; note that the index can start at any value, usually 1;
- *NCollection_List* – plain list;
- *NCollection_Sequence* – double-connected list with access by index; note that the index starts at 1.

These classes provide STL-style iterators (methods begin() and end()) and thus can be used in STL algorithms.

**Maps**

NCollection provides several classes for storage of objects by value, providing fast search due to use of hash:

- *NCollection_Map* – hash set;
- *NCollection_IndexedMap* – set with a prefixed order of elements, allowing fast access by index or by value (hash-based);
- *NCollection_DataMap* – hash map;
- *NCollection_IndexedDataMap* – map with a prefixed order of elements, allowing fast access by index or by value (hash-based);
- *NCollection_DoubleMap* – two-side hash map (with two keys).

**Other collection types**

There are 4 collection types provided as template classes:

- *NCollection_Vector*
- *NCollection_UBTree*
- *NCollection_SparseArray*
- *NCollection_CellFilter*

**Vector**

This type is implemented internally as a list of arrays of the same size. Its properties:

- Direct (constant-time) access to members like in Array1 type; data are allocated in compact blocks, this provides faster iteration.
- Can grow without limits, like List, Stack or Queue types.
- Once having the size LEN, it cannot be reduced to any size less than
LEN – there is no operation of removal of items.

Insertion in a Vector-type class is made by two methods:

- **SetValuex(ind, theValue)** – array-type insertion, where ind is the index of the inserted item, can be any non-negative number. If it is greater than or equal to Length(), then the vector is enlarged (its Length() grows).
- **Append(theValue)** – list-type insertion equivalent to
  \[ \text{myVec.SetValue(myVec.Length(), theValue)} \]
  incrementing the size of the collection.

Other essential properties coming from List and Array1 type collections:

- Like in List, the method **Clear()** destroys all contained objects and releases the allocated memory.
- Like in Array1, the methods **Value()** and **ChangeValue()** return a contained object by index. Also, these methods have the form of overloaded operator ()

**UBTree**

The name of this type stands for “Unbalanced Binary Tree”. It stores the members in a binary tree of overlapped bounding objects (boxes or else). Once the tree of boxes of geometric objects is constructed, the algorithm is capable of fast geometric selection of objects. The tree can be easily updated by adding to it a new object with bounding box. The time of adding to the tree of one object is O(log(N)), where N is the total number of objects, so the time of building a tree of N objects is O(N(log(N)). The search time of one object is O(log(N)).

Defining various classes inheriting **NCollection_UBTree::Selector** we can perform various kinds of selection over the same b-tree object.

The object may be of any type allowing copying. Among the best suitable solutions there can be a pointer to an object, handled object or integer index of object inside some collection. The bounding object may have any dimension and geometry. The minimal interface of **TheBndType** (besides public empty and copy constructor and operator =) used in UBTTree algorithm as follows:
class MyBndType
{
    public:
    inline void Add (const MyBndType& other);
    // Updates me with other bounding type instance

    inline Standard_Boolean IsOut (const MyBndType& other) const;
    // Classifies other bounding type instance relatively me

    inline Standard_Real SquareExtent() const;
    // Computes the squared maximal linear extent of me (for a box it is the squared diagonal of the box).
};

This interface is implemented in types of Bnd package: Bnd_Box, Bnd_Box2d, Bnd_B2x, Bnd_B3x.

To select objects you need to define a class derived from UBTree::Selector that should redefine the necessary virtual methods to maintain the selection condition. Usually this class instance is also used to retrieve selected objects after search. The class UBTreeFiller is used to randomly populate a UBTree instance. The quality of a tree is better (considering the speed of searches) if objects are added to it in a random order trying to avoid the addition of a chain of nearby objects one following another. Instantiation of UBTreeFiller collects objects to be added, and then adds them at once to the given UBTree instance in a random order using the Fisher-Yates algorithm. Below is the sample code that creates an instance of NCollection_UBTree indexed by 2D boxes (Bnd_B2f), then a selection is performed returning the objects whose bounding boxes contain the given 2D point.

typedef NCollection_UBTree<MyData, Bnd_B2f> UBTree;
typedef NCollection_List<MyData>ListOfSelected;
//! Tree Selector type
class MyTreeSelector : public UBTree::Selector
public:
  // This constructor initializes the selection criterion (e.g., a point)
  MyTreeSelector (const gp_XY& thePnt) :
    myPnt(thePnt) {}
  // Get the list of selected objects
  const ListOfSelected& ListAccepted () const
  { return myList; }
  // Bounding box rejection - definition of virtual method. @return True if theBox is outside the selection criterion.
  Standard_Boolean Reject (const Bnd_B2f& theBox) const
  { return theBox.IsOut(myPnt); }
  // Redefined from the base class. Called when the bounding of theData conforms to the selection criterion. This method updates myList.
  Standard_Boolean Accept (const MyData& theData)
  { myList.Append(theData); }
private:
  gp_XY myPnt;
  ListOfSelected myList;
};

// Create a UBTree instance and fill it with data, each data item having the corresponding 2D box.
UBTTree aTree;
NCollection_UBTreeFiller <MyData, Bnd_B2f> aTreeFiller(aTree);
for(;;) {
  const MyData& aData = ...;
```cpp
const Bnd_B2d& aBox = aData.GetBox();
aTreeFiller.Add(aData, aBox);
}
aTreeFiller.Fill();

// Perform selection based on ‘aPoint2d’
MyTreeSelector aSel(aPoint2d);
aTree.Select(aSel);
const ListOfSelected = aSel.ListAccepted();
```

**SparseArray**

This type has almost the same features as Vector but it allows to store items having scattered indices. In Vector, if you set an item with index 1000000, the container will allocate memory for all items with indices in the range 0-1000000. In SparseArray, only one small block of items will be reserved that contains the item with index 1000000.

This class can be also seen as equivalence of `DataMap<int, TheItemType>` with the only one practical difference: it can be much less memory-expensive if items are small (e.g. Integer or Handle).

This type has both interfaces of DataMap and Vector to access items.

**CellFilter**

This class represents a data structure for sorting geometric objects in n-dimensional space into cells, with associated algorithm for fast checking of coincidence (overlapping, intersection, etc.) with other objects. It can be considered as a functional alternative to UBTree, as in the best case it provides the direct access to an object like in an n-dimensional array, while search with UBTree provides logarithmic law access time.

**Features**

NCollection defines some specific features, in addition to the public API inherited from TCollection classes.
Iterators

Every collection defines its Iterator class capable of iterating the members in some predefined order. Every Iterator is defined as a subtype of the particular collection type (e.g., MyPackage_StackOfPnt::Iterator). The order of iteration is defined by a particular collection type. The methods of Iterator are:

- `void Init (const MyCollection&)` – initializes the iterator on the collection object;
- `Standard_Boolean More () const` – makes a query if there is another non-iterated member;
- `void Next ()` – increments the iterator;
- `const ItemType& Value () const` – returns the current member;
- `ItemType& ChangeValue () const` – returns the mutable current member

```cpp
typedef Ncollection_Sequence<gp_Pnt> MyPackage_SequenceOfPnt
void Perform (const MyPackage_SequenceOfPnt& theSequence)
{
    MyPackage_SequenceOfPnt::Iterator anIter (theSequence);
    for (; anIter.More(); anIter.Next()) {
        const gp_Pnt aPnt& = anIter.Value();
        ....
    }
}
```

This feature is present only for some classes in TCollection (Stack, List, Set, Map, DataMap, DoubleMap). In NCollection it is generalized.

Class BaseCollection

There is a common abstract base class for all collections for a given item type (e.g., gp_Pnt). Developer X can arbitrarily name this base class like MyPackage_BaseCollPnt in the examples above. This name is further used in the declarations of any (non-abstract) collection class to
designate the C++ inheritance.

This base class has the following public API:

- abstract class Iterator as the base class for all Iterators described above;
- `Iterator& CreateIterator() const` – creates and returns the Iterator on this collection;
- `Standard_Integer Size() const` – returns the number of items in this collection;
- `void Assign(const NCollection_BaseCollection& theOther)` – copies the contents of the Other to this collection object;

These members enable accessing any collection without knowing its exact type. In particular, it makes possible to implement methods receiving objects of the abstract collection type:

```cpp
#include <NColection_Map.hxx>
typedef NCollection_Map<gp_Pnt> MyPackage_MapOfPnt;
typedef NCollection_BaseCollection<gp_Pnt> MyPackage_BaseCollPnt;
MyPackage_MapOfPnt aMapPnt;
...
gp_Pnt aResult = COG(aMapPnt);
...
gp_Pnt COG(const MyPackage_BaseCollPnt& theColl)
{
    gp_XYZ aCentreOfGravity(0., 0., 0.);
    // create type-independent iterator (it is abstract type instance)
    MyPackage_BaseCollString::Iterator& anIter = theColl.CreateIterator();
    for (; anIter.More(); anIter.Next()) {
        aCentreOfGravity += anIter.Value().XYZ();
    }
    return aCentreOfGravity / theColl.Size();
}
```

Note that there are fundamental differences between the shown type-
independent iterator and the iterator belonging to a particular non-abstract collection:

- Type-independent iterator can only be obtained via the call `CreateIterator()`; the typed iterator – only via the explicit construction.
- Type-independent iterator is an abstract class, so it is impossible to copy it or to assign it to another collection object; the typed iterators can be copied and reassigned using the method `Init()`.
- Type-independent iterator is actually destroyed when its collection object is destroyed; the typed iterator is destroyed as any other C++ object in the corresponding C++ scope.

The common point between them is that it is possible to create any number of both types of iterators on the same collection object.

**Heterogeneous Assign**

The semantics of the method `Assign()` has been changed in comparison to `TCollection`. In `NCollection` classes the method `Assign()` is virtual and it receives the object of the abstract `BaseCollection` class (see the previous section). Therefore this method can be used to assign any collection type to any other if only these collections are instantiated on the same `ItemType`.

For example, conversion of `Map` into `Array1` is performed like this:

```cpp
#include <NCollection_Map.hxx>
#include <NCollection_Array1.hxx>
typedef NCollection_Map<gp_Pnt> MyPackage_MapOfPnt;
typedef NCollection_Array1<gp_Pnt> MyPackage_Array1OfPnt;
    MyPackage_Array1OfPnt anArr1Pnt (1, aMapPnt.Size());
anArr1Pnt.Assign (aMapPnt); // heterogeneous assignment
```

There are some aspects to mention:
Unlike in \textit{TCollection}, in \textit{NCollection} the methods \textit{Assign} and \textit{operator=} do not coincide. The former is a virtual method defined in the \textit{BaseCollection} class. The latter is always defined in instance classes as a non-virtual inline method and it corresponds exactly to the method \textit{Assign} in \textit{TCollection} classes. Therefore it is always profitable to use \textit{operator=} instead of \textit{Assign} wherever the types on both sides of assignment are known.

If the method \textit{Assign} copies to \textit{Array1} or \textit{Array2} structure, it first checks if the size of the array is equal to the number of items in the copied collection object. If the sizes differ, an exception is thrown, as in \textit{TCollection\_Array1.gxx}.

Copying to \textit{Map}, \textit{IndexedMap}, \textit{DataMap} and \textit{IndexedDataMap} can bring about a loss of data: when two or more copied data items have the same key value, only one item is copied and the others are discarded. It can lead to an error in the code like the following:

```cpp
MyPackage\_Array10fPnt anArr1Pnt (1, 100);
MyPackage\_Map0fPnt aMapPnt;
....
aMapPnt.Assign(anArr1Pnt);
anArr1Pnt.Assign(aMapPnt);
```

Objects of classes parameterised with two types (\textit{DoubleMap}, \textit{DataMap} and \textit{IndexedDataMap}) cannot be assigned. Their method \textit{Assign} throws the exception \textit{Standard\_TypeMismatch} (because it is impossible to check if the passed \textit{BaseCollection} parameter belongs to the same collection type).

**Allocator**

All constructors of \textit{NCollection} classes receive the \textit{Allocator} Object as the last parameter. This is an object of a type managed by Handle, inheriting \textit{NCollection\_BaseAllocator}, with the following (mandatory) methods redefined:

```cpp
Standard\__EXPORT virtual void\* Allocate (const size_t size);
Standard\__EXPORT virtual void Free (void \* anAddress);
```
It is used internally every time when the collection allocates memory for its item(s) and releases this memory. The default value of this parameter (empty Handle) designates the use of NCollection_BaseAllocator X where the functions Standard::Allocate and Standard::Free are called. Therefore if the user of NCollection does not specify any allocator as a parameter to the constructor of his collection, the memory management will be identical to the one in TCollection and other Open CASCADE Technology classes.

Nevertheless, the it is possible to define a custom Allocator type to manage the memory in the most optimal or convenient way for his algorithms.

As one possible choice, the class NCollection_IncAllocator is included. Unlike BaseAllocator, it owns all memory it allocates from the system. Memory is allocated in big blocks (about 20kB) and the allocator keeps track of the amount of occupied memory. The method Allocate just increments the pointer to non-occupied memory and returns its previous value. Memory is only released in the destructor of IncAllocator, the method Free is empty. If used efficiently, this Allocator can greatly improve the performance of OCCT collections.
Strings

Strings are classes that handle dynamically sized sequences of characters based on ASCII/Unicode UTF-8 (normal 8-bit character type) and UTF-16/UCS-2 (16-bit character type). They provide editing operations with built-in memory management which make the relative objects easier to use than ordinary character arrays.

String classes provide the following services to manipulate character strings:

- Editing operations on string objects, using a built-in string manager
- Handling of dynamically-sized sequences of characters
- Conversion from/to ASCII and UTF-8 strings.

Strings may also be manipulated by handles and therefore shared.

Examples

TCollection_AsciiString

A variable-length sequence of ASCII characters (normal 8-bit character type). It provides editing operations with built-in memory management to make AsciiString objects easier to use than ordinary character arrays. AsciiString objects follow value semantics; that is, they are the actual strings, not handles to strings, and are copied through assignment. You may use HAsciiString objects to get handles to strings.

TCollection_ExtendedString

A variable-length sequence of "extended" (UNICODE) characters (16-bit character type). It provides editing operations with built-in memory management to make ExtendedString objects easier to use than ordinary extended character arrays.

ExtendedString objects follow value semantics; that is, they are the actual strings, not handles to strings, and are copied through assignment. You may use HExtendedString objects to get handles to strings.
TCollection_HAsciiString

A variable-length sequence of ASCII characters (normal 8-bit character type). It provides editing operations with built-in memory management to make HAsciiString objects easier to use than ordinary character arrays. HAsciiString objects are handles to strings.

- HAsciiString strings may be shared by several objects.
- You may use an AsciiString object to get the actual string. HAsciiString objects use an AsciiString string as a field.

TCollection_HExtendedString

A variable-length sequence of extended; (UNICODE) characters (16-bit character type). It provides editing operations with built-in memory management to make ExtendedString objects easier to use than ordinary extended character arrays. HExtendedString objects are handles to strings.

- HExtendedString strings may be shared by several objects.
- You may use an ExtendedString object to get the actual string. HExtendedString objects use an ExtendedString string as a field.

Conversion

Resource_Unicode provides functions to convert a non-ASCII C string given in ANSI, EUC, GB or SJIS format, to a Unicode string of extended characters, and vice versa.
Quantities

Quantities are various classes supporting date and time information and fundamental types representing most physical quantities such as length, area, volume, mass, density, weight, temperature, pressure etc.

Quantity classes provide the following services:

- Definition of primitive types representing most of mathematical and physical quantities;
- Unit conversion tools providing a uniform mechanism for dealing with quantities and associated physical units: check unit compatibility, perform conversions of values between different units, etc. (see package UnitsAPI)
- Resources to manage time information such as dates and time periods
- Resources to manage color definition

A mathematical quantity is characterized by the name and the value (real).

A physical quantity is characterized by the name, the value (real) and the unit. The unit may be either an international unit complying with the International Unit System (SI) or a user defined unit. The unit is managed by the physical quantity user.

The fact that both physical and mathematical quantities are manipulated as real values means that:

- They are defined as aliases of real values, so all functions provided by the Standard_Real class are available on each quantity.
- It is possible to mix several physical quantities in a mathematical or physical formula involving real values.

Quantity package includes all commonly used basic physical quantities.
Unit Conversion

The *UnitsAPI* global functions are used to convert a value from any unit into another unit. Conversion is executed among three unit systems:

- the **SI System**,
- the user’s **Local System**,  
- the user’s **Current System**. The **SI System** is the standard international unit system. It is indicated by *SI* in the signatures of the *UnitsAPI* functions.

The OCCT (former MDTV) System corresponds to the SI international standard but the length unit and all its derivatives use the millimeter instead of the meter.

Both systems are proposed by Open CASCADE Technology; the SI System is the standard option. By selecting one of these two systems, you define your **Local System** through the *SetLocalSystem* function. The **Local System** is indicated by *LS* in the signatures of the *UnitsAPI* functions. The Local System units can be modified in the working environment. You define your **Current System** by modifying its units through the *SetCurrentUnit* function. The Current System is indicated by *Current* in the signatures of the *UnitsAPI* functions. A physical quantity is defined by a string (example: LENGTH).
Math Primitives and Algorithms

Overview

Math primitives and algorithms available in Open CASCADE Technology include:

- Vectors and matrices
- Geometric primitives
- Math algorithms
Vectors and Matrices

The Vectors and Matrices component provides a C++ implementation of the fundamental types Vector and Matrix, which are regularly used to define more complex data structures.

The Vector and Matrix classes provide commonly used mathematical algorithms which include:

- Basic calculations involving vectors and matrices;
- Computation of eigenvalues and eigenvectors of a square matrix;
- Solvers for a set of linear algebraic equations;
- Algorithms to find the roots of a set of non-linear equations;
- Algorithms to find the minimum function of one or more independent variables.

These classes also provide a data structure to represent any expression, relation, or function used in mathematics, including the assignment of variables.

Vectors and matrices have arbitrary ranges which must be defined at declaration time and cannot be changed after declaration.

```cpp
math_Vector v(1, 3);
// a vector of dimension 3 with range (1..3)
math_Matrix m(0, 2, 0, 2);
// a matrix of dimension 3x3 with range (0..2, 0..2)
math_Vector v(N1, N2);
// a vector of dimension N2-N1+1 with range (N1..N2)
```

Vector and Matrix objects use value semantics. In other words, they cannot be shared and are copied through assignment.

```cpp
math_Vector v1(1, 3), v2(0, 2);
v2 = v1;
// v1 is copied into v2. a modification of v1 does not affect v2
```
Vector and Matrix values may be initialized and obtained using indexes which must lie within the range definition of the vector or the matrix.

```cpp
math_Vector v(1, 3);
math_Matrix m(1, 3, 1, 3);
Standard_Real value;

v(2) = 1.0;
value = v(1);
m(1, 3) = 1.0;
value = m(2, 2);
```

Some operations on Vector and Matrix objects may not be legal. In this case an exception is raised. Two standard exceptions are used:

- *Standard_DimensionError* exception is raised when two matrices or vectors involved in an operation are of incompatible dimensions.
- *Standard_RangeError* exception is raised if an access outside the range definition of a vector or of a matrix is attempted.

```cpp
math_Vector v1(1, 3), v2(1, 2), v3(0, 2);
v1 = v2;
// error: Standard_DimensionError is raised

v1 = v3;
// OK: ranges are not equal but dimensions are compatible

v1(0) = 2.0;
// error: Standard_RangeError is raised
```
Primitive Geometric Types

Open CASCADE Technology primitive geometric types are a STEP-compliant implementation of basic geometric and algebraic entities. They provide:

- Descriptions of primitive geometric shapes, such as:
  - Points;
  - Vectors;
  - Lines;
  - Circles and conics;
  - Planes and elementary surfaces;
- Positioning of these shapes in space or in a plane by means of an axis or a coordinate system;
- Definition and application of geometric transformations to these shapes:
  - Translations;
  - Rotations;
  - Symmetries;
  - Scaling transformations;
  - Composed transformations;
- Tools (coordinates and matrices) for algebraic computation.

All these functions are provided by geometric processor package gp. Its classes for 2d and 3d objects are handled by value rather than by reference. When this sort of object is copied, it is copied entirely. Changes in one instance will not be reflected in another.

The gp package defines the basic geometric entities used for algebraic calculation and basic analytical geometry in 2d & 3d space. It also provides basic transformations such as identity, rotation, translation, mirroring, scale transformations, combinations of transformations, etc. Entities are handled by value.

Please, note that gp curves and surfaces are analytic: there is no parameterization and no orientation on gp entities, i.e. these entities do not provide functions which work with these properties.

If you need, you may use more evolved data structures provided by
Geom (in 3D space) and Geom2d (in the plane). However, the definition of gp entities is identical to the one of equivalent Geom and Geom2d entities, and they are located in the plane or in space with the same kind of positioning systems. They implicitly contain the orientation, which they express on the Geom and Geom2d entities, and they induce the definition of their parameterization.

Therefore, it is easy to give an implicit parameterization to gp curves and surfaces, which is the parametrization of the equivalent Geom or Geom2d entity. This property is particularly useful when computing projections or intersections, or for operations involving complex algorithms where it is particularly important to manipulate the simplest data structures, i.e. those of gp. Thus, ElCLib and ElSLib packages provide functions to compute:

- the point of parameter u on a 2D or 3D gp curve,
- the point of parameter (u,v) on a gp elementary surface, and
- any derivative vector at this point.

Note: the gp entities cannot be shared when they are inside more complex data structures.
Collections of Primitive Geometric Types

Before creating a geometric object, you must decide whether you are in a 2d or in a 3d context and how you want to handle the object. If you do not need a single instance of a geometric primitive but a set of them then the package which deals with collections of this sort of object, \textit{TColgp}, will provide the necessary functionality. In particular, this package provides standard and frequently used instantiations of generic classes with geometric objects, i.e. \textit{XY, XYZ, Pnt, Pnt2d, Vec, Vec2d, Lin, Lin2d, Circ, Circ2d}. 
Basic Geometric Libraries

There are various library packages available which offer a range of basic computations on curves and surfaces. If you are dealing with objects created from the gp package, the useful algorithms are in the elementary curves and surfaces libraries – the EICLib and EISLib packages.

- **EICLib** provides methods for analytic curves. This is a library of simple computations on curves from the gp package (Lines, Circles and Conics). It is possible to compute points with a given parameter or to compute the parameter for a point.
- **EISLib** provides methods for analytic surfaces. This is a library of simple computations on surfaces from the package gp (Planes, Cylinders, Spheres, Cones, Tori). It is possible to compute points with a given pair of parameters or to compute the parameter for a point. There is a library for calculating normals on curves and surfaces.

Additionally, Bnd package provides a set of classes and tools to operate with bounding boxes of geometric objects in 2d and 3d space.
Common Math Algorithms

The common math algorithms library provides a C++ implementation of the most frequently used mathematical algorithms. These include:

- Algorithms to solve a set of linear algebraic equations,
- Algorithms to find the minimum of a function of one or more independent variables,
- Algorithms to find roots of one, or of a set, of non-linear equations,
- An algorithm to find the eigenvalues and eigenvectors of a square matrix.

All mathematical algorithms are implemented using the same principles. They contain: A constructor performing all, or most of, the calculation, given the appropriate arguments. All relevant information is stored inside the resulting object, so that all subsequent calculations or interrogations will be solved in the most efficient way.

A function IsDone returning the boolean true if the calculation was successful. A set of functions, specific to each algorithm, enabling all the various results to be obtained. Calling these functions is legal only if the function IsDone answers true, otherwise the exception StdFail_NotDone is raised.

The example below demonstrates the use of the Gauss class, which implements the Gauss solution for a set of linear equations. The following definition is an extract from the header file of the class math_Gauss:

```cpp
class Gauss {
public:
    Gauss (const math_Matrix& A);
    Standard_Boolean IsDone() const;
    void Solve (const math_Vector& B,
                math_Vector& X) const;
};
```

Now the main program uses the Gauss class to solve the equations a*x1=b1 and a*x2=b2:
The next example demonstrates the use of the *BissecNewton* class, which implements a combination of the Newton and Bissection algorithms to find the root of a function known to lie between two bounds. The definition is an extract from the header file of the class *math*BissecNewton:

```cpp
class BissecNewton {
    public:
        BissecNewton (math_FunctionWithDerivative& f, const Standard_Real bound1, const Standard_Real bound2, const Standard_Real tolx);
        Standard_Boolean IsDone() const;
}
```
The abstract class `math_FunctionWithDerivative` describes the services which have to be implemented for the function \( f \) which is to be used by a \textit{BissecNewton} algorithm. The following definition corresponds to the header file of the abstract class `math_FunctionWithDerivative`:

```cpp
class math_FunctionWithDerivative {
public:
    virtual Standard_Boolean Value (const Standard_Real x, Standard_Real& f) = 0;
    virtual Standard_Boolean Derivative (const Standard_Real x, Standard_Real& d) = 0;
    virtual Standard_Boolean Values (const Standard_Real x, Standard_Real& f, Standard_Real& d) = 0;
};
```

Now the test sample uses the \textit{BissecNewton} class to find the root of the equation \( f(x) = x^2 - 4 \) in the interval \([1.5, 2.5] \): the function to solve is implemented in the class `myFunction` which inherits from the class `math_FunctionWithDerivative`, then the main program finds the required root.

```cpp
#include <math_BissecNewton.hxx>
#include <math_FunctionWithDerivative.hxx>
class myFunction : public math_FunctionWithDerivative {
    Standard_Real coefa, coefb, coefc;

public:
    myFunction (const Standard_Real a, const Standard_Real b,
                const Standard_Real c) :
```
virtual Standard_Boolean Value (const Standard_Real x,
                                          Standard_Real& f)
{
    f = coefa * x * x + coefb * x + coefc;
}

virtual Standard_Boolean Derivative (const Standard_Real x,
                                           Standard_Real& d)
{
    d = coefa * x * 2.0 + coefb;
}

virtual Standard_Boolean Values (const Standard_Real x,
                                       Standard_Real& f,
                                       Standard_Real& d)
{
    f = coefa * x * x * x + coefb * x + coefc;
    d = coefa * x * 2.0 + coefb;
}
};

main()
{
    myFunction f(1.0, 0.0, 4.0);
    math_BissecNewton sol(F, 1.5, 2.5, 0.0000001);
    if(Sol.IsDone()) { // is it OK ?
        Standard_Real x = sol.Root(); // yes.
    } else { // no
}
Precision

On the OCCT platform, each object stored in the database should carry its own precision value. This is important when dealing with systems where objects are imported from other systems as well as with various associated precision values.

The *Precision* package addresses the daily problem of the geometric algorithm developer: what precision setting to use to compare two numbers. Real number equivalence is clearly a poor choice. The difference between the numbers should be compared to a given precision setting.

Do not write `if (X1 == X2)`, instead write `if (Abs(X1-X2) < Precision)`. 

Also, to order real numbers, keep in mind that `if (X1 < X2 - Precision)` is incorrect. `if (X2 - X1 > Precision)` is far better when `X1` and `X2` are high numbers.

This package proposes a set of methods providing precision settings for the most commonly encountered situations.

In Open CASCADE Technology, precision is usually not implicit; low-level geometric algorithms accept precision settings as arguments. Usually these should not refer directly to this package.

High-level modeling algorithms have to provide a precision setting to the low level geometric algorithms they call. One way is to use the settings provided by this package. The high-level modeling algorithms can also have their own strategy for managing precision. As an example the Topology Data Structure stores precision values which are later used by algorithms. When a new topology is created, it takes the stored value. Different precision settings offered by this package cover the most common needs of geometric algorithms such as *Intersection* and *Approximation*. The choice of a precision value depends both on the algorithm and on the geometric space. The geometric space may be either:

- a real space, 3d or 2d where the lengths are measured in meters,
micron, inches, etc.

- a parametric space, 1d on a curve or 2d on a surface where numbers have no dimension. The choice of precision value for parametric space depends not only on the accuracy of the machine, but also on the dimensions of the curve or the surface. This is because it is desirable to link parametric precision and real precision. If you are on a curve defined by the equation $P(t)$, you would want to have equivalence between the following:

$$\text{Abs}(t_1-t_2) < \text{ParametricPrecision}$$
$$\text{Distance } (P(t_1),P(t_2)) < \text{RealPrecision}.$$

**The Precision package**

The *Precision* package offers a number of package methods and default precisions for use in dealing with angles, distances, intersections, approximations, and parametric space. It provides values to use in comparisons to test for real number equalities.

- Angular precision compares angles.
- Confusion precision compares distances.
- Intersection precision is used by intersection algorithms.
- Approximation precision is used by approximation algorithms.
- Parametric precision gets a parametric space precision from a 3D precision.
- *Infinite* returns a high number that can be considered to be infinite. Use *-Infinite* for a high negative number.

**Standard Precision values**

This package provides a set of real space precision values for algorithms. The real space precisions are designed for precision to 0.1 nanometers. The only unit available is the millimeter. The parametric precisions are derived from the real precisions by the *Parametric* function. This applies a scaling factor which is the length of a tangent to the curve or the surface. You, the user, provide this length. There is a default value for a curve with $[0,1]$ parameter space and a length less than 100 meters. The geometric packages provide Parametric precisions for the different types of curves. The *Precision* package provides methods to test whether a real number can be considered to be infinite.
Precision::Angular

This method is used to compare two angles. Its current value is $Epsilon(2 \pi)$ i.e. the smallest number $x$ such that $2\pi + x$ is different of $2\pi$.

It can be used to check confusion of two angles as follows: $\text{Abs}(\text{Angle1} - \text{Angle2}) < \text{Precision::Angular()}$

It is also possible to check parallelism of two vectors ($\text{Vec}$ from $\text{gp}$) as follows: $\text{V1.IsParallel(V2,\text{Precision::Angular()})}$

Note that Precision::Angular() can be used on both dot and cross products because for small angles the $\text{Sine}$ and the $\text{Angle}$ are equivalent. So to test if two directions of type $\text{gp\_Dir}$ are perpendicular, it is legal to use the following code: $\text{Abs}(\text{D1} \ast \text{D2}) < \text{Precision::Angular()}$

Precision::Confusion

This method is used to test 3D distances. The current value is $1.e-7$, in other words, 1/10 micron if the unit used is the millimeter.

It can be used to check confusion of two points ($\text{Pnt}$ from $\text{gp}$) as follows: $\text{P1.IsEqual(P2,\text{Precision::Confusion()})}$

It is also possible to find a vector of null length ($\text{Vec}$ from $\text{gp}$) : $\text{V.Magnitude()} < \text{Precision::Confusion()}$

Precision::Intersection

This is reasonable precision to pass to an Intersection process as a limit of refinement of Intersection Points. Intersection is high enough for the process to converge quickly. Intersection is lower than Confusion so that you still get a point on the intersected geometries. The current value is $\text{Confusion() / 100}$.

Precision::Approximation

This is a reasonable precision to pass to an approximation process as a limit of refinement of fitting. The approximation is greater than the other
precisions because it is designed to be used when the time is at a premium. It has been provided as a reasonable compromise by the designers of the Approximation algorithm. The current value is $\text{Confusion}() \times 10$. Note that Approximation is greater than Confusion, so care must be taken when using Confusion in an approximation process.
Modeling Data

Table of Contents

- Introduction
- Geometry Utilities
  - Interpolations and Approximations
    - Analysis of a set of points
    - Basic Interpolation and Approximation
  - Advanced Approximation
- Direct Construction
  - Simple geometric entities
  - Geometric entities manipulated by handle
  - Conversion to and from BSplines
- Points on Curves
- Extrema
- 2D Geometry
- 3D Geometry
- Properties of Shapes
  - Local Properties of Shapes
  - Local Properties of
Curves and Surfaces

- Continuity of Curves and Surfaces
- Regularity of Shared Edges
- Global Properties of Shapes
- Adaptors for Curves and Surfaces

Topology

- Shape Location
- Naming shapes, sub-shapes, their orientation and state
  - Topological types
  - Orientation
  - State
- Manipulating shapes and sub-shapes
- Exploration of Topological Data Structures
- Lists and Maps of Shapes
  - Wire Explorer
- Storage of shapes
Introduction

Modeling Data supplies data structures to represent 2D and 3D geometric models.

This manual explains how to use Modeling Data. For advanced information on modeling data, see our E-learning & Training offerings.
Geometry Utilities

Geometry Utilities provide the following services:

- Creation of shapes by interpolation and approximation
- Direct construction of shapes
- Conversion of curves and surfaces to BSpline curves and surfaces
- Computation of the coordinates of points on 2D and 3D curves
- Calculation of extrema between shapes.
Interpolations and Approximations

In modeling, it is often required to approximate or interpolate points into curves and surfaces. In interpolation, the process is complete when the curve or surface passes through all the points; in approximation, when it is as close to these points as possible.

Approximation of Curves and Surfaces groups together a variety of functions used in 2D and 3D geometry for:

- the interpolation of a set of 2D points using a 2D BSpline or Bezier curve;
- the approximation of a set of 2D points using a 2D BSpline or Bezier curve;
- the interpolation of a set of 3D points using a 3D BSpline or Bezier curve, or a BSpline surface;
- the approximation of a set of 3D points using a 3D BSpline or Bezier curve, or a BSpline surface.

You can program approximations in two ways:

- Using high-level functions, designed to provide a simple method for obtaining approximations with minimal programming,
- Using low-level functions, designed for users requiring more control over the approximations.

Analysis of a set of points

The class PEquation from GProp package allows analyzing a collection or cloud of points and verifying if they are coincident, collinear or coplanar within a given precision. If they are, the algorithm computes the mean point, the mean line or the mean plane of the points. If they are not, the algorithm computes the minimal box, which includes all the points.

Basic Interpolation and Approximation

Packages Geom2dAPI and GeomAPI provide simple methods for approximation and interpolation with minimal programming
2D Interpolation

The class *Interpolate* from *Geom2dAPI* package allows building a constrained 2D BSpline curve, defined by a table of points through which the curve passes. If required, the parameter values and vectors of the tangents can be given for each point in the table.

3D Interpolation

The class *Interpolate* from *GeomAPI* package allows building a constrained 3D BSpline curve, defined by a table of points through which the curve passes. If required, the parameter values and vectors of the tangents can be given for each point in the table.

Approximation of a BSpline from scattered points

This class may be instantiated as follows:

```cpp
GeomAPI_Interpolate Interp(Points);
```

From this object, the BSpline curve may be requested as follows:

```cpp
Handle(Geom_BSplineCurve) C = Interp.Curve();
```

2D Approximation

The class *PointsToBSpline* from *Geom2dAPI* package allows building a 2DBSpline curve, which approximates a set of points. You have to define the lowest and highest degree of the curve, its continuity and a tolerance value for it. The tolerance value is used to check that points are not too close to each other, or tangential vectors not too small. The resulting
BSpline curve will be C2 or second degree continuous, except where a tangency constraint is defined on a point through which the curve passes. In this case, it will be only C1 continuous.

3D Approximation

The class *PointsToBSpline* from GeomAPI package allows building a 3D BSpline curve, which approximates a set of points. It is necessary to define the lowest and highest degree of the curve, its continuity and tolerance. The tolerance value is used to check that points are not too close to each other, or that tangential vectors are not too small.

The resulting BSpline curve will be C2 or second degree continuous, except where a tangency constraint is defined on a point, through which the curve passes. In this case, it will be only C1 continuous. This class is instantiated as follows:

```
GeomAPI_PointsToBSpline
Approx(Points, DegMin, DegMax, Continuity, Tol);
```

From this object, the BSpline curve may be requested as follows:

```
Handle(Geom_BSplineCurve) K = Approx.Curve();
```

Surface Approximation

The class *PointsToBSplineSurface* from GeomAPI package allows building a BSpline surface, which approximates or interpolates a set of points.

Advanced Approximation

Packages *AppDef* and *AppParCurves* provide low-level functions, allowing more control over the approximations.

The low-level functions provide a second API with functions to:

- Define compulsory tangents for an approximation. These tangents have origins and extremities.
Approximate a set of curves in parallel to respect identical parameterization.
Smooth approximations. This is to produce a faired curve.

You can also find functions to compute:

- The minimal box which includes a set of points
- The mean plane, line or point of a set of coplanar, collinear or coincident points.

**Approximation by multiple point constraints**

*AppDef* package provides low-level tools to allow parallel approximation of groups of points into Bezier or B-Spline curves using multiple point constraints.

The following low level services are provided:

- Definition of an array of point constraints:

  The class *MultiLine* allows defining a given number of multi-point constraints in order to build the multi-line, multiple lines passing through ordered multiple point constraints.

In this image:
- $P_i, Q_i, R_i \ldots S_i$ can be 2D or 3D points.
- Defined as a group: $P_n, Q_n, R_n, \ldots S_n$ form a MultipointConstraint. They possess the same passage, tangency and curvature constraints.
- $P_1, P_2, \ldots P_n$, or the $Q, R, \ldots$ or $S$ series represent the lines to be approximated.

- Definition of a set of point constraints:

  The class *MultiPointConstraint* allows defining a multiple point constraint and computing the approximation of sets of points to several curves.

- Computation of an approximation of a Bezier curve from a set of points:

  The class *Compute* allows making an approximation of a set of points to a Bezier curve

- Computation of an approximation of a BSpline curve from a set of points:

  The class *BSplineCompute* allows making an approximation of a set of points to a BSpline curve.

- Definition of Variational Criteria:

  The class *TheVariational* allows fairing the approximation curve to a given number of points using a least squares method in conjunction with a variational criterion, usually the weights at each constraint point.

**Approximation by parametric or geometric constraints**

*AppParCurves* package provides low-level tools to allow parallel approximation of groups of points into Bezier or B-Spline curve with parametric or geometric constraints, such as a requirement for the curve to pass through given points, or to have a given tangency or curvature at a particular point.

The algorithms used include:

- the least squares method
- a search for the best approximation within a given tolerance value.

The following low-level services are provided:

- Association of an index to an object:

The class `ConstraintCouple` allows you associating an index to an object to compute faired curves using `AppDef_TheVariational`.

- Definition of a set of approximations of Bezier curves:

The class `MultiCurve` allows defining the approximation of a multi-line made up of multiple Bezier curves.

- Definition of a set of approximations of BSpline curves:

The class `MultiBSpCurve` allows defining the approximation of a multi-line made up of multiple BSpline curves.

- Definition of points making up a set of point constraints

The class `MultiPoint` allows defining groups of 2D or 3D points making up a multi-line.

**Example: How to approximate a curve with respect to tangency**

To approximate a curve with respect to tangency, follow these steps:

1. Create an object of type `AppDef_MultiPointConstraints` from the set of points to approximate and use the method `SetTang` to set the tangency vectors.
2. Create an object of type `AppDef_MultiLine` from the `AppDef_MultiPointConstraint`.
3. Use `AppDef_BSplineCompute`, which instantiates `Approx_BSplineComputeLine` to perform the approximation.
Direct Construction

Direct Construction methods from gce, GC and GCE2d packages provide simplified algorithms to build elementary geometric entities such as lines, circles and curves. They complement the reference definitions provided by the gp, Geom and Geom2d packages.

The algorithms implemented by gce, GCE2d and GC packages are simple: there is no creation of objects defined by advanced positional constraints (for more information on this subject, see Geom2dGcc and GccAna, which describe geometry by constraints).

For example, to construct a circle from a point and a radius using the gp package, it is necessary to construct axis Ax2d before creating the circle. If gce package is used, and Ox is taken for the axis, it is possible to create a circle directly from a point and a radius.

Another example is the class gce_MakeCirc providing a framework for defining eight problems encountered in the geometric construction of circles and implementing the eight related construction algorithms.

The object created (or implemented) is an algorithm which can be consulted to find out, in particular:

- its result, which is a gp_Circ, and
- its status. Here, the status indicates whether or not the construction was successful.

If it was unsuccessful, the status gives the reason for the failure.

```cpp
gp_Pnt P1 (0.,0.,0.);
gp_Pnt P2 (0.,10.,0.);
gp_Pnt P3 (10.,0.,0.);
gce_MakeCirc MC (P1,P2,P3);
if (MC.IsDone()) {
    const gp_Circ& C = MC.Value();
}
```

In addition, gce, GCE2d and GC each have a Root class. This class is
the root of all classes in the package, which return a status. The returned status (successful construction or construction error) is described by the enumeration `gce_ErrorType`.

Note, that classes, which construct geometric transformations do not return a status, and therefore do not inherit from `Root`.

**Simple geometric entities**

The following algorithms used to build entities from `gp` package are provided by `gce` package.

- 2D line parallel to another at a distance,
- 2D line parallel to another passing through a point,
- 2D circle passing through two points,
- 2D circle parallel to another at a distance,
- 2D circle parallel to another passing through a point,
- 2D circle passing through three points,
- 2D circle from a center and a radius,
- 2D hyperbola from five points,
- 2D hyperbola from a center and two apexes,
- 2D ellipse from five points,
- 2D ellipse from a center and two apexes,
- 2D parabola from three points,
- 2D parabola from a center and an apex,
- line parallel to another passing through a point,
- line passing through two points,
- circle coaxial to another passing through a point,
- circle coaxial to another at a given distance,
- circle passing through three points,
- circle with its center, radius, and normal to the plane,
- circle with its axis (center + normal),
- hyperbola with its center and two apexes,
- ellipse with its center and two apexes,
- plane passing through three points,
- plane from its normal,
- plane parallel to another plane at a given distance,
- plane parallel to another passing through a point,
- plane from an array of points,
- cylinder from a given axis and a given radius,
- cylinder from a circular base,
- cylinder from three points,
- cylinder parallel to another cylinder at a given distance,
- cylinder parallel to another cylinder passing through a point,
- cone from four points,
- cone from a given axis and two passing points,
- cone from two points (an axis) and two radii,
- cone parallel to another at a given distance,
- cone parallel to another passing through a point,
- all transformations (rotations, translations, mirrors, scaling transformations, etc.).

Each class from `gp` package, such as `Circ`, `Circ2d`, `Mirror`, `Mirror2d`, etc., has the corresponding `MakeCirc`, `MakeCirc2d`, `MakeMirror`, `MakeMirror2d`, etc. class from `gce` package.

It is possible to create a point using a `gce` package class, then question it to recover the corresponding `gp` object.

```plaintext
gp_Pnt2d Point1, Point2;
...
// Initialization of Point1 and Point2
gce_MakeLin2d L = gce_MakeLin2d(Point1, Point2);
if (L.Status() == gce_Done()) {
    gp_Lin2d l = L.Value();
}
```

This is useful if you are uncertain as to whether the arguments can create the `gp` object without raising an exception. In the case above, if `Point1` and `Point2` are closer than the tolerance value required by `MakeLin2d`, the function `Status` will return the enumeration `gce_ConfusedPoint`. This tells you why the `gp` object cannot be created. If you know that the points `Point1` and `Point2` are separated by the value exceeding the tolerance value, then you may create the `gp` object directly, as follows:

```plaintext
gp_Lin2d l = gce_MakeLin2d(Point1, Point2);
```

**Geometric entities manipulated by handle**

`GC` and `GCE2d` packages provides an implementation of algorithms used
to build entities from *Geom* and *Geom2D* packages. They implement the same algorithms as the *gce* package, and also contain algorithms for trimmed surfaces and curves. The following algorithms are available:

- arc of a circle trimmed by two points,
- arc of a circle trimmed by two parameters,
- arc of a circle trimmed by one point and one parameter,
- arc of an ellipse from an ellipse trimmed by two points,
- arc of an ellipse from an ellipse trimmed by two parameters,
- arc of an ellipse from an ellipse trimmed by one point and one parameter,
- arc of a parabola from a parabola trimmed by two points,
- arc of a parabola from a parabola trimmed by two parameters,
- arc of a parabola from a parabola trimmed by one point and one parameter,
- arc of a hyperbola from a hyperbola trimmed by two points,
- arc of a hyperbola from a hyperbola trimmed by two parameters,
- arc of a hyperbola from a hyperbola trimmed by one point and one parameter,
- segment of a line from two points,
- segment of a line from two parameters,
- segment of a line from one point and one parameter,
- trimmed cylinder from a circular base and a height,
- trimmed cylinder from three points,
- trimmed cylinder from an axis, a radius, and a height,
- trimmed cone from four points,
- trimmed cone from two points (an axis) and a radius,
- trimmed cone from two coaxial circles.

Each class from *GCE2d* package, such as *Circle*, *Ellipse*, *Mirror*, etc., has the corresponding *MakeCircle*, *MakeEllipse*, *MakeMirror*, etc. class from *Geom2d* package. Besides, the class *MakeArcOfCircle* returns an object of type *TrimmedCurve* from *Geom2d*.

Each class from *GC* package, such as *Circle*, *Ellipse*, *Mirror*, etc., has the corresponding *MakeCircle*, *MakeEllipse*, *MakeMirror*, etc. class from *Geom* package. The following classes return objects of type *TrimmedCurve* from *Geom*:

- *MakeArcOfCircle*
- *MakeArcOfEllipse*
• MakeArcOfHyperbola
• MakeArcOfParabola
• MakeSegment
Conversion to and from BSplines

The Conversion to and from BSplines component has two distinct purposes:

- Firstly, it provides a homogeneous formulation which can be used to describe any curve or surface. This is useful for writing algorithms for a single data structure model. The BSpline formulation can be used to represent most basic geometric objects provided by the components which describe geometric data structures ("Fundamental Geometry Types", "2D Geometry Types" and "3D Geometry Types" components).
- Secondly, it can be used to divide a BSpline curve or surface into a series of curves or surfaces, thereby providing a higher degree of continuity. This is useful for writing algorithms which require a specific degree of continuity in the objects to which they are applied. Discontinuities are situated on the boundaries of objects only.

The "Conversion to and from BSplines" component is composed of three packages.

The Convert package provides algorithms to convert the following into a BSpline curve or surface:

- a bounded curve based on an elementary 2D curve (line, circle or conic) from the gp package,
- a bounded surface based on an elementary surface (cylinder, cone, sphere or torus) from the gp package,
- a series of adjacent 2D or 3D Bezier curves defined by their poles.

These algorithms compute the data needed to define the resulting BSpline curve or surface. This elementary data (degrees, periodic characteristics, poles and weights, knots and multiplicities) may then be used directly in an algorithm, or can be used to construct the curve or the surface by calling the appropriate constructor provided by the classes Geom2d_BSplineCurve, Geom_BSplineCurve or Geom_BSplineSurface.

The Geom2dConvert package provides the following:
- a global function which is used to construct a BSpline curve from a bounded curve based on a 2D curve from the Geom2d package,
- a splitting algorithm which computes the points at which a 2D BSpline curve should be cut in order to obtain arcs with the same degree of continuity,
- global functions used to construct the BSpline curves created by this splitting algorithm, or by other types of segmentation of the BSpline curve,
- an algorithm which converts a 2D BSpline curve into a series of adjacent Bezier curves.

The GeomConvert package also provides the following:

- a global function used to construct a BSpline curve from a bounded curve based on a curve from the Geom package,
- a splitting algorithm, which computes the points at which a BSpline curve should be cut in order to obtain arcs with the same degree of continuity,
- global functions to construct BSpline curves created by this splitting algorithm, or by other types of BSpline curve segmentation,
- an algorithm, which converts a BSpline curve into a series of adjacent Bezier curves,
- a global function to construct a BSpline surface from a bounded surface based on a surface from the Geom package,
- a splitting algorithm, which determines the curves along which a BSpline surface should be cut in order to obtain patches with the same degree of continuity,
- global functions to construct BSpline surfaces created by this splitting algorithm, or by other types of BSpline surface segmentation,
- an algorithm, which converts a BSpline surface into a series of adjacent Bezier surfaces,
- an algorithm, which converts a grid of adjacent Bezier surfaces into a BSpline surface.
Points on Curves

The Points on Curves component comprises high level functions providing an API for complex algorithms that compute points on a 2D or 3D curve.

The following characteristic points exist on parameterized curves in 3d space:

- points equally spaced on a curve,
- points distributed along a curve with equal chords,
- a point at a given distance from another point on a curve.

GCPnts package provides algorithms to calculate such points:

- \textit{AbscissaPoint} calculates a point on a curve at a given distance from another point on the curve.
- \textit{UniformAbscissa} calculates a set of points at a given abscissa on a curve.
- \textit{UniformDeflection} calculates a set of points at maximum constant deflection between the curve and the polygon that results from the computed points.

Example: Visualizing a curve.

Let us take an adapted curve \( C \), i.e. an object which is an interface between the services provided by either a 2D curve from the package Geom2d (in case of an Adaptor\_Curve2d curve) or a 3D curve from the package Geom (in case of an Adaptor\_Curve curve), and the services required on the curve by the computation algorithm. The adapted curve is created in the following way:

2D case:

\begin{verbatim}
Handle(Geom2d_Curve) mycurve = ... ;
Geom2dAdaptor_Curve C (mycurve) ;
\end{verbatim}

3D case:
Handle(Geom_Curve) mycurve = ... ;  
GeomAdaptor_Curve C (mycurve) ; 

The algorithm is then constructed with this object:

GCPnts_UniformDeflection myAlgo () ;  
Standard_Real Deflection = ... ;  
myAlgo.Initialize ( C , Deflection ) ;  
if ( myAlgo.IsDone() ) 
{
  Standard_Integer nbr = myAlgo.NbPoints() ;  
  Standard_Real param ;  
  for ( Standard_Integer i = 1 ; i <= nbr ; i++ ) 
  {
    param = myAlgo.Parameter ( i ) ;  
    ...  
  }
}
Extrema

The classes to calculate the minimum distance between points, curves, and surfaces in 2d and 3d are provided by GeomAPI and Geom2dAPI packages.

These packages calculate the extrema of distance between:

- point and a curve,
- point and a surface,
- two curves,
- a curve and a surface,
- two surfaces.

Extrema between Point and Curve / Surface

The GeomAPI_ProjectPointOnCurve class allows calculation of all extrema between a point and a curve. Extrema are the lengths of the segments orthogonal to the curve. The GeomAPI_ProjectPointOnSurface class allows calculation of all extrema between a point and a surface. Extrema are the lengths of the segments orthogonal to the surface. These classes use the "Projection" criteria for optimization.

Extrema between Curves

The Geom2dAPI_ExtremaCurveCurve class allows calculation of all minimal distances between two 2D geometric curves. The GeomAPI_ExtremaCurveCurve class allows calculation of all minimal distances between two 3D geometric curves. These classes use Euclidean distance as the criteria for optimization.

Extrema between Curve and Surface

The GeomAPI_ExtremaCurveSurface class allows calculation of one extrema between a 3D curve and a surface. Extrema are the lengths of the segments orthogonal to the curve and the surface. This class uses the "Projection" criteria for optimization.
Extrema between Surfaces

The GeomAPI_ExtremaSurfaceSurface class allows calculation of one minimal and one maximal distance between two surfaces. This class uses Euclidean distance to compute the minimum, and "Projection" criteria to compute the maximum.
2D Geometry

*Geom2d* package defines geometric objects in 2dspace. All geometric entities are STEP processed. The objects are handled by reference.

In particular, *Geom2d* package provides classes for:

- description of points, vectors and curves,
- their positioning in the plane using coordinate systems,
- their geometric transformation, by applying translations, rotations, symmetries, scaling transformations and combinations thereof.

The following objects are available:

- point,
- Cartesian point,
- vector,
- direction,
- vector with magnitude,
- axis,
- curve,
- line,
- conic: circle, ellipse, hyperbola, parabola,
- rounded curve: trimmed curve, NURBS curve, Bezier curve,
- offset curve.

Before creating a geometric object, it is necessary to decide how the object is handled. The objects provided by *Geom2d* package are handled by reference rather than by value. Copying an instance copies the handle, not the object, so that a change to one instance is reflected in each occurrence of it. If a set of object instances is needed rather than a single object instance, *TColGeom2d* package can be used. This package provides standard and frequently used instantiations of one-dimensional arrays and sequences for curves from *Geom2d* package. All objects are available in two versions:

- handled by reference and
- handled by value.
The key characteristic of *Geom2d* curves is that they are parameterized. Each class provides functions to work with the parametric equation of the curve, and, in particular, to compute the point of parameter \( u \) on a curve and the derivative vectors of order 1, 2..., \( N \) at this point.

As a consequence of the parameterization, a *Geom2d* curve is naturally oriented.

Parameterization and orientation differentiate elementary *Geom2d* curves from their equivalent as provided by *gp* package. *Geom2d* package provides conversion functions to transform a *Geom2d* object into a *gp* object, and vice-versa, when this is possible.

Moreover, *Geom2d* package provides more complex curves, including Bezier curves, BSpline curves, trimmed curves and offset curves.

*Geom2d* objects are organized according to an inheritance structure over several levels.

Thus, an ellipse (specific class *Geom2d_Ellipse*) is also a conical curve and inherits from the abstract class *Geom2d_Conic*, while a Bezier curve (concrete class *Geom2d_BezierCurve*) is also a bounded curve and inherits from the abstract class *Geom2d_BoundedCurve*; both these examples are also curves (abstract class *Geom2d_Curve*). Curves, points and vectors inherit from the abstract class *Geom2d_Geometry*, which describes the properties common to any geometric object from the *Geom2d* package.

This inheritance structure is open and it is possible to describe new objects, which inherit from those provided in the *Geom2d* package, provided that they respect the behavior of the classes from which they are to inherit.

Finally, *Geom2d* objects can be shared within more complex data structures. This is why they are used within topological data structures, for example.

*Geom2d* package uses the services of the *gp* package to:

- implement elementary algebraic calculus and basic analytic geometry,
• describe geometric transformations which can be applied to \textit{Geom2d} objects,
• describe the elementary data structures of \textit{Geom2d} objects.

However, the \textit{Geom2d} package essentially provides data structures and not algorithms. You can refer to the \textit{GCE2d} package to find more evolved construction algorithms for \textit{Geom2d} objects.
3D Geometry

The *Geom* package defines geometric objects in 3d space and contains all basic geometric transformations, such as identity, rotation, translation, mirroring, scale transformations, combinations of transformations, etc. as well as special functions depending on the reference definition of the geometric object (e.g. addition of a control point on a B-Spline curve, modification of a curve, etc.). All geometrical entities are STEP processed.

In particular, it provides classes for:

- description of points, vectors, curves and surfaces,
- their positioning in 3D space using axis or coordinate systems, and
- their geometric transformation, by applying translations, rotations, symmetries, scaling transformations and combinations thereof.

The following objects are available:

- Point
- Cartesian point
- Vector
- Direction
- Vector with magnitude
- Axis
- Curve
- Line
- Conic: circle, ellipse, hyperbola, parabola
- Offset curve
- Elementary surface: plane, cylinder, cone, sphere, torus
- Bounded curve: trimmed curve, NURBS curve, Bezier curve
- Bounded surface: rectangular trimmed surface, NURBS surface, Bezier surface
- Swept surface: surface of linear extrusion, surface of revolution
- Offset surface.

The key characteristic of *Geom* curves and surfaces is that they are parameterized. Each class provides functions to work with the parametric equation of the curve or surface, and, in particular, to compute:
the point of parameter \( u \) on a curve, or
the point of parameters \((u, v)\) on a surface, together with the derivative vectors of order 1, 2, ... \( N \) at this point.

As a consequence of this parameterization, a Geom curve or surface is naturally oriented.

Parameterization and orientation differentiate elementary Geom curves and surfaces from the classes of the same (or similar) names found in \( gp \) package. Geom package also provides conversion functions to transform a Geom object into a \( gp \) object, and vice-versa, when such transformation is possible.

Moreover, Geom package provides more complex curves and surfaces, including:

- Bezier and BSpline curves and surfaces,
- swept surfaces, for example surfaces of revolution and surfaces of linear extrusion,
- trimmed curves and surfaces, and
- offset curves and surfaces.

Geom objects are organized according to an inheritance structure over several levels. Thus, a sphere (concrete class Geom\_SphericalSurface) is also an elementary surface and inherits from the abstract class Geom\_ElementarySurface, while a Bezier surface (concrete class Geom\_BezierSurface) is also a bounded surface and inherits from the abstract class Geom\_BoundedSurface; both these examples are also surfaces (abstract class Geom\_Surface). Curves, points and vectors inherit from the abstract class Geom\_Geometry, which describes the properties common to any geometric object from the Geom package.

This inheritance structure is open and it is possible to describe new objects, which inherit from those provided in the Geom package, on the condition that they respect the behavior of the classes from which they are to inherit.

Finally, Geom objects can be shared within more complex data structures. This is why they are used within topological data structures, for example.
If a set of object instances is needed rather than a single object instance, *TColGeom* package can be used. This package provides instantiations of one- and two-dimensional arrays and sequences for curves from *Geom* package. All objects are available in two versions:

- handled by reference and
- handled by value.

The *Geom* package uses the services of the *gp* package to:

- implement elementary algebraic calculus and basic analytic geometry,
- describe geometric transformations which can be applied to Geom objects,
- describe the elementary data structures of Geom objects.

However, the Geom package essentially provides data structures, not algorithms.

You can refer to the *GC* package to find more evolved construction algorithms for Geom objects.
Properties of Shapes

Local Properties of Shapes

\textit{BRepLProp} package provides the Local Properties of Shapes component, which contains algorithms computing various local properties on edges and faces in a BRep model.

The local properties which may be queried are:

- for a point of parameter \textit{u} on a curve which supports an edge:
  - the point,
  - the derivative vectors, up to the third degree,
  - the tangent vector,
  - the normal,
  - the curvature, and the center of curvature;
- for a point of parameter \textit{(u, v)} on a surface which supports a face:
  - the point,
  - the derivative vectors, up to the second degree,
  - the tangent vectors to the \textit{u} and \textit{v} isoparametric curves,
  - the normal vector,
  - the minimum or maximum curvature, and the corresponding directions of curvature;
- the degree of continuity of a curve which supports an edge, built by the concatenation of two other edges, at their junction point.

Analyzed edges and faces are described as \textit{BRepAdaptor} curves and surfaces, which provide shapes with an interface for the description of their geometric support. The base point for local properties is defined by its \textit{u} parameter value on a curve, or its \textit{(u, v)} parameter values on a surface.
Local Properties of Curves and Surfaces

The "Local Properties of Curves and Surfaces" component provides algorithms for computing various local properties on a Geom curve (in 2D or 3D space) or a surface. It is composed of:

- **Geom2dLProp** package, which allows computing Derivative and Tangent vectors (normal and curvature) of a parametric point on a 2D curve;
- **GeomLProp** package, which provides local properties on 3D curves and surfaces;
- **LProp** package, which provides an enumeration used to characterize a particular point on a 2D curve.

Curves are either Geom_Curve curves (in 3D space) or Geom2d_Curve curves (in the plane). Surfaces are Geom_Surface surfaces. The point on which local properties are calculated is defined by its u parameter value on a curve, and its (u,v) parameter values on a surface.

It is possible to query the same local properties for points as mentioned above, and additionally for 2D curves:

- the points corresponding to a minimum or a maximum of curvature;
- the inflection points.

**Example: How to check the surface concavity**

To check the concavity of a surface, proceed as follows:

1. Sample the surface and compute at each point the Gaussian curvature.
2. If the value of the curvature changes of sign, the surface is concave or convex depending on the point of view.
3. To compute a Gaussian curvature, use the class SLprops from GeomLProp, which instantiates the generic class SLProps from LProp and use the method GaussianCurvature.
Continuity of Curves and Surfaces

Types of supported continuities for curves and surfaces are described in `GeomAbs_Shape` enumeration.

In respect of curves, the following types of continuity are supported (see the figure below):

- **C0** (`GeomAbs_C0`) - parametric continuity. It is the same as G0 (geometric continuity), so the last one is not represented by separate variable.
- **G1** (`GeomAbs_G1`) - tangent vectors on left and on right are parallel.
- **C1** (`GeomAbs_C1`) - indicates the continuity of the first derivative.
- **G2** (`GeomAbs_G2`) - in addition to G1 continuity, the centers of curvature on left and on right are the same.
- **C2** (`GeomAbs_C2`) - continuity of all derivatives till the second order.
- **C3** (`GeomAbs_C3`) - continuity of all derivatives till the third order.
- **CN** (`GeomAbs_CN`) - continuity of all derivatives till the N-th order (infinite order of continuity).

*Note:* Geometric continuity (G1, G2) means that the curve can be reparametrized to have parametric (C1, C2) continuity.
Continuity of Curves

- **C0 continuity**
  - (coincident boundaries)

- **G1 continuity**
  - (C0 + parallel tangent vectors)

- **C1 continuity**
  - (C0 + equal tangent vectors)

- **G2 continuity**
  - (G1 + same center of curvature)

- **C2 continuity**
  - (C1 + same center of curvature)
The following types of surface continuity are supported:

- C0 (GeomAbs_C0) - parametric continuity (the surface has no points or curves of discontinuity).
- G1 (GeomAbs_G1) - surface has single tangent plane in each point.
- C1 (GeomAbs_C1) - indicates the continuity of the first derivatives.
- G2 (GeomAbs_G2) - in addition to G1 continuity, principal curvatures and directions are continuous.
- C2 (GeomAbs_C2) - continuity of all derivatives till the second order.
- C3 (GeomAbs_C3) - continuity of all derivatives till the third order.
- CN (GeomAbs_CN) - continuity of all derivatives till the N-th order (infinite order of continuity).
C0 continuity
(different normal vectors along shared curve)

G1 continuity
(same tangent plane)

C1 continuity
(G1 + equal tangent vectors)

G2 continuity
(G1 + equal principal curvatures)

C2 continuity
(C1 + equal principal curvatures and directions)

Continuity of Surfaces
Against single surface, the connection of two surfaces (see the figure above) defines its continuity in each intersection point only. Smoothness of connection is a minimal value of continuities on the intersection curve.
Regularity of Shared Edges

Regularity of an edge is a smoothness of connection of two faces sharing this edge. In other words, regularity is a minimal continuity between connected faces in each point on edge.

Edge's regularity can be set by `BRep_Builder::Continuity` method. To get the regularity use `BRep_Tool::Continuity` method.

Some algorithms like `Fillet` set regularity of produced edges by their own algorithms. On the other hand, some other algorithms (like `Boolean Operations`, `Shape Healing`, etc.) do not set regularity. If the regularity is needed to be set correctly on a shape, the method `BRepLib::EncodeRegularity` can be used. It calculates and sets correct values for all edges of the shape.

The regularity flag is extensively used by the following high level algorithms: `Chamfer, Draft Angle, Hidden Line Removal, Gluer`. 
Global Properties of Shapes

The Global Properties of Shapes component provides algorithms for computing the global properties of a composite geometric system in 3D space, and frameworks to query the computed results.

The global properties computed for a system are:

- mass,
- mass center,
- matrix of inertia,
- moment about an axis,
- radius of gyration about an axis,
- principal properties of inertia such as principal axis, principal moments, and principal radius of gyration.

Geometric systems are generally defined as shapes. Depending on the way they are analyzed, these shapes will give properties of:

- lines induced from the edges of the shape,
- surfaces induced from the faces of the shape, or
- volumes induced from the solid bounded by the shape.

The global properties of several systems may be brought together to give the global properties of the system composed of the sum of all individual systems.

The Global Properties of Shapes component is composed of:

- seven functions for computing global properties of a shape: one function for lines, two functions for surfaces and four functions for volumes. The choice of functions depends on input parameters and algorithms used for computation (BRepGProp global functions),
- a framework for computing global properties for a set of points (GProp_PGProps),
- a general framework to bring together the global properties retained by several more elementary frameworks, and provide a general programming interface to consult computed global properties.
Packages \textit{GeomLProp} and \textit{Geom2dLProp} provide algorithms calculating the local properties of curves and surfaces.

A curve (for one parameter) has the following local properties:

- Point
- Derivative
- Tangent
- Normal
- Curvature
- Center of curvature.

A surface (for two parameters U and V) has the following local properties:

- point
- derivative for U and V)
- tangent line (for U and V)
- normal
- max curvature
- min curvature
- main directions of curvature
- mean curvature
- Gaussian curvature

The following methods are available:

- \textit{CLProps} – calculates the local properties of a curve (tangency, curvature,normal);
- \textit{CurAndInf2d} – calculates the maximum and minimum curvatures and the inflection points of 2d curves;
- \textit{SLProps} – calculates the local properties of a surface (tangency, the normal and curvature).
- \textit{Continuity} – calculates regularity at the junction of two curves.

Note that the B-spline curve and surface are accepted but they are not cut into pieces of the desired continuity. It is the global continuity, which is seen.
Adaptors for Curves and Surfaces

Some Open CASCADE Technology general algorithms may work theoretically on numerous types of curves or surfaces.

To do this, they simply get the services required of the analyzed curve or surface through an interface so as to a single API, whatever the type of curve or surface. These interfaces are called adaptors.

For example, Adaptor3d_Curve is the abstract class which provides the required services by an algorithm which uses any 3d curve.

GeomAdaptor package provides interfaces:

- On a Geom curve;
- On a curve lying on a Geom surface;
- On a Geom surface;

Geom2dAdaptor package provides interfaces:

- On a Geom2d curve.

BRepAdaptor package provides interfaces:

- On a Face
- On an Edge

When you write an algorithm which operates on geometric objects, use Adaptor3d (or Adaptor2d) objects.

As a result, you can use the algorithm with any kind of object, if you provide for this object an interface derived from Adaptor3d or Adaptor2d. These interfaces are easy to use: simply create an adapted curve or surface from a Geom2d curve, and then use this adapted curve as an argument for the algorithm which requires it.
Topology

OCCT Topology allows accessing and manipulating data of objects without dealing with their 2D or 3D representations. Whereas OCCT Geometry provides a description of objects in terms of coordinates or parametric values, Topology describes data structures of objects in parametric space. These descriptions use location in and restriction of parts of this space.

Topological library allows you to build pure topological data structures. Topology defines relationships between simple geometric entities. In this way, you can model complex shapes as assemblies of simpler entities. Due to a built-in non-manifold (or mixed-dimensional) feature, you can build models mixing:

- 0D entities such as points;
- 1D entities such as curves;
- 2D entities such as surfaces;
- 3D entities such as volumes.

You can, for example, represent a single object made of several distinct bodies containing embedded curves and surfaces connected or non-connected to an outer boundary.

Abstract topological data structure describes a basic entity – a shape, which can be divided into the following component topologies:

- Vertex – a zero-dimensional shape corresponding to a point in geometry;
- Edge – a shape corresponding to a curve, and bound by a vertex at each extremity;
- Wire – a sequence of edges connected by their vertices;
- Face – part of a plane (in 2D geometry) or a surface (in 3D geometry) bounded by a closed wire;
- Shell – a collection of faces connected by some edges of their wire boundaries;
- Solid – a part of 3D space bound by a shell;
- Compound solid – a collection of solids.
The wire and the solid can be either infinite or closed.

A face with 3D underlying geometry may also refer to a collection of connected triangles that approximate the underlying surface. The surfaces can be undefined leaving the faces represented by triangles only. If so, the model is purely polyhedral.

Topology defines the relationship between simple geometric entities, which can thus be linked together to represent complex shapes.

Abstract Topology is provided by six packages. The first three packages describe the topological data structure used in Open CASCADE Technology:

- **TopAbs** package provides general resources for topology-driven applications. It contains enumerations that are used to describe basic topological notions: topological shape, orientation and state. It also provides methods to manage these enumerations.
- **TopLoc** package provides resources to handle 3D local coordinate systems: **Datum3D** and **Location**. **Datum3D** describes an elementary coordinate system, while **Location** comprises a series of elementary coordinate systems.
- **TopoDS** package describes classes to model and build data structures that are purely topological.

Three additional packages provide tools to access and manipulate this abstract topology:

- **TopTools** package provides basic tools to use on topological data structures.
- **TopExp** package provides classes to explore and manipulate the topological data structures described in the TopoDS package.
- **BRepTools** package provides classes to explore, manipulate, read and write BRep data structures. These more complex data structures combine topological descriptions with additional geometric information, and include rules for evaluating equivalence of different possible representations of the same object, for example, a point.
Shape Location

A local coordinate system can be viewed as either of the following:

- A right-handed trihedron with an origin and three orthonormal vectors. The *gp_Ax2* package corresponds to this definition.
- A transformation of a +1 determinant, allowing the transformation of coordinates between local and global references frames. This corresponds to the *gp_Trsf*.

*TopLoc* package distinguishes two notions:

- *TopLoc_Datum3D* class provides the elementary reference coordinate, represented by a right-handed orthonormal system of axes or by a right-handed unitary transformation.
- *TopLoc_Location* class provides the composite reference coordinate made from elementary ones. It is a marker composed of a chain of references to elementary markers. The resulting cumulative transformation is stored in order to avoid recalculating the sum of the transformations for the whole list.
Structure of TopLoc_Location

Two reference coordinates are equal if they are made up of the same elementary coordinates in the same order. There is no numerical comparison. Two coordinates can thus correspond to the same transformation without being equal if they were not built from the same elementary coordinates.

For example, consider three elementary coordinates: R1, R2, R3. The composite coordinates are: C1 = R1 * R2, C2 = R2 * R3, C3 = C1 * R3, C4 = R1 * C2

NOTE C3 and C4 are equal because they are both R1 * R2 * R3.

The TopLoc package is chiefly targeted at the topological data structure, but it can be used for other purposes.
Change of coordinates

*TopLoc_Datum3D* class represents a change of elementary coordinates. Such changes must be shared so this class inherits from *Standard_Transient*. The coordinate is represented by a transformation *gp_Trsfpackage*. This transformation has no scaling factor.
Naming shapes, sub-shapes, their orientation and state

The **TopAbs** package provides general enumerations describing the basic concepts of topology and methods to handle these enumerations. It contains no classes. This package has been separated from the rest of the topology because the notions it contains are sufficiently general to be used by all topological tools. This avoids redefinition of enumerations by remaining independent of modeling resources. The TopAbs package defines three notions:

- **Type** `TopAbs_ShapeEnum`;
- **Orientation** `TopAbs_Orientation`;
- **State** `StateTopAbs_State`

**Topological types**

TopAbs contains the `TopAbs_ShapeEnum` enumeration, which lists the different topological types:

- **COMPOUND** – a group of any type of topological objects.
- **COMPSOLID** – a composite solid is a set of solids connected by their faces. It expands the notions of WIRE and SHELL to solids.
- **SOLID** – a part of space limited by shells. It is three dimensional.
- **SHELL** – a set of faces connected by their edges. A shell can be open or closed.
- **FACE** – in 2D it is a part of a plane; in 3D it is a part of a surface. Its geometry is constrained (trimmed) by contours. It is two dimensional.
- **WIRE** – a set of edges connected by their vertices. It can be an open or closed contour depending on whether the edges are linked or not.
- **EDGE** – a topological element corresponding to a restrained curve. An edge is generally limited by vertices. It has one dimension.
- **VERTEX** – a topological element corresponding to a point. It has zero dimension.
- **SHAPE** – a generic term covering all of the above.

A topological model can be considered as a graph of objects with adjacency relationships. When modeling a part in 2D or 3D space it must
belong to one of the categories listed in the ShapeEnum enumeration. The TopAbs package lists all the objects, which can be found in any model. It cannot be extended but a subset can be used. For example, the notion of solid is useless in 2D.

The terms of the enumeration appear in order from the most complex to the most simple, because objects can contain simpler objects in their description. For example, a face references its wires, edges, and vertices.

**Orientation**

The notion of orientation is represented by the `TopAbs_Orientation` enumeration. Orientation is a generalized notion of the sense of direction found in various modelers. This is used when a shape limits a geometric domain; and is closely linked to the notion of boundary. The three cases are the following:
- Curve limited by a vertex.
- Surface limited by an edge.
- Space limited by a face.

In each case the topological form used as the boundary of a geometric domain of a higher dimension defines two local regions of which one is arbitrarily considered as the **default region**.

For a curve limited by a vertex the default region is the set of points with parameters greater than the vertex. That is to say it is the part of the curve after the vertex following the natural direction along the curve.

For a surface limited by an edge the default region is on the left of the edge following its natural direction. More precisely it is the region pointed to by the vector product of the normal vector to the surface and the vector tangent to the curve.

For a space limited by a face the default region is found on the negative side of the normal to the surface.

Based on this default region the orientation allows definition of the region to be kept, which is called the **interior or material**. There are four orientations defining the interior.

<table>
<thead>
<tr>
<th>Orientation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORWARD</td>
<td>The interior is the default region.</td>
</tr>
<tr>
<td>REVERSED</td>
<td>The interior is the region complementary to the default.</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>The interior includes both regions. The boundary lies inside the material. For example a surface inside a solid.</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>The interior includes neither region. The boundary lies outside the material. For example an edge in a wire-frame model.</td>
</tr>
</tbody>
</table>
The notion of orientation is a very general one, and it can be used in any context where regions or boundaries appear. Thus, for example, when describing the intersection of an edge and a contour it is possible to describe not only the vertex of intersection but also how the edge crosses the contour considering it as a boundary. The edge would therefore be divided into two regions: exterior and interior and the intersection vertex would be the boundary. Thus an orientation can be associated with an intersection vertex as in the following figure:

<table>
<thead>
<tr>
<th>Orientation</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORWARD</td>
<td>Entering</td>
</tr>
<tr>
<td>REVERSED</td>
<td>Exiting</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>Touching from inside</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>Touching from outside</td>
</tr>
</tbody>
</table>

Along with the Orientation enumeration the TopAbs package defines four methods:

**State**
The **TopAbs_State** enumeration described the position of a vertex or a set of vertices with respect to a region. There are four terms:

<table>
<thead>
<tr>
<th>Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>The point is interior.</td>
</tr>
<tr>
<td>OUT</td>
<td>The point is exterior.</td>
</tr>
<tr>
<td>ON</td>
<td>The point is on the boundary(within tolerance).</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>The state of the point is indeterminate.</td>
</tr>
</tbody>
</table>

The **UNKNOWN** term has been introduced because this enumeration is often used to express the result of a calculation, which can fail. This term can be used when it is impossible to know if a point is inside or outside, which is the case with an open wire or face.

The State enumeration can also be used to specify various parts of an object. The following figure shows the parts of an edge intersecting a face.
State specifies the parts of an edge intersecting a face
Manipulating shapes and sub-shapes

The TopoDS package describes the topological data structure with the following characteristics:

- reference to an abstract shape with neither orientation nor location.
- Access to the data structure through the tool classes.

As stated above, OCCT Topology describes data structures of objects in parametric space. These descriptions use localization in and restriction of parts of this space. The types of shapes, which can be described in these terms, are the vertex, the face and the shape. The vertex is defined in terms of localization in parametric space, and the face and shape, in terms of restriction of this space.

OCCT topological descriptions also allow the simple shapes defined in these terms to be combined into sets. For example, a set of edges forms a wire; a set of faces forms a shell, and a set of solids forms a composite solid (CompSolid in Open CASCADE Technology). You can also combine shapes of either sort into compounds. Finally, you can give a shape an orientation and a location.

Listing shapes in order of complexity from vertex to composite solid leads us to the notion of the data structure as knowledge of how to break a shape down into a set of simpler shapes. This is in fact, the purpose of the TopoDS package.

The model of a shape is a shareable data structure because it can be used by other shapes. (An edge can be used by more than one face of a solid). A shareable data structure is handled by reference. When a simple reference is insufficient, two pieces of information are added: an orientation and a local coordinate reference.

- An orientation tells how the referenced shape is used in a boundary (Orientation from TopAbs).
- A local reference coordinate (Location from TopLoc) allows referencing a shape at a position different from that of its definition.

The TopoDS_TShape class is the root of all shape descriptions. It
contains a list of shapes. Classes inheriting TopoDS_TShape can carry the description of a geometric domain if necessary (for example, a geometric point associated with a TVertex). A TopoDS_TShape is a description of a shape in its definition frame of reference. This class is manipulated by reference.

The TopoDS_Shape class describes a reference to a shape. It contains a reference to an underlying abstract shape, an orientation, and a local reference coordinate. This class is manipulated by value and thus cannot be shared.

The class representing the underlying abstract shape is never referenced directly. The TopoDS_Shape class is always used to refer to it.

The information specific to each shape (the geometric support) is always added by inheritance to classes deriving from TopoDS_TShape. The following figures show the example of a shell formed from two faces connected by an edge.
In the previous diagram, the shell is described by the underlying shape TS, and the faces by TF1 and TF2. There are seven edges from TE1 to TE7 and six vertices from TV1 to TV6.

The wire TW1 references the edges from TE1 to TE4; TW2 references from TE4 to TE7.

The vertices are referenced by the edges as follows: TE1(TV1,TV4), TE2(TV1,TV2), TE3(TV2,TV3), TE4(TV3,TV4), TE5(TV4,TV5), TE6(T5,TV6), TE7(TV3,TV6).

Note that this data structure does not contain any back references. All references go from more complex underlying shapes to less complex ones. The techniques used to access the information are described later. The data structure is as compact as possible. Sub-objects can be shared among different objects.

Two very similar objects, perhaps two versions of the same object, might
share identical sub-objects. The usage of local coordinates in the data structure allows the description of a repetitive sub-structure to be shared.

The compact data structure avoids the loss of information associated with copy operations which are usually used in creating a new version of an object or when applying a coordinate change.

The following figure shows a data structure containing two versions of a solid. The second version presents a series of identical holes bored at different positions. The data structure is compact and yet keeps all information on the sub-elements.

The three references from $TSh2$ to the underlying face $TFcyl$ have associated local coordinate systems, which correspond to the successive positions of the hole.
Classes inheriting TopoDS_Shape

TopoDS is based on class TopoDS_Shape and the class defining its underlying shape. This has certain advantages, but the major drawback is that these classes are too general. Different shapes they could represent do not type them (Vertex, Edge, etc.) hence it is impossible to introduce checks to avoid incoherences such as inserting a face in an edge.

TopoDS package offers two sets of classes, one set inheriting the underlying shape with neither orientation nor location and the other inheriting TopoDS_Shape, which represent the standard topological shapes enumerated in TopAbs package.

The following classes inherit Shape: TopoDS_Vertex, TopoDS_Edge, TopoDS_Wire, TopoDS_Face, TopoDS_Shell, TopoDS_Solid, TopoDS_CompSolid, and TopoDS_Compound. In spite of the similarity of names with those inheriting from TopoDS_TShape there is a profound difference in the way they are used.

TopoDS_Shape class and the classes, which inherit from it, are the natural means to manipulate topological objects. TopoDS_TShape classes are hidden. TopoDS_TShape describes a class in its original local coordinate system without orientation. TopoDS_Shape is a reference to TopoDS_TShape with an orientation and a local reference.

TopoDS_TShape class is deferred; TopoDS_Shape class is not. Using TopoDS_Shape class allows manipulation of topological objects without knowing their type. It is a generic form. Purely topological algorithms often use the TopoDS_Shape class.

TopoDS_TShape class is manipulated by reference; TopoDS_Shape class by value. A TopoDS_Shape is nothing more than a reference enhanced with an orientation and a local coordinate. The sharing of TopoDS_Shapes is meaningless. What is important is the sharing of the underlying TopoDS_TShapes. Assignment or passage in argument does not copy the data structure: this only creates new TopoDS_Shapes which refer to the same TopoDS_TShape.
Although classes inheriting TopoDS_TShape are used for adding extra information, extra fields should not be added in a class inheriting from TopoDS_Shape. Classes inheriting from TopoDS_Shape serve only to specialize a reference in order to benefit from static type control (carried out by the compiler). For example, a routine that receives a TopoDS_Face in argument is more precise for the compiler than the one, which receives a TopoDS_Shape. It is pointless to derive other classes than those found in TopoDS. All references to a topological data structure are made with the Shape class and its inheritors defined in TopoDS.

There are no constructors for the classes inheriting from the TopoDS_Shape class, otherwise the type control would disappear through implicit casting (a characteristic of C++). The TopoDS package provides package methods for casting an object of the TopoDS_Shape class in one of these sub-classes, with type verification.

The following example shows a routine receiving an argument of the TopoDS_Shape type, then putting it into a variable V if it is a vertex or calling the method ProcessEdge if it is an edge.

```cpp
#include <TopoDS_Vertex.hxx>
#include <TopoDS_Edge.hxx>
#include <TopoDS_Shape.hxx>

void ProcessEdge(const TopoDS_Edge&);

void Process(const TopoDS_Shape& aShape) {
  if (aShape.Shapetype() == TopAbs_VERTEX) {
    TopoDS_Vertex V;
    V = TopoDS::Vertex(aShape); // Also correct
    TopoDS_Vertex V2 = aShape; // Rejected by the compiler
    TopoDS_Vertex V3 = TopoDS::Vertex(aShape); // Correct
  }
  else if (aShape.ShapeType() == TopAbs_EDGE){
    ProcessEdge(aShape); // This is rejected
    ProcessEdge(TopoDS::Edge(aShape)); // Correct
  }
}
```
} else {
    cout << "Neither a vertex nor an edge ?";
    ProcessEdge(TopoDS::Edge(aShape));
    // OK for compiler but an exception will be raised at run-time
}
Exploration of Topological Data Structures

The TopExp package provides tools for exploring the data structure described with the TopoDS package. Exploring a topological structure means finding all sub-objects of a given type, for example, finding all the faces of a solid.

The TopExp package provides the class TopExp_Explorer to find all sub-objects of a given type. An explorer is built with:

- The shape to be explored.
- The type of shapes to be found e.g. VERTEX, EDGE with the exception of SHAPE, which is not allowed.
- The type of Shapes to avoid. e.g. SHELL, EDGE. By default, this type is SHAPE. This default value means that there is no restriction on the exploration.

The Explorer visits the whole structure in order to find the shapes of the requested type not contained in the type to avoid. The example below shows how to find all faces in the shape S:

```cpp
void test() {
    TopoDS_Shape S;
    TopExp_Explorer Ex;
    for (Ex.Init(S,TopAbs_FACE); Ex.More(); Ex.Next()) {
        ProcessFace(Ex.Current());
    }
}
```

Find all the vertices which are not in an edge

```cpp
for (Ex.Init(S,TopAbs_VERTEX,TopAbs_EDGE); ...)
```

Find all the faces in a SHELL, then all the faces not in a SHELL:

```cpp
void test() {
    TopExp_Explorer Ex1, Ex2;
    TopoDS_Shape S;
```
for (Ex1.Init(S,TopAbs_SHELL);Ex1.More();
   Ex1.Next()){
   // visit all shells
for (Ex2.Init(Ex1.Current(),TopAbs_FACE);Ex2.More();
   Ex2.Next()){
   //visit all the faces of the current shell
   ProcessFaceinAshell(Ex2.Current());
   ...
}
}
for(Ex1.Init(S,TopAbs_FACE,TopAbs_SHELL);Ex1.More();
   Ex1.Next()){
   // visit all faces not in a shell.
   ProcessFace(Ex1.Current());
}

The Explorer presumes that objects contain only objects of an equal or inferior type. For example, if searching for faces it does not look at wires, edges, or vertices to see if they contain faces.

The MapShapes method from TopExp package allows filling a Map. An exploration using the Explorer class can visit an object more than once if it is referenced more than once. For example, an edge of a solid is generally referenced by two faces. To process objects only once, they have to be placed in a Map.

Example

```cpp
void TopExp::MapShapes (const TopoDS_Shape& S,
   const TopAbs_ShapeEnum T,
   TopTools_IndexedMapOfShape& M)
{
   TopExp_Explorer Ex(S,T);
   while (Ex.More()) {
      M.Add(Ex.Current());
      Ex.Next();
   }
```
In the following example all faces and all edges of an object are drawn in accordance with the following rules:

- The faces are represented by a network of $NbIso$ iso-parametric lines with $FacelIsoColor$ color.
- The edges are drawn in a color, which indicates the number of faces sharing the edge:
  - $FreeEdgeColor$ for edges, which do not belong to a face (i.e. wireframe element).
  - $BorderEdgeColor$ for an edge belonging to a single face.
  - $SharedEdgeColor$ for an edge belonging to more than one face.
- The methods $DrawEdge$ and $DrawFacelIso$ are also available to display individual edges and faces.

The following steps are performed:

1. Storing the edges in a map and create in parallel an array of integers to count the number of faces sharing the edge. This array is initialized to zero.
2. Exploring the faces. Each face is drawn.
3. Exploring the edges and for each of them increment the counter of faces in the array.
4. From the Map of edges, drawing each edge with the color corresponding to the number of faces.

```cpp
void DrawShape ( const TopoDS_Shape& aShape,
const Standard_Integer nbIsos,
const Color FaceIsocolor,
const Color FreeEdgeColor,
const Color BorderEdgeColor,
const Color SharedEdgeColor)
{
    // Store the edges in aMap.
    TopTools_IndexedMapOfShape edgemap;
    TopExp::MapShapes(aShape,TopAbs_EDGE,edgeMap);
    // Create an array set to zero.
    TColStd_Array1OfInteger faceCount(1,edgeMap.Extent());
```
faceCount.Init (0);
// Explore the faces.
  TopExp_Explorer expFace(aShape,TopAbs_FACE);
while (expFace.More()) {
  // Draw the current face.

  DrawFaceIsos(TopoDS::Face(expFace.Current()),nbIsos,FaceIsoColor);
  // Explore the edges of the face.
  TopExp_Explorer expEdge(expFace.Current(),TopAbs_EDGE);
  while (expEdge.More()) {
    // Increment the face count for this edge.

    faceCount(edgemap.FindIndex(expEdge.Current()))+;
    expEdge.Next();
  }
  expFace.Next();
}
// Draw the edges of the map
Standard_Integer i;
for (i=1;i<=edgemap.Extent();i++) {
  switch (faceCount(i)) {
    case 0 :
      DrawEdge(TopoDS::Edge(edgemap(i)),FreeEdgeColor);
      break;
    case 1 :
      DrawEdge(TopoDS::Edge(edgemap(i)),BorderEdgeColor);
      break;
    default :
      DrawEdge(TopoDS::Edge(edgemap(i)),SharedEdgeColor);
      break;
  }
r);
break;
}
Lists and Maps of Shapes

**TopTools** package contains tools for exploiting the *TopoDS* data structure. It is an instantiation of the tools from *TCollection* package with the Shape classes of *TopoDS*.

- **TopTools_Array1OfShape, HArray1OfShape** – instantiation of the *TCollection_Array1* and *TCollection_HArray1* with *TopoDS_Shape*.
- **TopTools_SequenceOfShape** – instantiation of the *TCollection_Sequence* with *TopoDS_Shape*.
- **TopTools_MapOfShape** - instantiation of the *TCollection_Map*. Allows the construction of sets of shapes.
- **TopTools_IndexedMapOfShape** - instantiation of the *TCollection_IndexedMap*. Allows the construction of tables of shapes and other data structures.

With a **TopTools_Map**, a set of references to Shapes can be kept without duplication. The following example counts the size of a data structure as a number of *TShapes*.

```c++
#include <TopoDS_Iterator.hxx>
Standard_Integer Size(const TopoDS_Shape & aShape)
{
    // This is a recursive method.
    // The size of a shape is 1 + the sizes of the subshapes.
    TopoDS_Iterator It;
    Standard_Integer size = 1;
    for (It.Initialize(aShape); It.More(); It.Next()) {
        size += Size(It.Value());
    }
    return size;
}
```

This program is incorrect if there is sharing in the data structure.

Thus for a contour of four edges it should count 1 wire + 4 edges + 4 vertices with the result 9, but as the vertices are each shared by two
edges this program will return 13. One solution is to put all the Shapes in a Map so as to avoid counting them twice, as in the following example:

```cpp
#include <TopoDS_Iterator.hxx>
#include <TopTools_MapOfShape.hxx>

void MapShapes(const TopoDS_Shape& aShape, TopTools_MapOfShape& aMap)
{
    // This is a recursive auxiliary method. It stores all subShapes of aShape in a Map.
    if (aMap.Add(aShape)) {
        // Add returns True if aShape was not already in the Map.
        TopoDS_Iterator It;
        for (It.Initialize(aShape); It.More(); It.Next()) {
            MapShapes(It.Value(), aMap);
        }
    }
}

Standard_Integer Size(const TopoDS_Shape& aShape)
{
    // Store Shapes in a Map and return the size.
    TopTools_MapOfShape M;
    MapShapes(aShape, M);
    return M.Extent();
}
```

**Note** For more details about Maps please, refer to the TCollection documentation. (Foundation Classes Reference Manual)

The following example is more ambitious and writes a program which copies a data structure using an `IndexedMap`. The copy is an identical structure but it shares nothing with the original. The principal algorithm is as follows:

- All Shapes in the structure are put into an `IndexedMap`. 

- A table of Shapes is created in parallel with the map to receive the copies.
- The structure is copied using the auxiliary recursive function, which copies from the map to the array.

```cpp
#include <TopoDS_Shape.hxx>
#include <TopoDS_Iterator.hxx>
#include <TopTools_IndexedMapOfShape.hxx>
#include <TopTools_Array1OfShape.hxx>
#include <TopoDS_Location.hxx>

TopoDS_Shape Copy(const TopoDS_Shape& aShape, const TopoDS_Builder& aBuilder) {
    // Copies the whole structure of aShape using aBuilder.
    // Stores all the sub-Shapes in an IndexedMap.
    TopTools_IndexedMapOfShape theMap;
    TopoDS_Iterator It;
    Standard_Integer i;
    TopoDS_Shape S;
    TopLoc_Location Identity;
    S = aShape;
    S.Location(Identity);
    S.Orientation(TopAbs_FORWARD);
    theMap.Add(S);
    for (i=1; i<= theMap.Extent(); i++) {
        for (It.Initialize(theMap(i)); It.More(); It.Next()) {
            S=It.Value();
            S.Location(Identity);
            S.Orientation(TopAbs_FORWARD);
            theMap.Add(S);
        }
    }
}
```

In the above example, the index $i$ is that of the first object not treated in
the Map. When $i$ reaches the same size as the Map this means that everything has been treated. The treatment consists in inserting in the Map all the sub-objects, if they are not yet in the Map, they are inserted with an index greater than $i$.

**Note** that the objects are inserted with a local reference set to the identity and a FORWARD orientation. Only the underlying TShape is of great interest.

```cpp
// Create an array to store the copies.
TopTools_Array1OfShapetheCopies(1,theMap.Extent());

// Use a recursive function to copy the first element.
void AuxiliaryCopy (Standard_Integer,
const TopTools_IndexedMapOfShape&,
TopTools_Array1OfShape&,
const TopoDS_Builder&);

AuxiliaryCopy(1,theMap,theCopies,aBuilder);

// Get the result with the correct local reference and orientation.
S = theCopies(1);
S.Location(aShape.Location());
S.Orientation(aShape.Orientation());
return S;
```

Below is the auxiliary function, which copies the element of rank $i$ from the map to the table. This method checks if the object has been copied; if not copied, then an empty copy is performed into the table and the copies of all the sub-elements are inserted by finding their rank in the map.

```cpp
void AuxiliaryCopy(Standard_Integer index,
const TopTools_IndexedMapOfShapes& sources,
TopTools_Array1OfShape& copies,
const TopoDS_Builder& aBuilder)
{
    // If the copy is a null Shape the copy is not done.
```
if (copies(index).IsNull()) {
    copies(index) = sources(index).EmptyCopied();
    //Insert copies of the sub-shapes.
    TopoDS_Iterator It;
    TopoDS_Shape S;
    TopLoc_Location Identity;
    for (It.Initialize(sources(index)), It.More(), It.Next () ) {
        S = It.Value();
        S.Location(Identity);
        S.Orientation(TopAbs_FORWARD);

        AuxiliaryCopy(sources.FindIndex(S), sources, copies,aBuilder);

        S.Location(It.Value().Location()); S.Orientation( It.Value().Orientation());
        aBuilder.Add(copies(index), S);
    }
}
A wire composed of 6 edges.

TopExp_Explorer, however, recuperates the lines in any order.

```
TopoDS_Wire W = ...;
BRepTools_WireExplorer Ex;
for(Ex.Init(W); Ex.More(); Ex.Next()) {
    ProcessTheCurrentEdge(Ex.Current());
    ProcessTheVertexConnectingTheCurrentEdgeToThePreviousOne(Ex.CurrentVertex());
}
```
Storage of shapes

*BRepTools* and *BinTools* packages contain methods *Read* and *Write* allowing to read and write a Shape to/from a stream or a file. The methods provided by *BRepTools* package use ASCII storage format; *BinTools* package uses binary format. Each of these methods has two arguments:

- a *TopoDS_Shape* object to be read/written;
- a stream object or a file name to read from/write to.

The following sample code reads a shape from ASCII file and writes it to a binary one:

```cpp
TopoDS_Shape aShape;
if (BRepTools::Read (aShape, "source_file.txt")) {
    BinTools::Write (aShape, "result_file.bin");
}
```

Generated on Wed Aug 30 2017 17:04:20 for Open CASCADE Technology by doxygen 1.8.13
# BRep Format

## Table of Contents

- Introduction
- Format Common Structure
- Locations
- Geometry
  - 3D curves
    - Line - <3D curve record 1>
    - Circle - <3D curve record 2>
    - Ellipse - <3D curve record 3>
    - Parabola - <3D curve record 4>
    - Hyperbola - <3D curve record 5>
    - Bezier Curve - <3D curve record 6>
    - B-spline Curve - <3D curve record 7>
    - Trimmed Curve - <3D curve record 8>
    - Offset Curve - <3D curve record 9>
  - Surfaces
    - Plane - <
surface record 1
  ✔ Cylinder - < surface record 2
  ✔ Cone - < surface record 3
  ✔ Sphere - < surface record 4
  ✔ Torus - < surface record 5
  ✔ Linear Extrusion - < surface record 6
  ✔ Revolution Surface - < surface record 7
  ✔ Bezier Surface - < surface record 8
  ✔ B-spline Surface - < surface record 9
  ✔ Rectangular Trim Surface - < surface record 10
  ✔ Offset Surface - < surface record 11

2D curves
  ✔ Line - <2D curve record 1>
  ✔ Circle - <2D curve record 2>
  ✔ Ellipse - <2D curve record 3>
  ✔ Parabola - <2D curve record 4>
Hyperbola - <2D curve record 5>
Bezier Curve - <2D curve record 6>
B-spline Curve - <2D curve record 7>
Trimmed Curve - <2D curve record 8>
Offset Curve - <2D curve record 9>
3D polygons
Triangulations
Polygons on triangulations
Geometric Sense of a Curve
Shapes
Common Terms
Vertex data
Edge data
Face data
Appendix
Introduction

BREP format is used to store 3D models and allows to store a model which consists of vertices, edges, wires, faces, shells, solids, compsolids, compounds, edge triangulations, face triangulations, polylines on triangulations, space location and orientation. Any set of such models may be stored as a single model which is a compound of the models.

The format is described in an order which is convenient for understanding rather than in the order the format parts follow each other. BNF-like definitions are used in this document. Most of the chapters contain BREP format descriptions in the following order:

- format file fragment to illustrate the part;
- BNF-like definition of the part;
- detailed description of the part.

Note that the format is a part of Open CASCADE Technology (OCCT).

Some data fields of the format have additional values, which are used in OCCT.

Some data fields of the format are specific for OCCT.
Format Common Structure

ASCII encoding is used to read/write BREP format from/to file. The format data are stored in a file as text data.

BREP format uses the following BNF terms:

- `<\n>`: It is the operating-system-dependent ASCII character sequence which separates ASCII text strings in the operating system used;
- `<\_\n>`: = " "*`<\n>`;
- `<_>`: = " "; It is a not empty sequence of space characters with ASCII code 21h;
- `<flag>`: = "0" | "1";
- `<int>`: It is an integer number from $-2^{31}$ to $2^{31}-1$ which is written in denary system;
- `<real>`: It is a real from $-1.7976931348623158 \cdot 10^{308}$ to $1.7976931348623158 \cdot 10^{308}$ which is written in decimal or E form with base 10. The point is used as a delimiter of the integer and fractional parts;
- `<2D point>`: = `<real>` `<_>` `<real>`;
- `<3D point>`: = `<real>` `<(_>` `<real)` $^2$;
- `<2D direction>`: It is a `<2D point>` $x$ $y$* so that $x^2 + y^2 = 1$;
- `<3D direction>`: It is a `<3D point>` $x$ $y$ $z$* so that $x^2 + y^2 + z^2 = 1$;
- `<+>`: It is an arithmetic operation of addition.

The format consists of the following sections:

- `<content type>`;
- `<version>`;
- `<locations>`;
- `<geometry>`;
- `<shapes>`.

`<content type>` = "DBRep_DrawableShape" `<\n>` `<\_\n>`; `<content type>` have other values [1].

`<version>` = ("CASCADE Topology V1, (c) Matra-Datavision" |
"CASCADE Topology V2, (c) Matra-Datavision")<\n>; The difference of the versions is described in the document.

Sections <locations>, <geometry> and <shapes> are described below in separate chapters of the document.
# Locations

**Example**

<table>
<thead>
<tr>
<th>Locations 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0 0 1 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
</tr>
<tr>
<td>4 0 1 0</td>
</tr>
<tr>
<td>5 0 0 1</td>
</tr>
<tr>
<td>2 1 1 2 1 0</td>
</tr>
</tbody>
</table>

**BNF-like Definition**

```plaintext
<locations> = <location header> <\n> <location records>;
<location header> = "Locations" <_> <location record count>;
<location record count> = <int>;
<location records> = <location record> ^ <location record count>;
<location record> = <location record 1> | <location record 2>;
<location record 1> = "1" <\n> <location data 1>;
<location record 2> = "2" <_> <location data 2>;
```
Description

<location data 1> is interpreted as a 3 x 4 matrix $(Q = \begin{pmatrix} q_{1,1} & q_{1,2} & q_{1,3} & q_{1,4} \\ q_{2,1} & q_{2,2} & q_{2,3} & q_{2,4} \\ q_{3,1} & q_{3,2} & q_{3,3} & q_{3,4} \end{pmatrix})$ which describes transformation of 3 dimensional space and satisfies the following constraints:

- $(d \neq 0)$ where $(d = |Q_{2}|)$ where $(Q_{2} = \begin{pmatrix} q_{1,1} & q_{1,2} & q_{1,3} & q_{1,4} \\ q_{2,1} & q_{2,2} & q_{2,3} & q_{2,4} \\ q_{3,1} & q_{3,2} & q_{3,3} & q_{3,4} \end{pmatrix})$
- $(Q_{3}^{T} = Q_{3}^{-1})$ where $(Q_{3} = Q_{2}/d^{1/3})$.

The transformation transforms a point $(x, y, z)$ to another point $(u, v, w)$ by the rule:

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = Q \cdot (x \; y \; z \; 1)^{T} = \begin{pmatrix} q_{1,1} \cdot x + q_{1,2} \cdot y + q_{1,3} \cdot z + q_{1,4} \\ q_{2,1} \cdot x + q_{2,2} \cdot y + q_{2,3} \cdot z + q_{2,4} \\ q_{3,1} \cdot x + q_{3,2} \cdot y + q_{3,3} \cdot z + q_{3,4} \end{pmatrix}.$$

$Q$ may be a composition of matrices for the following elementary transformations:

- parallel translation – $(D_{x}, D_{y}, D_{z})$ by an angle $(\varphi)$
- rotation around an axis with a direction $(D_{x}, D_{y}, D_{z})$ by an angle $(\varphi)$

$$\begin{pmatrix} D \cdot \{x\}^{2} \cdot \text{c} (1-\cos(\varphi)) + \cos(\varphi) \cdot \text{D} \cdot \{x\} \\ \text{D} \cdot \{y\} \cdot \text{c} (1-\cos(\varphi)) - \text{D} \cdot \{z\} \cdot \text{s}(\varphi) \cdot \text{D} \cdot \{x\} \\ \text{D} \cdot \{z\} \cdot \text{c} (1-\cos(\varphi)) + \text{D} \cdot \{y\} \cdot \text{s}(\varphi) \cdot \text{D} \cdot \{x\} \cdot \text{c}$$
\[ D_y \cdot (1 - \cos(\varphi)) + D_z \cdot \sin(\varphi) \]
\[ D_y^2 \cdot (1 - \cos(\varphi)) + \cos(\varphi) \]
\[ D_y \cdot D_z \cdot (1 - \cos(\varphi)) - D_x \cdot \sin(\varphi) \]
\[ D_y \cdot D_z \cdot (1 - \cos(\varphi)) + D_x \cdot \sin(\varphi) \]
\[ D_z^2 \cdot (1 - \cos(\varphi)) + \cos(\varphi) \]
\[ 0 \]
\[ \begin{pmatrix} s^0 & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \end{pmatrix} \]
where \( S \in (-\infty, \infty) \setminus \{0\}; \)
scaling – \( \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}; \)
central symmetry – \( \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}; \)
axis symmetry – \( \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}; \)
plane symmetry – \( \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}. \)
Geometry

<geometry> =
<2D curves>
<3D curves>
<3D polygons>
<polygons on triangulations>
<surfaces>
<triangulations>;
3D curves

Example

Curves 13
1 0 0 0 0 0 1
1 0 0 3 -0 1 0
1 0 2 0 0 0 1
1 0 0 0 -0 1 0
1 1 0 0 0 0 1
1 1 0 3 0 1 0
1 1 2 0 0 0 1
1 1 0 0 -0 1 0
1 0 0 0 1 0 -0
1 0 0 3 1 0 -0
1 0 2 0 1 0 -0
1 0 2 3 1 0 -0
1 1 0 0 1 0 0

BNF-like Definition

<3D curves> = <3D curve header> <\n> <3D curve records>;

<3D curve header> = "Curves" <_> <3D curve count >;

<3D curve count> = <int>;

<3D curve records> = <3D curve record> ^ <3D curve count>;

<3D curve record> =<3D curve record 1> | <3D curve record 2> | <3D curve record 3> |
Line - <3D curve record 1>

Example

```
1 1 0 3 0 1 0
```

BNF-like Definition

```
<3D curve record 1> = "1" <_> <3D point> <_> <3D direction> </_\n>;
```

Description

<3D curve record 1> describes a line. The line data consist of a 3D point P* and a 3D direction *D. The line passes through the point P, has the direction *D* and is defined by the following parametric equation:

```
\[ C(u)=P+u \cdot D, \; u \in (-\infty, \; \infty). \]
```

The example record is interpreted as a line which passes through a point P=(1, 0, 3), has a direction D=(0, 1, 0) and is defined by the following parametric equation: \( ( C(u)=(1,0,3)+u \cdot (0,1,0) ) \).

Circle - <3D curve record 2>

Example

```
2 1 2 3 0 0 1 1 0 -0 -0 1 0 4
```

BNF-like Definition
Description

<3D curve record 2> describes a circle. The circle data consist of a 3D point \( P \), pairwise orthogonal 3D directions \( N, D_{x} \) and \( D_{y} \) and a non-negative real \( *r* \). The circle has a center \( P \) and is located in a plane with a normal \( N \). The circle has a radius \( *r* \) and is defined by the following parametric equation:

\[
C(u) = P + r \cdot (\cos(u) \cdot D_{x} + \sin(u) \cdot D_{y}), \quad u \in [0, 2\pi).
\]

The example record is interpreted as a circle which has its center \( P=(1, 2, 3) \), is located in plane with a normal \( N=(0, 0, 1) \). Directions for the circle are \( D_{x}=(1, 0, 0) \) and \( D_{y}=(0, 1, 0) \). The circle has a radius \( r=4 \) and is defined by the following parametric equation: \(( C(u) = (1,2,3) + 4 \cdot (\cos(u) \cdot cdot(1,0,0) + \sin(u) \cdot cdot(0,1,0)) ) \).

Ellipse - <3D curve record 3>

Example

| 3 | 1 | 2 | 3 | 0 | 0 | 1 | 1 | 0 | -0 | -0 | 1 | 0 | 5 | 4 |

BNF-like Definition
Description

<3D curve record 3> describes an ellipse. The ellipse data consist of a 3D point \( P \), pairwise orthogonal 3D directions \( N \), \( D_{maj} \) and \( D_{min} \) and non-negative reals \( r_{maj} \) and \( r_{min} \) so that \( r_{min} \leq r_{maj} \). The ellipse has its center \( P \), is located in plane with the normal \( N \), has major and minor axis directions \( D_{maj} \) and \( D_{min} \), major and minor radii \( r_{maj} \) and \( r_{min} \) and is defined by the following parametric equation:

\[
C(u) = P + r_{maj} \cdot \cos(u) \cdot D_{maj} + r_{min} \cdot \sin(u) \cdot D_{min}, \quad u \in [0, 2\pi].
\]

The example record is interpreted as an ellipse which has its center \( P = (1, 2, 3) \), is located in plane with a normal \( N = (0, 0, 1) \), has major and minor axis directions \( D_{maj} = (1, 0, 0) \) and \( D_{min} = (0, 1, 0) \), major and minor radii \( r_{maj} = 5 \) and \( r_{min} = 4 \) and is defined by the following parametric equation: \( C(u) = (1, 2, 3) + 5 \cdot \cos(u) \cdot (1, 0, 0) + 4 \cdot \sin(u) \cdot (0, 1, 0) \).

**Parabola - <3D curve record 4>**
Example

| 4 | 1 | 2 | 3 | 0 | 0 | 1 | 1 | 0 | -0 | -0 | 1 | 0 | 16 |

BNF-like Definition

```plaintext
<3D curve record 4> = "4" <_> <3D parabola origin> <_> <3D parabola N> <_> <3D parabola Dx> <_> <3D parabola Dy> <_> <3D parabola focal length> 
<\n>;

<3D parabola origin> = <3D point>;
<3D parabola N> = <3D direction>;
<3D parabola Dx> = <3D direction>;
<3D parabola Dy> = <3D direction>;
<3D parabola focal length> = <real>;
```

Description

<3D curve record 4> describes a parabola. The parabola data consist of a 3D point \( P \), pairwise orthogonal 3D directions \( N \), \( D_x \) * and \( D_y \) * and a non-negative real \( f \). The parabola is located in plane which passes through the point \( P \) * and has the normal \( N \). The parabola has a focus length \( f \) * and is defined by the following parametric equation:

\[
C(u)=P+\frac{u^2}{4 \cdot f} \cdot D_x + u \cdot D_y, \quad u \in (-\infty, +\infty) \iff f \neq 0;
\]

\[
C(u)=P+u \cdot D_x, \quad u \in (-\infty, +\infty) \iff f = 0; \quad \text{(degenerated case).}
\]

The example record is interpreted as a parabola in plane which passes through a point \( P=(1, 2, 3) \) and has a normal \( N=(0, 0, 1) \). Directions for the parabola are \( D_x=(1, 0, 0) \) and \( D_y=(0, 1, 0) \). The parabola has a focus length \( f=16 \) and is defined by the following parametric equation: \( C(u) = \)
Hyperbola - <3D curve record 5>

Example

\[
\begin{array}{cccccccc}
5 & 1 & 2 & 3 & 0 & 0 & 1 & 1
\end{array}
\begin{array}{cccc}
0 & -0 & 1 & 0 & 5 & 4
\end{array}
\]

BNF-like Definition

<3D curve record 5> = "5" <_> <3D hyperbola origin> <_> <3D hyperbola N> <_> <3D hyperbola Dx> <_> <3D hyperbola Dy> <_> <3D hyperbola Kx> <_> <3D hyperbola Ky> <_>
;

<3D hyperbola origin> = <3D point>;

<3D hyperbola N> = <3D direction>;

<3D hyperbola Dx> = <3D direction>;

<3D hyperbola Dy> = <3D direction>;

<3D hyperbola Kx> = <real>;

<3D hyperbola Ky> = <real>;

Description

<3D curve record 5> describes a hyperbola. The hyperbola data consist of a 3D point \( P \), pairwise orthogonal 3D directions \( N, D_x \) and \( D_y \) and non-negative reals \( k_x \) and \( k_y \). The hyperbola is located in plane which passes through the point \( P \) and has the normal \( N \). The hyperbola is defined by the following parametric equation:

\[
[ C(u) = P + k_x \cdot \operatorname{cosh}(u) \cdot D_x + k_y \cdot \operatorname{sinh}(u) \cdot D_y, u \in (-\infty; \infty) ]
\]
The example record is interpreted as a hyperbola in plane which passes through a point \( P=(1, 2, 3) \) and has a normal \( N=(0, 0, 1) \). Other hyperbola data are \( D_x=(1, 0, 0), D_y=(0, 1, 0), k_x=5 \) and \( k_y=4 \). The hyperbola is defined by the following parametric equation: \( (C(u) = (1,2,3) + 5 \cdot \cosh(u) \cdot (1,0,0) + 4 \cdot \sinh(u) \cdot (0,1,0) ) \).

**Bezier Curve - <3D curve record 6>**

**Example**

\[
\begin{array}{cccccccc}
6 & 1 & 2 & 0 & 1 & 0 & 4 & 1 & -2 & 0 & 5 & 2 & 3 & 0 & 6
\end{array}
\]

**BNF-like Definition**

```plaintext
<3D curve record 6> = "6" <_> <3D Bezier rational flag> <_> <3D Bezier degree>
  <3D Bezier weight poles> <_>n;

  <3D Bezier rational flag> = <flag>;

  <3D Bezier degree> = <int>;

  3D Bezier weight poles> = (<_> <3D Bezier weight pole>) ^ (<3D Bezier degree> <+ "1");

  <3D Bezier weight pole> = <3D point> [<_> <real> ];
```

**Description**

<3D curve record 6> describes a Bezier curve. The curve data consist of a rational \( r \), a degree \( m \leq 25 \) and weight poles.

The weight poles are \( m+1 \) 3D points \( B_0 \ldots B_m \) *if the flag *r is 0. The weight poles are \( m+1 \) pairs \( *B_0h_0 \ldots B_mh_m \) if flag *r* is 1. Here \( *B_i \) is a 3D point and \( *h_i \) is a positive real \( (0 \leq h_i \leq m) \). \( h_{-i}=1 \); (0 \leq i \leq m) if the flag *r* is 0.
The Bezier curve is defined by the following parametric equation:

\[ C(u) = \frac{\sum_{i=0}^{m} B_{i} \cdot h_{i} \cdot C_{m}^{i} \cdot u^{i} \cdot (1-u)^{m-i}}{\sum_{i=0}^{m} h_{i} \cdot C_{m}^{i} \cdot u^{i} \cdot (1-u)^{m-i}}, \quad u \in [0, 1] \]

where \( 0^0 \equiv 1 \).

The example record is interpreted as a Bezier curve with a rational flag \( r=1 \), degree \( m=2 \) and weight poles \( B_0=(0, 1, 0) \), \( h_0=4 \), \( B_1=(1, -2, 0) \), \( h_1=5 \) and \( B_2=(2, 3, 0) \), \( h_2=6 \). The Bezier curve is defined by the following parametric equation:

\[ C(u) = \frac{(0,1,0) \cdot 4 \cdot (1-u)^2 + (1,-2,0) \cdot 5 \cdot 2 \cdot u \cdot (1-u) + (2,3,0) \cdot 6 \cdot u^2}{4 \cdot (1-u)^2 + 5 \cdot 2 \cdot u \cdot (1-u) + 6 \cdot u^2}. \]

**B-spline Curve - <3D curve record 7>**

**Example**

<table>
<thead>
<tr>
<th>7</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>-2</th>
<th>0</th>
<th>5</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0.25</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>0.75</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BNF-like Definition**

```plaintext
<3D curve record 7> = "7" <_> <3D B-spline rational flag> <_> "0" <_> <3D B-spline degree> <_
<3D B-spline pole count> <_> <3D B-spline multiplicity knot count> <3D B-spline weight poles
<_\n> <3D B-spline multiplicity knots> <_\n>;

<3D B-spline rational flag> = <flag>;

<3D B-spline degree> = <int>;

<3D B-spline pole count> = <int>;
```
Description

<3D curve record 7> describes a B-spline curve. The curve data consist of a rational flag \( r \), a degree \( \leq 25 \), pole count \( \geq 2 \), multiplicity knot count \( k \), weight poles and multiplicity knots.

The weight poles are \( n \) 3D points \( *B_1 ... B_n* \) if the flag \( *r* \) is 0. The weight poles are \( *n* \) pairs \( *B_1h_1 ... B_nh_n* \) if the flag \( *r* \) is 1. Here \( *B_i* \) is a 3D point and \( *h_i* \) is a positive real \((1 \leq i \leq n)\). \( h_{\{i\}}=1; (1 \leq i \leq n) \) if the flag \( *r* \) is 0.

The multiplicity knots are \( k \) pairs \( *u_1q_1 ... u_kq_k*. \) Here \( *u_i* \) is a knot with a multiplicity \( \leq m+1 \) so that

\[
\{ u_{\{i\}} < u_{\{i+1\}} (1 \leq i \leq k-1), \}
\{ q_{\{1\}} \leq m+1,; q_{\{k\}} \leq m+1,; q_{\{i\}} \leq m; (2 \leq i \leq k-1),
\sum_{i=1}^{k}q_{\{i\}}=m+n+1. \}
\]

The B-spline curve is defined by the following parametric equation:

\[
C(u) = \frac{\{ \sum_{i=1}^{n}B_{\{i\}} \cdot h_{\{i\}} \cdot N_{\{i,m+1\}}(u) \}}{\{ \sum_{i=1}^{n}h_{\{i\}} \cdot N_{\{i,m+1\}}(u) \}}, u \in [u_{\{1\}}, u_{\{k\}}]
\]

where functions \( \{ N_{\{i,j\}} \} \) have the following recursion definition by \( j \):
\[ N_{i,1}(u) = \begin{cases} 1 & \text{if} \; \bar{u}_{i} \leq u \leq \bar{u}_{i+1} \\ 0 & \text{if} \; u < \bar{u}_{i} \lor \bar{u}_{i+1} \leq u \end{cases} \]
\[ N_{i,j}(u) = \frac{(u-\bar{u}_{i}) \cdot N_{i,j-1}(u)}{\bar{u}_{i+j-1}-\bar{u}_{i}} + \frac{(\bar{u}_{i+j}-u) \cdot N_{i+1,j-1}(u)}{\bar{u}_{i+j}-\bar{u}_{i+1}}, \quad (2 \leq j \leq m+1) \]

where
\[ \bar{u}_{i} = u_{j}, \quad (1 \leq j \leq k, \sum_{l=1}^{j-1}q_{l}+1 \leq i \leq \sum_{l=1}^{j}q_{l}). \]

The example record is interpreted as a B-spline curve with a rational flag \( r = 1 \), a degree \( m = 1 \), pole count \( n = 3 \), multiplicity knot count \( k = 5 \), weight poles \( B_{1} = (0,1,0), \; h_{1} = 4, \; B_{2} = (1,-2,0), \; h_{2} = 5 \) and \( B_{3} = (2,3,0), \; h_{3} = 6 \), multiplicity knots \( u_{1} = 0, \; q_{1} = 1, \; u_{2} = 0.25, \; q_{2} = 1, \; u_{3} = 0.5, \; q_{3} = 1, \; u_{4} = 0.75, \; q_{4} = 1 \) and \( u_{5} = 1, \; q_{5} = 1 \). The B-spline curve is defined by the following parametric equation:
\[ C(u) = \frac{(0,1,0) \cdot 4 \cdot N_{1,2}(u) + (1,-2,0) \cdot 5 \cdot N_{2,2}(u) + (2,3,0) \cdot 6 \cdot N_{3,2}(u)}{4 \cdot N_{1,2}(u) + 5 \cdot N_{2,2}(u) + 6 \cdot N_{3,2}(u)}. \]

**Trimmed Curve - <3D curve record 8>**

**Example**

<table>
<thead>
<tr>
<th>8</th>
<th>-4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**BNF-like Definition**

```
<3D curve record 8> = "8" <__> <3D trimmed curve u min> <__> <3D trimmed curve u max> <__> <3D curve record>;

<3D trimmed curve u min> = <real>;

<3D trimmed curve u max> = <real>;
```
<3D curve record 8> describes a trimmed curve. The trimmed curve data consist of reals \( u_{min} \) and \( u_{max} \) and <3D curve record> so that \( u_{min} < u_{max} \). The trimmed curve is a restriction of the base curve *B* described in the record to the segment \( [u_{min}, u_{max}] \subseteq \text{domain}(B) \). The trimmed curve is defined by the following parametric equation:

\[
C(u) = B(u), \quad u \in [u_{min}, u_{max}].
\]

The example record is interpreted as a trimmed curve with \( u_{min} = -4 \) and \( u_{max} = 5 \) for the base curve \( B(u) = (1,2,3) + u \cdot (1,0,0) \). The trimmed curve is defined by the following parametric equation: \( C(u) = (1,2,3) + u \cdot (1,0,0), \quad u \in [-4, 5] \).

**Offset Curve - <3D curve record 9>**

**Example**

<table>
<thead>
<tr>
<th>9 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0</td>
</tr>
<tr>
<td>1 1 2 3 1 0 0</td>
</tr>
</tbody>
</table>

**BNF-like Definition**

```
<3D curve record 9> = "9" <_<3D offset curve distance> <<_\n>;<3D offset curve direction> <<_\n>;<3D curve record>;
<3D offset curve distance> = <real>;
<3D offset curve direction> = <3D direction>;
```

**Description**

<3D curve record 9> describes an offset curve. The offset curve data consist of a distance \( d \), a 3D direction \( D \) and a <3D curve record>. The
offset curve is the result of offsetting the base curve *B* described in the record to the distance *d* along the vector \([B'(u), D]\ \neq \vec{0} \). The offset curve is defined by the following parametric equation:

\[
C(u) = B(u) + d \cdot \frac{[B'(u), D]}{|[B'(u), D]|}, \quad u \in \text{domain}(B).
\]

The example record is interpreted as an offset curve with a distance \(d=2\), direction \(D=(0, 1, 0)\), base curve \(B(u)=(1,2,3)+u \cdot (1,0,0)\) and defined by the following parametric equation: \(C(u)=(1,2,3)+u \cdot (1,0,0)+2 \cdot (0,0,1)\).
Surfaces

Example

```
Surfaces 6
  1 0 0 0 1 0 -0 0 0 1 0 -1 0
  1 0 0 0 -0 1 0 0 0 1 1 0 -0
  1 0 0 3 0 0 1 1 0 -0 -0 1 0
  1 0 2 0 -0 1 0 0 0 1 1 0 -0
  1 0 0 0 0 0 1 1 0 -0 -0 1 0
  1 1 0 0 1 0 -0 0 0 1 0 -1 0
```

BNF-like Definition

```
<surfaces> = <surface header> <\n> <surface records>;

<surface header> = "Surfaces" <_> <surface count>;

<surface records> = <surface record> ^ <surface count>;

<surface record> = <surface record 1> | <surface record 2> | <surface record 3> | <surface record 4> | <surface record 5> | <surface record 6> | <surface record 7> | <surface record 8> | <surface record 9> | <surface record 10> | <surface record 11>;
```
Plane - < surface record 1 >

Example

```
1 0 0 3 0 0 1 1 0 -0 -0 1 0
```

BNF-like Definition

```
<surface record 1> = "1" <_> <3D point> (<_> <3D direction>) ^ 3 <_
```

Description

<surface record 1> describes a plane. The plane data consist of a 3D point \( P^* \) and pairwise orthogonal 3D directions \(*N, D_u^*, and \*D_v\). The plane passes through the point \( P \), has the normal \(*N^* \) and is defined by the following parametric equation:

\[
S(u,v)=P+u \cdot D_u^*+v \cdot D_v^*, \quad (u,v) \in (-\infty, \infty) \times (-\infty, \infty).
\]

The example record is interpreted as a plane which passes through a point \( P=(0, 0, 3) \), has a normal \( N=(0, 0, 1) \) and is defined by the following parametric equation: \( S(u,v)=(0,0,3)+u \cdot (1,0,0) + v \cdot (0,1,0) \).

Cylinder - < surface record 2 >

Example

```
2 1 2 3 0 0 1 1 0 -0 -0 1 0 4
```

BNF-like Definition

```
<surface record 2> = "2" <_> <3D point> (<_> <3D direction>) ^ 3 <_> <real> <_
```

Description
<surface record 2> describes a cylinder. The cylinder data consist of a 3D point \( P \), pairwise orthogonal 3D directions \( D_v \), \( D_X \) and \( D_Y \) and a non-negative real \( *r* \). The cylinder axis passes through the point \( P* \) and has the direction \( *D_v* \). The cylinder has the radius \( *r* \) and is defined by the following parametric equation:

\[
S(u,v)=P+r \cdot (\cos(u) \cdot D_x+\sin(u) \cdot D_y)+v \cdot D_v, \quad (u,v) \in \mathbb{R}^2
\]

The example record is interpreted as a cylinder which axis passes through a point \( P=(1,2,3) \) and has a direction \( D_v=(0,0,1) \). Directions for the cylinder are \( D_X=(1,0,0) \) and \( D_Y=(0,1,0) \). The cylinder has a radius \( r=4 \) and is defined by the following parametric equation:

\[
S(u,v) = (1,2,3)+4 \cdot (\cos(u) \cdot D_X+\sin(u) \cdot D_Y) + v \cdot D_v.
\]

**Cone - < surface record 3 >**

**Example**

```
3 1 2 3 0 0 1 1 0 -0 -0 1 0 4
0.75
```

**BNF-like Definition**

```
<surface record 3> = "3" <_> <3D point> (<_> <3D direction>) ^ 3 (<_> <real>) ^ 2 <_/n>;
```

**Description**

<surface record 3> describes a cone. The cone data consist of a 3D point \( P \), pairwise orthogonal 3D directions \( D_Z \), \( D_X \) and \( D_Y \), a non-negative real \( *r* \) and a real \( \varphi \) \in (-\pi/2, \pi/2) \setminus \{0\}. The cone axis passes through the point \( P* \) and has the direction \( *D_Z* \). The plane which passes through the point \( P* \) and is parallel to directions \( *D_X* \) and \( *D_Y* \) is the cone referenced plane. The cone section by the plane is a circle with the radius \( *r* \). The direction from the point \( *P* \) to the cone apex is \( -\text{sgn}(\varphi) \cdot D_Z \). The cone has a half-angle \( |\varphi| \) and is
defined by the following parametric equation:

\[
S(u,v)=P+(r+v \cdot \sin(\varphi)) \cdot (\cos(u) \cdot D_{x}+\sin(u) \cdot D_{y})+v \cdot \cos(\varphi) \cdot D_{z}, \ (u,v) \in \[0, \ 2 \cdot \pi) \times (-\infty, \ \infty).
\]

The example record is interpreted as a cone with an axis which passes through a point \(P=(1, \ 2, \ 3)\) and has a direction \(D_{z}=(0, \ 0, \ 1)\). Other cone data are \(D_{x}=(1, \ 0, \ 0), \ D_{y}=(0, \ 1, \ 0), \ r=4\) and \(\varphi = 0.75\). The cone is defined by the following parametric equation:

\[
S(u,v)=(1,2,3)+(4 + v \cdot \sin(0.75)) \cdot (\cos(u) \cdot (1,0,0) + \sin(u) \cdot (0,1,0)) + v \cdot \cos(0.75) \cdot (0,0,1).
\]

**Sphere - < surface record 4 >**

**Example**

```
4 1 2 3 0 0 1 1 0 -0 -0 1 0 4
```

**BNF-like Definition**

```
<surface record 4> = "4" <[]> <3D point> (<> <3D direction>) ^ 3 <> <real> <_
>;
```

**Description**

<surface record 4> describes a sphere. The sphere data consist of a 3D point \(P\), pairwise orthogonal 3D directions \(D_{z}, \ D_{x}\) and \(D_{y}\) *and a non-negative real* \(*r\). The sphere has the center \(P\), radius \(*r*\) and is defined by the following parametric equation:

\[
S(u,v)=P+r \cdot \cos(v) \cdot (\cos(u) \cdot D_{x}+\sin(u) \cdot D_{y}) +r \cdot \sin(v) \cdot D_{z}, \ (u,v) \in \[0, \ 2 \cdot \pi) \times [-\pi/2, \ \pi/2). \]

The example record is interpreted as a sphere with its center \(P=(1, \ 2, \ 3)\). Directions for the sphere are \(D_{z}=(0, \ 0, \ 1), \ D_{x}=(1, \ 0, \ 0)\) and \(D_{y}=(0, \ 1, \ 0)\). The sphere has a radius \(r=4\) and is defined by the following parametric equation:
\[ S(u,v)=(1,2,3)+ 4 \cdot \cos(v) \cdot \cos(u) \cdot (0,1,0) + 4 \cdot \sin(v) \cdot (0,0,1). \]

**Torus - < surface record 5 >**

**Example**

| 5 | 1 | 2 | 3 | 0 | 0 | 1 | 1 | 0 | -0 | -0 | 1 | 0 | 8 | 4 |

**BNF-like Definition**

<surface record 5> = "5" <_> <3D point> (<> <3D direction>) ^ 3 (<> <real>) ^ 2 <_[n>;

**Description**

<surface record 5> describes a torus. The torus data consist of a 3D point \( P \), pairwise orthogonal 3D directions \( D_Z, D_X \) and \( D_Y \), and non-negative reals \( *r_1, *r_2 \). The torus axis passes through the point \( P \) and has the direction \( *D_Z \). \( r_1 \) is the distance from the torus circle center to the axis. The torus circle has the radius \( *r_2 \). The torus is defined by the following parametric equation:

\[ S(u,v)=P+(r_{1}+r_{2}\cdot\cos(v)) \cdot (\cos(u) \cdot D_{x}+\sin(u) \cdot D_{y})+r_{2}\cdot\sin(v) \cdot D_{z}, \quad (u,v) \in [0,2\cdot\pi) \times [0,2\cdot\pi). \]

The example record is interpreted as a torus with an axis which passes through a point \( P=(1, 2, 3) \) and has a direction \( D_Z=(0, 0, 1) \). \( D_X=(1, 0, 0) \), \( D_Y=(0, 1, 0) \), \( r_1=8 \) and \( r_2=4 \) for the torus. The torus is defined by the following parametric equation:

\[ S(u,v)=(1,2,3)+ (8+4 \cdot \cos(v)) \cdot \cos(u) \cdot (1,0,0) + (8+4 \cdot \sin(v)) \cdot (0,0,1). \]

**Linear Extrusion - < surface record 6 >**

**Example**
Description

&lt;surface record 6&gt; describes a linear extrusion surface. The surface data consist of a 3D direction \( D_v \) and a &lt;3D curve record&gt;. The linear extrusion surface has the direction \(*D_v\), the base curve \(*C*\) described in the record and is defined by the following parametric equation:

\[
S(u,v)=C(u)+v \cdot D_v, \quad (u,v) \in \text{domain}(C) \times (-\infty, \infty).
\]

The example record is interpreted as a linear extrusion surface with a direction \( D_v=(0, 0.6, 0.8) \). The base curve is a circle for the surface. The surface is defined by the following parametric equation:

\[
S(u,v)=(1,2,3)+4 \cdot (\cos(u) \cdot (1,0,0)+\sin(u) \cdot (0,1,0))+v \cdot (0, 0.6, 0.8), \quad (u,v) \in [0, 2 \cdot \pi) \times (-\infty, \infty).
\]

Revolution Surface - &lt; surface record 7 &gt;

Example

\[
\begin{array}{ccccccccc}
7 & -4 & 0 & 3 & 0 & 1 & 0 & -0 & -0 & 1 & 0 & 4 \\
2 & 1 & 2 & 3 & 0 & 0 & 1 & 1 & 0 & -0 & -0 & 1 & 0 & 4
\end{array}
\]

BNF-like Definition

\[
&lt;\text{surface record 7}&gt; = "7" \_\_ &lt;\text{3D point}&gt; \_\_ &lt;\text{3D direction}&gt; \_\_\_\n&lt;\text{3D curve record}&gt;;
\]

Description
<surface record 7> describes a revolution surface. The surface data consist of a 3D point \( P \), a 3D direction \( D^* \) and a <3D curve record>. The surface axis passes through the point \( *P* \) and has the direction \( *D* \). The base curve \( *C* \) described by the record and the axis are coplanar. The surface is defined by the following parametric equation:

\[
S(u,v) = P + V_{D}(v) + \cos(u) \cdot (V(v) - V_{D}(v)) + \sin(u) \cdot [D, V(v)], \quad (u,v) \in [0, 2 \cdot \pi) \times \text{domain}(C)
\]

where \( V(v) = C(v) - P \), \( V_{D}(v) = (D, V(v)) \cdot D \).

The example record is interpreted as a revolution surface with an axis which passes through a point \( P = (-4, 0, 3) \) and has a direction \( D = (0, 1, 0) \). The base curve is a circle for the surface. The surface is defined by the following parametric equation:

\[
S(u,v) = (-4,0,3) + V_{D}(v) + \cos(u) \cdot (V(v) - V_{D}(v)) + \sin(u) \cdot [(0,1,0), V(v)], (u,v) \in [0, \pi) \times [0, \pi)
\]

where \( V(v) = (5,2,0) + 4 \cdot (\cos(v) \cdot (1,0,0) + \sin(v) \cdot (0,1,0)) \), \( V_{D}(v) = ((0,1,0), V(v)) \cdot (0,1,0) \).

**Bezier Surface - < surface record 8 >**

Example

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-2</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BNF-like Definition**

```
<surface record 8> = "8" <_> <Bezier surface u rational flag> <_> <Bezier surface v rational flag> <_> <Bezier surface u degree> <_> <Bezier surface v degree> <_>
<Bezier surface weight poles>;

<Bezier surface u rational flag> = <flag>;
```
Description

<surface record 8> describes a Bezier surface. The surface data consist of a u rational flag $r_u$, v rational flag $r_v$, u degree $(m_u \leq 25)$, v degree $(m_v \leq 25)$ and weight poles.

The weight poles are \((m_u+1) \cdot (m_v+1)\) 3D points \((B_{i,j})\) \((i,j) \in \left\{0,...,m_u\right\} \times \left\{0,...,m_v\right\}\) if \((r_u+r_v=0)\). The weight poles are \((m_u+1) \cdot (m_v+1)\) pairs \((B_{i,j},h_{i,j})\) \((i,j) \in \left\{0,...,m_u\right\} \times \left\{0,...,m_v\right\}\) if \((r_u+r_v \neq 0)\). Here \((B_{i,j})\) is a 3D point and \((h_{i,j})\) is a positive real \((h_{i,j} = 1)\) \((i,j) \in \left\{0,...,m_u\right\} \times \left\{0,...,m_v\right\}\).

The Bezier surface is defined by the following parametric equation:

\[
S(u,v)=\frac{\sum_{i=0}^{m_u} \sum_{j=0}^{m_v} B_{i,j} \cdot h_{i,j} \cdot \frac{C_{m_u}^{i}}{i!} \cdot \frac{C_{m_v}^{j}}{j!} \cdot u^i \cdot (1-u)^{m_u-i} \cdot v^j \cdot (1-v)^{m_v-j} \cdot \sum_{i=0}^{m_u} \sum_{j=0}^{m_v}}{\sum_{i=0}^{m_u} \sum_{j=0}^{m_v} \sum_{i=0}^{m_u} \sum_{j=0}^{m_v}}
\]

\[
S(u,v)=\frac{\sum_{i=0}^{m_u} \sum_{j=0}^{m_v} B_{i,j} \cdot h_{i,j} \cdot \frac{C_{m_u}^{i}}{i!} \cdot \frac{C_{m_v}^{j}}{j!} \cdot u^i \cdot (1-u)^{m_u-i} \cdot v^j \cdot (1-v)^{m_v-j} \cdot \sum_{i=0}^{m_u} \sum_{j=0}^{m_v}}{\sum_{i=0}^{m_u} \sum_{j=0}^{m_v} \sum_{i=0}^{m_u} \sum_{j=0}^{m_v}}
\]
where \((0^0) \equiv 1\).

The example record is interpreted as a Bezier surface with a \(u\) rational flag \(r_u=1\), \(v\) rational flag \(r_v=1\), \(u\) degree \(m_u=2\), \(v\) degree \(m_v=1\), weight poles \(B_{0,0}=(0, 0, 1)\), \(B_{0,1}=(1, 0, -4)\), \(B_{1,0}=(0, 1, -2)\), \(h_{0,0}=8\), \(B_{1,1}=(1, 1, 5)\), \(h_{1,1}=11\), \(B_{2,0}=(0, 2, 3)\), \(h_{2,0}=9\) and \(B_{2,1}=(1, 2, 6)\), \(h_{2,1}=12\).

The surface is defined by the following parametric equation:

\[
S(u,v) = \frac{(0,0,1) \cdot 7 \cdot (1-u)^2 \cdot (1-v) + (1,0,-4) \cdot 10 \cdot (1-u)^2 \cdot v + (0,1,-2) \cdot 8 \cdot 2 \cdot u \cdot (1-u) \cdot (1-v) + (1,1,5) \cdot 11 \cdot 2 \cdot u \cdot (1-u) \cdot v + (0,2,3) \cdot 9 \cdot u^2 \cdot (1-v) + (1,2,6) \cdot 12 \cdot u^2 \cdot v \cdot \text{div} [7 \cdot (1-u)^2 \cdot (1-v) + 10 \cdot (1-u)^2 \cdot v + 8 \cdot 2 \cdot u \cdot (1-u) \cdot (1-v) + 11 \cdot 2 \cdot u \cdot (1-u) \cdot v + 9 \cdot u^2 \cdot (1-v) + 12 \cdot u^2 \cdot v]}{[7 \cdot (1-u)^2 \cdot (1-v) + 10 \cdot (1-u)^2 \cdot v + 8 \cdot 2 \cdot u \cdot (1-u) \cdot (1-v) + 11 \cdot 2 \cdot u \cdot (1-u) \cdot v + 9 \cdot u^2 \cdot (1-v) + 12 \cdot u^2 \cdot v]}
\]

**B-spline Surface - < surface record 9 >**

**Example**

```
<table>
<thead>
<tr>
<th>9</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>5</th>
<th>4</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>7</th>
<th>1</th>
<th>0</th>
<th>-4</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-2</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>0.75</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0.3</td>
<td>1</td>
</tr>
<tr>
<td>0.7</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
BNF-like Definition

```plaintext
<surface record 9> = "9" <_> <B-spline surface u rational flag> <_> 
   <B-spline surface v rational flag> <_> "0" <_> "0" <_> <B-spline surface u degree> <_> 
   <B-spline surface v degree> <_> <B-spline surface u pole count> <_> 
   <B-spline surface v pole count> <_> <B-spline surface u multiplicity knot count> <_> 
   <B-spline surface v multiplicity knot count> <_> <B-spline surface weight poles> <_> 
   <B-spline surface u multiplicity knots> <_> <B-spline surface v multiplicity knots>; 

   <B-spline surface u rational flag> = <flag>; 
   <B-spline surface v rational flag> = <flag>; 
   <B-spline surface u degree> = <int>; 
   <B-spline surface v degree> = <int>; 
   <B-spline surface u pole count> = <int>; 
   <B-spline surface v pole count> = <int>; 
   <B-spline surface u multiplicity knot count> = <int>; 
   <B-spline surface v multiplicity knot count> = <int>; 
   <B-spline surface weight poles> = (<B-spline surface weight pole group> <_> 
   (B-spline surface weight pole group) <_> 
   ^ <B-spline surface u pole count>); 
```

<B-spline surface weight pole group> = 
  (<B-spline surface weight pole> <_) ^ <B-spline surface v pole pole count>;

<B-spline surface weight pole> = <3D point> [<_<real>];

<B-spline surface u multiplicity knots> = 
  (<B-spline surface u multiplicity knot> <\n>) ^ <B-spline surface u multiplicity knot count>;

<B-spline surface u multiplicity knot> = <real> <_> <int>;

<B-spline surface v multiplicity knots> = 
  (<B-spline surface v multiplicity knot> <\n>) ^ <B-spline surface v multiplicity knot count>;

<B-spline surface v multiplicity knot> = <real> <_> <int>;

Description

<surface record 9> describes a B-spline surface. The surface data consist of a u rational flag \( r_u \), v rational flag \( r_v \), u degree \( m_u \leq 25 \), v degree \( m_v \leq 25 \), u pole count \( n_u \geq 2 \), v pole count \( n_v \geq 2 \), u multiplicity knot count \( k_u \), v multiplicity knot count \( k_v \), weight poles, u multiplicity knots, v multiplicity knots.

The weight poles are \(( n_u \cdot n_v \) \) 3D points \(( B_{i,j} \)\); \((i,j) \in \left\{ 1,\ldots,n_u \right\} \times \left\{ 1,\ldots,n_v \right\} \) if \( r_u + r_v \neq 0 \). The weight poles are \(( n_u \cdot n_v \) \) pairs \(( B_{i,j}h_{i,j} \); \((i,j) \in \left\{ 1,\ldots,n_u \right\} \times \left\{ 1,\ldots,n_v \right\} \) if \( r_u + r_v = 0 \). Here \(( B_{i,j} \)\) is a 3D point and \(( h_{i,j} \)\) is a positive real \((h_{i,j} = 1; \((i,j) \in \left\{ 1,\ldots,n_u \right\} \times \left\{ 1,\ldots,n_v \right\} \) \) if \( r_u + r_v = 0 \).
The $u$ multiplicity knots are $k_u$ pairs \( u_{\{1\}}q_{\{1\}} \ldots u_{\{k_u\}}q_{\{k_u\}} \). Here \( u_{\{i\}} \) is a knot with multiplicity \( q_{\{i\}} \geq 1 \) so that

\[
\begin{align*}
u_{\{i\}} < u_{\{i+1\}} ; (1 \lt i \leq k_u - 1), \quad & q_{\{1\}} \leq m_{\{u\}} + 1, \quad q_{\{k_u\}} \leq m_{\{u\}} + 1, \quad \sum_{i=1}^{k_u} q_{\{i\}} = m_{\{u\}} + n_{\{u\}} + 1. \\
\end{align*}
\]

The $v$ multiplicity knots are $k_v$ pairs \( v_{\{1\}}t_{\{1\}} \ldots v_{\{k_v\}}t_{\{k_v\}} \). Here \( v_{\{j\}} \) is a knot with multiplicity \( t_{\{j\}} \geq 1 \) so that

\[
\begin{align*}
v_{\{j\}} < v_{\{j+1\}} ; (1 \lt j \leq k_v - 1), \quad & t_{\{1\}} \leq m_{\{v\}} + 1, \quad t_{\{k_v\}} \leq m_{\{v\}} + 1, \quad \sum_{j=1}^{k_v} t_{\{j\}} = m_{\{v\}} + n_{\{v\}} + 1. \\
\end{align*}
\]

The B-spline surface is defined by the following parametric equation:

\[
S(u,v) = \frac{\sum_{i=1}^{n_u} \sum_{j=1}^{n_v} B_{i,j} \cdot h_{i,j} \cdot N_{i,m_{u}+1}(u) \cdot M_{j,m_{v}+1}(v)}{\sum_{i=1}^{n_u} \sum_{j=1}^{n_v} h_{i,j} \cdot N_{i,m_{u}+1}(u) \cdot M_{j,m_{v}+1}(v)}, \quad (u,v) \in [u_{\{1\}},u_{\{k_u\}}] \times [v_{\{1\}},v_{\{k_v\}}]
\]

where functions $N_{i,j}$ and $M_{i,j}$ have the following recursion definition by $j$:

\[
\begin{align*}
N_{i,1}(u) &= \left\{ \begin{array}{ll} 1 & \text{if } \bar{u}_{i} \leq u \leq \bar{u}_{i+1} \\ 0 & \text{otherwise} \end{array} \right. \\
N_{i,j}(u) &= \frac{(u - \bar{u}_{i}) \cdot N_{i,j-1}(u)}{\bar{u}_{i+j-1} - \bar{u}_{i}} + \frac{(\bar{u}_{i+j} - u) \cdot N_{i+1,j-1}(u)}{\bar{u}_{i+j} - \bar{u}_{i+1}}, \quad (2 \leq j \leq m_{u} + 1), \\
M_{i,1}(v) &= \left\{ \begin{array}{ll} 1 & \text{if } \bar{v}_{i} \leq v \leq \bar{v}_{i+1} \\ 0 & \text{otherwise} \end{array} \right. \\
M_{i,j}(v) &= \frac{(v - \bar{v}_{i}) \cdot M_{i,j-1}(v)}{\bar{v}_{i+j-1} - \bar{v}_{i}} + \frac{(\bar{v}_{i+j} - v) \cdot M_{i+1,j-1}(v)}{\bar{v}_{i+j} - \bar{v}_{i+1}}, \quad (2 \leq j \leq m_{v} + 1);
\end{align*}
\]

where

\[
\begin{align*}
\bar{u}_{i} &= u_{j} \quad (1 \leq j \leq k_{u} - 1), \quad \sum_{i=1}^{j-1} q_{\{i\}} \leq i \leq q_{\{j\}}, \quad \sum_{i=1}^{j-1} q_{\{i\}} \leq \sum_{i=1}^{j} q_{\{i\}}, \\
\bar{v}_{i} &= v_{j} \quad (1 \leq j \leq k_{v} - 1), \quad \sum_{i=1}^{j-1} t_{\{i\}} \leq j \leq t_{\{j\}}.
\end{align*}
\]
1\}t_{l}\leq i \leq \sum_{l=1}^{j}t_{l});

The example record is interpreted as a B-spline surface with a u rational flag \( r_u=1 \), v rational flag \( r_v=1 \), u degree \( m_u=1 \), v degree \( m_v=1 \), u pole count \( n_u=3 \), v pole count \( n_v=2 \), u multiplicity knot count \( k_u=5 \), v multiplicity knot count \( k_v=4 \), weight poles \( B_{1,1}=(0, 0, 1) \), \( h_{1,1}=7 \), \( B_{1,2}=(1, 0, -4) \), \( h_{1,2}=10 \), \( B_{2,1}=(0, 1, -2) \), \( h_{2,1}=8 \), \( B_{2,2}=(1, 1, 5) \), \( h_{2,2}=11 \), \( B_{3,1}=(0, 2, 3) \), \( h_{3,1}=9 \) and \( B_{3,2}=(1, 2, 6) \), \( h_{3,2}=12 \), u multiplicity knots \( u_1=0 \), \( q_1=1 \), \( u_2=0.25 \), \( q_2=1 \), \( u_3=0.5 \), \( q_3=1 \), \( u_4=0.75 \), \( q_4=1 \) and \( u_5=1 \), \( q_5=1 \), v multiplicity knots \( v_1=0 \), \( r_1=1 \), \( v_2=0.3 \), \( r_2=1 \), \( v_3=0.7 \), \( r_3=1 \) and \( v_4=1 \), \( r_4=1 \). The B-spline surface is defined by the following parametric equation:

\[
\begin{align}
S(u,v)= & \frac{(0,0,1) \cdot 7 \cdot N_{1,2}(u) \cdot M_{1,2}(v)+(1,0,-4) \cdot 10 \cdot N_{1,2}(u) \cdot M_{2,2}(v)+ (0,1,-2) \cdot 8 \cdot N_{2,2}(u) \cdot M_{1,2}(v)+(1,1,5) \cdot 11 \cdot N_{2,2}(u) \cdot M_{2,2}(v)+ (0,2,3) \cdot 9 \cdot N_{3,2}(u) \cdot M_{1,2}(v)+(1,2,6) \cdot 12 \cdot N_{3,2}(u) \cdot M_{2,2}(v))}{7 \cdot N_{1,2}(u) \cdot M_{1,2}(v)+10 \cdot N_{1,2}(u) \cdot M_{2,2}(v)+8 \cdot N_{2,2}(u) \cdot M_{1,2}(v)+9 \cdot N_{3,2}(u) \cdot M_{1,2}(v)+11 \cdot N_{2,2}(u) \cdot M_{2,2}(v)+12 \cdot N_{3,2}(u) \cdot M_{2,2}(v)}
\end{align}
\]

Rectangular Trim Surface - \(<\text{surface record 10}>\)

Example

\[
\begin{array}{ccccccccccc}
10 & -1 & 2 & -3 & 4 \\
1 & 1 & 2 & 3 & 0 & 0 & 1 & 1 & 0 & -0 & -0 & 1 & 0
\end{array}
\]

BNF-like Definition

\[
<\text{surface record 10}> = "10" \_<\_> <\text{trim surface u min}> \_<\_> <\text{trim surface u max}> \_<\_> <\text{trim surface v min}> \_<\_> <\text{trim surface v max}> \_<\_> <\text{surface record}>;
\]

\[
<\text{trim surface u min}> = <\text{real}>;
\]

\[
<\text{trim surface u max}> = <\text{real}>;
\]
Description

<surface record 10> describes a rectangular trim surface. The surface data consist of reals \( u_{\text{min}}, u_{\text{max}}, v_{\text{min}} * \text{ and } v_{\text{max}} * \text{ and a } <\text{surface record}> \text{ so that } u_{\text{min}} < u_{\text{max}} * \text{ and } v_{\text{min}} < v_{\text{max}}. \) The rectangular trim surface is a restriction of the base surface \( *B* \) described in the record to the set \([u_{\text{min}}, u_{\text{max}}] \times [v_{\text{min}}, v_{\text{max}}] \) \text{subseteq} \text{domain}(B) \). The rectangular trim surface is defined by the following parametric equation:

\[
S(u,v)=B(u,v), \quad (u,v) \in [u_{\text{min}}, u_{\text{max}}] \times [v_{\text{min}}, v_{\text{max}}].
\]

The example record is interpreted as a rectangular trim surface to the set \([-1, 2] \times [-3, 4]\) for the base surface \( B(u,v)=(1,2,3)+u \cdot (1,0,0)+v \cdot (0,1,0) \). The rectangular trim surface is defined by the following parametric equation: \( B(u,v)=(1,2,3)+u \cdot (1,0,0)+ v \cdot (0,1,0), \) \( (u,v) \in [-1,2] \times [-3,4] \).

Offset Surface - < surface record 11 >

Example

\[
\begin{array}{cccccccccccc}
11 & -2 \\
1 & 1 & 2 & 3 & 0 & 0 & 1 & 1 & 0 & -0 & -0 & 1 & 0
\end{array}
\]

BNF-like Definition

\[
<\text{surface record 11}> = "11" \ <_> \ <\text{surface record distance}> \ <\_\n> \ <\text{surface record}>;
\]

\[
<\text{surface record distance}> = <\text{real}>;
\]

Description

<surface record 11> describes an offset surface. The offset surface data
consist of a distance \( d^* \) and a \(<\text{surface record}>\). The offset surface is the result of offsetting the base surface \( *B* \) described in the record to the distance \( *d*^* \) along the normal \( *N*^* \) of surface \( *B* \). The offset surface is defined by the following parametric equation:

\[
S(u,v) = B(u,v) + d \cdot N(u,v), \quad (u,v) \in \text{domain}(B) . \quad N(u,v) = \left[ S'_{\{u\}}(u,v), S'_{\{v\}}(u,v) \right]
\]

if \( \left[ S'_{\{u\}}(u,v), S'_{\{v\}}(u,v) \right] \neq \vec{0} \).

The example record is interpreted as an offset surface with a distance \( d^* = -2 \) and base surface \( B(u,v) = (1,2,3) + u \cdot (1,0,0) + v \cdot (0,1,0) \). The offset surface is defined by the following parametric equation:

\[
S(u,v) = (1,2,3) + u \cdot (1,0,0) + v \cdot (0,1,0) - 2 \cdot (0,0,1)
\]
2D curves

Example

```
Curve2ds 24
1 0 0 1 0
1 0 0 1 0
1 3 0 0 -1
1 0 0 0 1
1 0 -2 1 0
1 0 0 1 0
1 0 0 0 -1
1 0 0 0 1
1 0 0 1 0
1 0 1 1 0
1 3 0 0 -1
1 1 0 0 1
1 0 -2 1 0
1 0 1 1 0
1 0 0 0 -1
1 1 0 0 1
1 0 0 0 1
1 0 0 1 0
1 3 0 0 1
1 0 0 1 0
1 0 0 0 1
1 0 2 1 0
1 3 0 0 1
1 0 2 1 0
```

BNF-like Definition

```
<2D curves> = <2D curve header> \n  <2D curve records>;

<2D curve header> = "Curve2ds" __ <2D curve cou
<2D curve count> = <int>;

<2D curve records> = <2D curve record> ^ <2D curve count>;

<2D curve record> =
<2D curve record 1> |
<2D curve record 2> |
<2D curve record 3> |
<2D curve record 4> |
<2D curve record 5> |
<2D curve record 6> |
<2D curve record 7> |
<2D curve record 8> |
<2D curve record 9>;

**Line - <2D curve record 1>**

**Example**

```
1 3 0 0 -1
```

**BNF-like Definition**

```
<2D curve record 1> = "1" <_> <2D point> <_> <2D direction> <_\n>;
```

**Description**

<2D curve record 1> describes a line. The line data consist of a 2D point *P* and a 2D direction *D*. The line passes through the point *P*, has the direction *D* and is defined by the following parametric equation:

\[
C(u) = P + u \cdot D, \quad u \in (-\infty, \infty).
\]

The example record is interpreted as a line which passes through a point...
\[ \text{Circle - <2D curve record 2>}
\]

\textbf{Example}

\begin{verbatim}
  2 1 2 1 0 -0 1 3
\end{verbatim}

\textbf{BNF-like Definition}

\begin{verbatim}
<2D curve record 2> = "2" <_> <2D circle center> <_> 
  <2D circle Dx> <_> <2D circle Dy> <_> <2D circle radius> <_> ;

<2D circle center> = <2D point> ;

<2D circle Dx> = <2D direction> ;

<2D circle Dy> = <2D direction> ;

<2D circle radius> = <real> ;
\end{verbatim}

\textbf{Description}

\texttt{<2D curve record 2> } describes a circle. The circle data consist of a 2D point \( P \), orthogonal 2D directions \( \text{D}_x \) * and \( \text{D}_y \) * and a non-negative real *r*. The circle has a center \( P \). The circle plane is parallel to directions \( \text{D}_x \) * and \( \text{D}_y \) *. The circle has a radius \( *r* \) * and is defined by the following parametric equation:

\[
\text{C(u)}=P+r \cdot (\cos(u) \cdot \text{D}_x + \sin(u) \cdot \text{D}_y) , \; u \in [0,2\cdot\pi) .
\]

The example record is interpreted as a circle which has a center \( P=(1,2) \). The circle plane is parallel to directions \( \text{D}_x=(1,0) \) and \( \text{D}_y=(0,1) \). The circle has a radius \( r=3 \) and is defined by the following parametric equation:

\[
\text{C(u)}=(1,2)+3 \cdot (\cos(u) \cdot (1,0) + \sin(u) \cdot (0,1)) .
\]
**Ellipse - <2D curve record 3>**

**Example**

| 3 | 1 | 2 | 1 | 0 | -0 | 1 | 4 | 3 |

**BNF-like Definition**

```
<2D curve record 3> = "3" <_> <2D ellipse center > <_> <2D ellipse Dmaj> <_> <2D ellipse Dmin> <_> <2D ellipse Rmaj> <_> <2D ellipse Rmin> <_\n>;

<2D ellipse center> = <2D point>;
<2D ellipse Dmaj> = <2D direction>;
<2D ellipse Dmin> = <2D direction>;
<2D ellipse Rmaj> = <real>;
<2D ellipse Rmin> = <real>;
```

**Description**

<2D curve record 3> describes an ellipse. The ellipse data are 2D point $P$, orthogonal 2D directions *$D_{maj}$* and *$D_{min}$* and non-negative reals *$r_{maj}$* and *$r_{min}$* that *$r_{maj}$* $\leq$ *$r_{min}$*. The ellipse has a center $P$, major and minor axis directions $D_{maj}$ and $D_{min}$, major and minor radii *$r_{maj}$* and *$r_{min}$* and is defined by the following parametric equation:

$$ C(u) = P + r_{maj} \cdot \cos(u) \cdot D_{maj} + r_{min} \cdot \sin(u) \cdot D_{min} , \quad u \in [0, \; 2 \cdot \pi) . $$

The example record is interpreted as an ellipse which has a center $P$ = (1,2), major and minor axis directions $D_{maj}$=(1,0) and $D_{min}$=(0,1), major and minor radii $r_{maj}$=4 and $r_{min}$=3 and is defined by the following
parametric equation: \[
C(u)=(1,2)+4 \cdot \cos(u) \cdot (1,0)+3 \cdot \sin(u) \cdot (0,1) \].

**Parabola - <2D curve record 4>**

**Example**

| 4 1 2 1 0 | -0 | 1 16 |

**BNF-like Definition**

```
<2D curve record 4> = "4" <_> <2D parabola origin> <_> <2D parabola Dx> <_> <2D parabola Dy> <_> <2D parabola focal length> 

<2D parabola origin> = <2D point>;
<2D parabola Dx> = <2D direction>;
<2D parabola Dy> = <2D direction>;
<2D parabola focal length> = <real>;
```

**Description**

<2D curve record 4> describes a parabola. The parabola data consist of a 2D point \( P \), orthogonal 2D directions \( D_x \) and \( D_y \) and a non-negative real \( f \). The parabola coordinate system has its origin \( P \) and axis directions \( D_x \) and \( D_y \). The parabola has a focus length \( f \) and is defined by the following parametric equation:

\[
C(u)=P+\frac{u^2}{4 \cdot f} \cdot D_x+u \cdot D_y, \quad u \in (-\infty, \infty) \Leftarrow f \neq 0; \quad C(u)=P+u \cdot D_x, \quad u \in (-\infty, \infty) \Leftarrow f = 0; \quad (degenerated\;case).
\]

The example record is interpreted as a parabola in plane which passes through a point \( P=(1,2) \) and is parallel to directions \( D_x=(1,0) \) and \( D_y=\).
The parabola has a focus length $f=16$ and is defined by the following parametric equation: $C(u) = (1,2) + \frac{u^2}{64} \cdot (1,0) + u \cdot (0,1)$.

Hyperbola - <2D curve record 5>

Example

5 1 2 1 0 -0 1 3 4

BNF-like Definition

```
<2D curve record 5> = "5" <_> <2D hyperbola origin> <_> <2D hyperbola Dx> <_> 
  <2D hyperbola Dy> <_> <2D hyperbola Kx> <_> <2D hyperbola Ky> <\n>;

<2D hyperbola origin> = <2D point>;
<2D hyperbola Dx> = <2D direction>;
<2D hyperbola Dy> = <2D direction>;
<2D hyperbola Kx> = <real>;
<2D hyperbola Ky> = <real>;
```

Description

<2D curve record 5> describes a hyperbola. The hyperbola data consist of a 2D point $P$, orthogonal 2D directions $D_x*$ and $D_y*$ and non-negative reals $k_x*$ and $k_y$. The hyperbola coordinate system has origin $P*$ and axis directions $D_x*$ and $D_y$. The hyperbola is defined by the following parametric equation:

\[
C(u) = P + k_x \cdot \cosh(u) \cdot D_x + k_y \cdot \sinh(u) \cdot D_y, \quad u \in (-\infty; \infty).
\]

The example record is interpreted as a hyperbola with coordinate system
which has origin \(P=(1,2)\) and axis directions \(D_x=(1,0)\) and \(D_y=(0,1)\). Other data for the hyperbola are \(k_x=5\) and \(k_y=4\). The hyperbola is defined by the following parametric equation: \(C(u)=(1,2)+3 \cdot \cosh(u) \cdot (1,0)+4 \cdot \sinh(u) \cdot (0,1)\).

**Bezier Curve - <2D curve record 6>**

**Example**

```
6 1 2 0 1 4 1 -2 5 2 3 6
```

**BNF-like Definition**

```latex
<2D curve record 6> = "6" <_> <2D Bezier rational flag> <_> <2D Bezier degree>
<2D Bezier weight poles> <\_\_\_n>;

<2D Bezier rational flag> = <flag>;

<2D Bezier degree> = <int>;

<2D Bezier weight poles> = (<_> <2D Bezier weight pole>) \& (<2D Bezier degree> <\_\_\_> "1");

<2D Bezier weight pole> = <2D point> [<\_\_\_> <real> ];
```

**Description**

<2D curve record 6> describes a Bezier curve. The curve data consist of a rational flag \(r\), a degree \((m \leq 25)\) and weight poles.

The weight poles are \(m+1\) 2D points \(B_0 \ldots B_m\) *if the flag \(*r* is 0. The weight poles are \(*m+1\) pairs \(*B_0h_0 \ldots B_mh_m\) if the flag \(*r* is 1. Here \(*B_i* is a 2D point and \(*h_i* is a positive real \((0 \leq i \leq m)\) \& \(h_i=1 \((0 \leq i \leq m)\) \) if the flag \(*r* is 0.

The Bezier curve is defined by the following parametric equation:
\[ C(u) = \frac{\sum_{i=0}^{m} B_i \cdot h_i \cdot C_m^i \cdot u^i \cdot (1-u)^{m-i}}{\sum_{i=0}^{m} h_i \cdot C_m^i \cdot u^i \cdot (1-u)^{m-i}}, \quad u \in [0,1] \]

where \( 0^0 \equiv 1 \).

The example record is interpreted as a Bezier curve with a rational flag \( r=1 \), a degree \( m=2 \) and weight poles \( B_0=(0,1), h_0=4, B_1=(1,-2), h_1=5 \) and \( B_2=(2,3), h_2=6 \). The Bezier curve is defined by the following parametric equation:

\[
C(u) = \frac{(0,1) \cdot 4 \cdot (1-u)^2 + (1,-2) \cdot 5 \cdot 2 \cdot u \cdot (1-u) + (2,3) \cdot 6 \cdot u^2}{4 \cdot (1-u)^2 + 5 \cdot 2 \cdot u \cdot (1-u) + 6 \cdot u^2}.
\]

B-spline Curve - <2D curve record 7>

Example

```
7 1 0 1 3 5 0 1 4 1 -2 5 2 3 6
0 1 0.25 1 0.5 1 0.75 1 1 1
```

BNF-like Definition

```
<2D curve record 7> = "7" <_> <2D B-spline rational flag> <_> "0" <_> <2D B-spline degree> <_> <2D B-spline pole count> <_> <2D B-spline multiplicity knot count> <2D B-spline weight poles> <_\n> <2D B-spline multiplicity knots> <_\n>;

<2D B-spline rational flag> = <flag>;

<2D B-spline degree> = <int>;

<2D B-spline pole count> = <int>;

<2D B-spline multiplicity knot count> = <int>;
```
\[ \langle 2D \text{ B-spline weight poles} \rangle = \langle 2D \text{ B-spline weight pole} \rangle \wedge \langle 2D \text{ B-spline pole count} \rangle; \]

\[ \langle 2D \text{ B-spline weight pole} \rangle = \langle \_\rangle \langle 2D \text{ point} \rangle [\langle \_\rangle \langle \_\rangle \langle \text{real} \rangle]; \]

\[ \langle 2D \text{ B-spline multiplicity knots} \rangle = \]

\[ \langle 2D \text{ B-spline multiplicity knot} \rangle \wedge \langle 2D \text{ B-spline multiplicity knot count} \rangle; \]

\[ \langle 2D \text{ B-spline multiplicity knot} \rangle = \langle \_\rangle \langle \_\rangle \langle \text{real} \rangle \langle \_\rangle \langle \text{int} \rangle; \]

**Description**

\(<2D \text{ curve record 7}>\) describes a B-spline curve. The curve data consist of a rational flag \( r \), a degree \(( m \leq 25 )\), a pole count \(( n \geq 2 )\), a multiplicity knot count \( k \), weight poles and multiplicity knots.

The weight poles are \( *n* \) 2D points \( *B_1 \ldots B_n* \) if the flag \(*r*\) is 0. The weight poles are \( *n* \) pairs \( *B_1 h_1 \ldots B_n h_n* \) if the flag \(*r*\) is 1. Here \( *B_i* \) is a 2D point and \( *h_i* \) is a positive real \(( 1 \leq i \leq n )\). \( h_i=1 \ ((1 \leq i \leq n) \) \) if the flag \(*r*\) is 0.

The multiplicity knots are \( k* \) pairs \( *u_1 q_1 \ldots u_k q_k* \). Here \( *u_i* \) is a knot with multiplicity \(( q_{\_i} \geq 1 \; (1 \leq i \leq k) \) \) so that

\[
\langle u_{\_i}\rangle < u_{\_i+1} \rangle; \langle 1 \leq i \leq k \rangle, \langle \sum_{\_i=1}^{k} q_{\_i} = m+n+1 \rangle
\]

The B-spline curve is defined by the following parametric equation:

\[
\frac{\sum_{i=1}^{n} B_{\_i} \cdot h_{\_i} \cdot N_{\_i,m+1}(u)}{\sum_{i=1}^{n} h_{\_i} \cdot N_{\_i,m+1}(u)}, \quad u \in [u_1, u_k]
\]

where functions \( N_{i,j} \) have the following recursion definition by \(*j*

\[
N_{\_i,1}(u) = \begin{bmatrix} 1\end{bmatrix}, \quad \bar{u}_{\_i} \leq u \leq \bar{u}_{\_i+1}
\]

\[
N_{\_i,j}(u) = \frac{u - \bar{u}_{\_i}}{\bar{u}_{\_i+1} - \bar{u}_{\_i}} N_{\_i,j-1}(u) + \frac{\bar{u}_{\_i+j} - u}{\bar{u}_{\_i+j} - \bar{u}_{\_i+1}} N_{\_i-j+1,j-1}(u), \quad u \in [u_{\_i}, u_{\_i+j}]
\]
where

\[
\bar{u}_{i} = u_{j} \quad (1 \leq j \leq k, \sum_{l=1}^{j-1} q_{l} + 1 \leq i \leq \sum_{l=1}^{j} q_{l}) \ .
\]

The example record is interpreted as a B-spline curve with a rational flag \( r = 1 \), a degree \( m = 1 \), a pole count \( n = 3 \), a multiplicity knot count \( k = 5 \), weight poles \( B_1 = (0,1) \), \( h_1 = 4 \), \( B_2 = (1,-2) \), \( h_2 = 5 \) and \( B_3 = (2,3) \), \( h_3 = 6 \) and multiplicity knots \( u_1 = 0, q_1 = 1, u_2 = 0.25, q_2 = 1, u_3 = 0.5, q_3 = 1, u_4 = 0.75, q_4 = 1 \) and \( u_5 = 1, q_5 = 1 \). The B-spline curve is defined by the following parametric equation:

\[
C(u) = \frac{(0,1) \cdot 4 \cdot N_{1,2}(u) + (1,-2) \cdot 5 \cdot N_{2,2}(u) + (2,3) \cdot 6 \cdot N_{3,2}(u)}{4 \cdot N_{1,2}(u) + 5 \cdot N_{2,2}(u) + 6 \cdot N_{3,2}(u)} .
\]

**Trimmed Curve - <2D curve record 8>**

**Example**

```
 8  -4  5
 1  1  2  1  0
```

**BNF-like Definition**

```
<2D curve record 8> = "8" <_> <2D trimmed curve u min> <_> <2D trimmed curve u max> <_

<2D trimmed curve u min> = <real>; 
<2D trimmed curve u max> = <real>; 
```

**Description**
<2D curve record 8> describes a trimmed curve. The trimmed curve data consist of reals \( u_{\min} \) and \( u_{\max} \) and a <2D curve record> so that \( u_{\min} < u_{\max} \). The trimmed curve is a restriction of the base curve \( B^* \) described in the record to the segment \([u_{\min}, u_{\max}] \subseteq \text{domain}(B) \). The trimmed curve is defined by the following parametric equation:

\[
\[ C(u) = B(u), \forall u \in [u_{\min}, u_{\max}] \].
\]

The example record is interpreted as a trimmed curve with \( u_{\min} = -4 \), \( u_{\max} = 5 \) and base curve \( B(u) = (1,2) + u \cdot (1,0) \). The trimmed curve is defined by the following parametric equation: \( C(u) = (1,2) + u \cdot (1,0), \forall u \in [-4,5] \).

**Offset Curve - <2D curve record 9>**

**Example**

<table>
<thead>
<tr>
<th>9 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 2 1 0</td>
</tr>
</tbody>
</table>

**BNF-like Definition**

<2D curve record 9> = "9" <_> <2D offset curve distance> <_\n> <2D curve record>;

<2D offset curve distance> = <real>;

**Description**

<2D curve record 9> describes an offset curve. The offset curve data consist of a distance \( d^* \) and a <2D curve record>. The offset curve is the result of offsetting the base curve \( B^* \) described in the record to the distance \( d^* \) along the vector \( (B'_Y(u), -B'_X(u)) \neq \vec{0} \) where \( B(u) = (B'_X(u), B'_Y(u)) \). The offset curve is defined by the following parametric equation:

\[
\[ C(u) = B(u) + d \cdot (B'_Y(u), -B'_X(u)), \forall u \in \text{domain}(B) \].
\]

The example record is interpreted as an offset curve with a distance
\( d=2 \) and base curve \( B(u)=(1,2)+u \cdot (1,0) \) and is defined by the following parametric equation: \( C(u)=(1,2)+u \cdot (1,0)+2 \cdot (0,-1) \).
3D polygons

Example

```
Polygon3D 1
  2 1
  0.1
  1 0 0 2 0 0
  0 1
```

BNF-like Definition

```plaintext
<3D polygons> = <3D polygon header> \<\n\> <3D polygon records>;

<3D polygon header> = "Polygon3D" \<\> <3D polygon record count>;

<3D polygon records> = <3D polygon record> ^ <3D polygon record count>;

<3D polygon record> =
  <3D polygon node count> \<\> <3D polygon flag of parameter presence> \<\n\>
  <3D polygon deflection> \<\n\>
  <3D polygon nodes> \<\n\>
  [<3D polygon parameters> \<\n\>];

<3D polygon node count> = <int>;

<3D polygon flag of parameter presence> = <flag> ;

<3D polygon deflection> = <real> ;

<3D polygon nodes> = (<3D polygon node> \<\>)^<
3D polygon node count;

<3D polygon node> = <3D point>;

<3D polygon u parameters> = (<3D polygon u parameter> <_>) ^ <3D polygon node count>;

<3D polygon u parameter> = <real>;

**Description**

<3D polygons> record describes a 3D polyline \( L^* \) which approximates a 3D curve \( *C \). The polyline data consist of a node count \(( m \geq 2 \) \), a parameter presence flag \( p \), a deflection \(( d \geq 0 \) \), nodes \( \{ N_{\{i\}}; (1 \leq i \leq m) \} \) and parameters \( \{ u_{\{i\}}; (1 \leq i \leq m) \} \). The parameters are present only if \( p=1 \). The polyline \( L^* \) passes through the nodes. The deflection \( *d* \) describes the deflection of polyline \( *L^* \) from the curve \( *C* \):

\[
\underset{P \in C}{\max}\ \underset{Q \in L}{\min}|Q-P| \leq d.
\]

The parameter \( \{ u_{\{i\}}; (1 \leq i \leq m) \} \) is the parameter of the node \( N_{i}^* \) on the curve \( *C* \):

\[
\{ C(u_{\{i\}})=N_{\{i\}} \}.
\]

The example record describes a polyline from \( m=2 \) nodes with a parameter presence flag \( p=1 \), a deflection \( d=0.1 \), nodes \( N_{1}=(1,0,0) \) and \( N_{2}=(2,0,0) \) and parameters \( u_{1}=0 \) and \( u_{2}=1 \).
# Triangulations

## Example

<table>
<thead>
<tr>
<th>Triangulations 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 2 1 0</td>
</tr>
<tr>
<td>0 0 0 0 0 3 0 2 3 0 2 0 0 0 3 0 3 -2 0 -2 2 4 3</td>
</tr>
<tr>
<td>2 1 4</td>
</tr>
<tr>
<td>4 2 1 0</td>
</tr>
<tr>
<td>0 0 0 1 0 0 1 0 3 0 0 3 0 0 0 1 3 1 3 0 3 2 1 3</td>
</tr>
<tr>
<td>1 4</td>
</tr>
<tr>
<td>4 2 1 0</td>
</tr>
<tr>
<td>0 0 3 0 2 3 1 2 3 1 0 3 0 0 0 2 1 2 1 0 3 2 1 3</td>
</tr>
<tr>
<td>1 4</td>
</tr>
<tr>
<td>4 2 1 0</td>
</tr>
<tr>
<td>0 2 0 1 2 0 1 2 3 0 2 3 0 0 0 1 3 1 3 0 3 2 1 3</td>
</tr>
<tr>
<td>1 4</td>
</tr>
<tr>
<td>4 2 1 0</td>
</tr>
<tr>
<td>0 0 0 0 2 0 1 2 0 1 0 0 0 0 0 2 1 2 1 0 3 2 1 3</td>
</tr>
<tr>
<td>1 4</td>
</tr>
<tr>
<td>4 2 1 0</td>
</tr>
<tr>
<td>1 0 0 1 0 3 1 2 3 1 2 0 0 0 3 0 3 -2 0 -2 2 4 3</td>
</tr>
<tr>
<td>2 1 4</td>
</tr>
</tbody>
</table>

## BNF-like Definition

\[
\text{<triangulations> = <triangulation header> } \_\text{\textbackslash n} \\
\text{<triangulation records>;}
\]

\[
\text{<triangulation header> = "Triangulations" } \_\text{>}
\]

\[
\text{<triangulation count>;
}\]

\[
\text{<triangulation records> = <triangulation record> } ^\wedge \\
\]

\[
\text{<triangulation record> = <triangulation node count>
}\]
<triangulation node count> = <int>;

<triangulation triangle count> = <int>;

<triangulation parameter presence flag> = <flag>;

<triangulation deflection> = <real>;

<triangulation nodes> = (<triangulation node> <_

<triangulation node> = <3D point>;

<triangulation u v parameters> =
(<triangulation u v parameter pair> <_

<triangulation u v parameter pair> = <real> <_

<real>;

<triangulation triangles> = (<triangulation triangle> <_

<triangulation triangle> = <int> <_

Description

<triangulation record> describes a triangulation T* which approximates a surface S. The triangulation data consist of a node count \( m \geq 3 \), a triangle count \( k \geq 1 \), a parameter presence flag \( p \), a deflection \( d \geq 0 \), nodes \( N_{i} \ (1 \leq i \leq m) \), parameter pairs \( u_{i} \ v_{i} \); \n
The parameters are present only if \( p = 1 \). The deflection describes the triangulation deflection from the surface:

\[
\underset{P \in S}{\max} \; \underset{Q \in T}{\min} |Q-P| \leq d.
\]

The parameter pair \((u_{i}, v_{i})\) describes the parameters of node \( N_{i} \) on the surface:

\[
S(u_{i}, v_{i}) = N_{i}.
\]

The triangle \((n_{j,1}, n_{j,2}, n_{j,3})\) is interpreted as a triangle of nodes \((N_{n_{j,1}}, N_{n_{j,2}}, N_{n_{j,3}})\) with circular traversal of the nodes in the order \((N_{n_{j,1}}, N_{n_{j,2}}, N_{n_{j,3}})\) and \((N_{n_{j,1}}, N_{n_{j,3}}, N_{n_{j,2}})\). From any side of the triangulation \( T \) all its triangles have the same direction of the node circular traversal: either clockwise or counterclockwise.

Triangulation record

<table>
<thead>
<tr>
<th>4 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
</tr>
<tr>
<td>3 0 2 3</td>
</tr>
<tr>
<td>0 2 0 0</td>
</tr>
<tr>
<td>3 0 3 -2</td>
</tr>
<tr>
<td>0 -2 2 4</td>
</tr>
<tr>
<td>3 2 1 4</td>
</tr>
</tbody>
</table>

describes a triangulation with \( m = 4 \) nodes, \( k = 2 \) triangles, parameter presence flag \( p = 1 \), deflection \( d = 0 \), nodes \( N_{1} = (0,0,0) \), \( N_{2} = (0,0,3) \), \( N_{3} = (0,2,3) \) and \( N_{4} = (0,2,0) \), parameters \((u_{1}, v_{1}) = (0,0)\), \((u_{2}, v_{2}) = (3,0)\), \((u_{3}, v_{3}) = (3,-2)\) and \((u_{4}, v_{4}) = (0,-2)\), and triangles \((n_{1,1}, n_{1,2}, n_{1,3}) = (2,4,3)\) and \((n_{2,1}, n_{2,2}, n_{2,3}) = (2,1,4)\). From the point \((1,0,0)\) \((-1,0,0)\) the triangles have clockwise (counterclockwise) direction of the node circular traversal.
Polygons on triangulations

Example

```
PolygonOnTriangulations 24
2 1 2
p 0.1 1 0 3
2 1 4
p 0.1 1 0 3
2 2 3
p 0.1 1 0 2
2 1 2
p 0.1 1 0 2
2 4 3
p 0.1 1 0 3
2 1 4
p 0.1 1 0 3
2 1 4
p 0.1 1 0 2
2 1 2
p 0.1 1 0 2
2 1 2
p 0.1 1 0 3
2 2 3
p 0.1 1 0 3
2 2 3
p 0.1 1 0 2
2 4 3
p 0.1 1 0 2
2 4 3
p 0.1 1 0 3
2 2 3
p 0.1 1 0 3
2 1 4
p 0.1 1 0 2
2 4 3
```
BNF-like Definition

\[
<\text{polygons on triangulations}> = <\text{polygons on triangulations header}> \_<\_\_n>
<\text{polygons on triangulations records}>;
\]

\[
<\text{polygons on triangulations header}> = 
"\text{PolygonOnTriangulations}" \_<\_\_<\text{polygons on triangulations record count}>;
\]

\[
<\text{polygons on triangulations record count}> = <\text{int}>;
\]

\[
<\text{polygons on triangulations records}> = 
<\text{polygons on triangulations record}> \_<\_\_<\text{polygons on triangulations record count}>;
\]

\[
<\text{polygons on triangulations record}> = 
<\text{polygons on triangulations node count}> \_<\_\_<\text{polygons on triangulations record count}>;
\]
Description

The polygons on triangulations node numbers describe a polyline $L^*$ on a triangulation which approximates a curve $C$. The polyline data consist of a node count \( m \geq 2 \), node numbers \( n_i \geq 1 \), deflection \( d \geq 0 \), a parameter presence flag \( p \) and parameters \( u_i \) (1 \( \leq i \leq m \)). The parameters are present only if \( p=1 \). The deflection \( d^* \) describes the deflection of polyline $L^*$ from the curve $C$: 
Parameter \( u_i \) \( (1 \leq i \leq m) \) is the \( i \)-th node \( C(u_i) \) parameter on curve \( C \).
Geometric Sense of a Curve

Geometric sense of curve *C* described above is determined by the direction of parameter *u* increasing.
Shapes

An example of section shapes and a whole *.brep file are given in chapter 7 Appendix.

BNF-like Definition

```
<shapes>  =  <shape header>  \n  <shape records>  
  <shape final record>

  <shape header>  =  "TShapes"  \_  <shape count>

  <shape count>  =  <int>

  <shape records>  =  <shape record>  \_  <shape count>

  <shape record>  =  <shape subrecord>  \n  <shape flag word>  \n  <shape subshapes>

  <shape flag word>  =  <flag>  \_  7

  <shape subshapes>  =  (<shape subshape>  \n)*  "\n"

  <shape subshape>  =  
  <shape subshape orientation>  <shape subshape number>

  <shape subshape orientation>  =  "+"  |  "-"  |  "i"  |  "e"

  <shape subshape number>  =  <int>

  <shape location number>  =  <int>
```
<shape final record> = <shape subshape>;

<shape subrecord> =
("Ve" <\n> <vertex data> <\n>) |
("Ed" <\n> <edge data> <\n>) |
("Wi" <\n> <\n>) |
("Fa" <\n> <face data>) |
("Sh" <\n> <\n>) |
("So" <\n> <\n>) |
("CS" <\n> <\n>) |
("Co" <\n> <\n>);

Description

<shape flag word> \{( f_.{1}; f_.{2}; f_.{3}; f_.{4}; f_.{5}; f_.{6}; f_.{7} \} <flag>s \{( f_.{i};(1\leq i \leq 7) \} are interpreted as shape flags in the following way:

- \{( f_.{1} \} – free;
- \{( f_.{2} \} – modified;
- \{( f_.{3} \} – IGNORED(version 1) \ checked (version 2);
- \{( f_.{4} \} – orientable;
- \{( f_.{5} \} – closed;
- \{( f_.{6} \} – infinite;
- \{( f_.{7} \} – convex.

The flags are used in a special way [1].

<shape subshape orientation> is interpreted in the following way:

- + – forward;
- - – reversed;
- i – internal;
- e – external.

<shape subshape orientation> is used in a special way [1].

<shape subshape number> is the number of a <shape record> which is located in this section above the <shape subshape number>. <shape record> numbering is backward and starts from 1.
<shape subrecord> types are interpreted in the following way:

- "Ve" – vertex;
- "Ed" – edge;
- "Wi" – wire;
- "Fa" – face;
- "Sh" – shell;
- "So" – solid;
- "CS" – compsolid;
- "Co" – compound.

<shape final record> determines the orientation and location for the whole model.
Common Terms

The terms below are used by <vertex data>, <edge data> and <face data>.

BNF-like Definition

<location number> = <int>;

<3D curve number> = <int>;

<surface number> = <int>;

<2D curve number> = <int>;

<3D polygon number> = <int>;

<triangulation number> = <int>;

<polygon on triangulation number> = <int>;

<curve parameter minimal and maximal values> = <real> _> <real>;

<curve values for parameter minimal and maximal values> =
real> _> <real> _> <real> _> <real> _> <real>;

Description

<location number> is the number of <location record> from section locations. <location record> numbering starts from 1. <location number> 0 is interpreted as the identity location.

<3D curve number> is the number of a <3D curve record> from subsection <3D curves> of section <geometry>. <3D curve record> numbering starts from 1.
<surface number> is the number of a <surface record> from subsection <surfaces> of section <geometry>. <surface record> numbering starts from 1.

<2D curve number> is the number of a <2D curve record> from subsection <2D curves> of section <geometry>. <2D curve record> numbering starts from 1.

<3D polygon number> is the number of a <3D polygon record> from subsection <3D polygons> of section <geometry>. <3D polygon record> numbering starts from 1.

<triangulation number> is the number of a <triangulation record> from subsection <triangulations> of section <geometry>. <triangulation record> numbering starts from 1.

<polygon on triangulation> number is the number of a <polygons on triangulations record> from subsection <polygons on triangulations> of section <geometry>. <polygons on triangulations record> numbering starts from 1.

<curve parameter minimal and maximal values> $u_{min} \text{ and } u_{max}$ are the curve parameter $u$ bounds: $u_{min} \leq u \leq u_{max}$.

<curve values for parameter minimal and maximal values> $u_{min} \text{ and } u_{max}$ are real pairs $(x_{min}, y_{min}) \text{ and } (x_{max}, y_{max})$ that $(x_{min}, y_{min}) = C(u_{min})$ and $(x_{max}, y_{max}) = C(u_{max})$ where $C$ is a parametric equation of the curve.
**Vertex data**

**BNF-like Definition**

```plaintext
<vertex data> = <vertex data tolerance> <\n> <vertex data 3D representation> <\n>
    <vertex data representations>;

<vertex data tolerance> = <real>;

<vertex data 3D representation> = <3D point>;

<vertex data representations> = (<vertex data representation> <\n>)* "0 0";

<vertex data representation> = <vertex data representation u parameter> <">
    <vertex data representation data> <"> <location number>;

<vertex data representation u parameter> = <real >;

<vertex data representation data> = 
    ("1" <"> <vertex data representation data 1>) | 
    ("2" <"> <vertex data representation data 2>) | 
    ("3" <"> <vertex data representation data 3>);

<vertex data representation data 1> = <3D curve number>;

<vertex data representation data 2> = <2D curve number> <"> <surface number>;

<vertex data representation data 3> = 
    <vertex data representation v parameter> <">
```
Description

The usage of \(<\text{vertex data representation u parameter}> *U*\) is described below.

\(<\text{vertex data representation data 1}>\) and parameter \(U*\) describe the position of the vertex \(*V*\) on a 3D curve \(*C*\). Parameter \(U*\) is a parameter of the vertex \(*V*\) on the curve \(*C*\): \(C(u)=V\).

\(<\text{vertex data representation data 2}>\) and parameter \(U*\) describe the position of the vertex \(*V*\) on a 2D curve \(*C*\) which is located on a surface. Parameter \(U*\) is a parameter of the vertex \(*V*\) on the curve \(*C*\): \(C(u)=V\).

\(<\text{vertex data representation data 3}>\) and parameter \(u*\) describe the position of the vertex \(*V*\) on a surface \(*S*\) through \(<\text{vertex data representation v parameter}> *v\): \(S(u,v)=V\).

\(<\text{vertex data tolerance}> t*\) describes the maximum distance from the vertex \(*V*\) to the set \(*R*\) of vertex \(*V*\) representations:

\[
\begin{array}{l}
\left\lfloor \underset{P \in R}{\max} |P-V| \right\rfloor \leq t.
\end{array}
\]
Edge data

BNF-like Definition

```
<edge data> = <_> <edge data tolerance> <_> <edge data same parameter flag> <_> <edge data same range flag> <_> <edge data degenerated flag> <_
> <edge data representations>;

<edge data tolerance> = <real>;
<edge data same parameter flag> = <flag>;
<edge data same range flag> = <flag>;
<edge data degenerated flag> = <flag>;
<edge data representations> = (<edge data representation> <_
>)* "0";
<edge data representation> = "1" <_> <edge data representation data 1>
"2" <_> <edge data representation data 2>
"3" <_> <edge data representation data 3>
"4" <_> <edge data representation data 4>
"5" <_> <edge data representation data 5>
"6" <_> <edge data representation data 6>
"7" <_> <edge data representation data 7>;
<edge data representation data 1> = <3D curve number> <_> <location number> <_
><curve parameter minimal and maximal values>;
<edge data representation data 2> = <2D curve number> <_> <surface number> <_
><location number> <_> <curve parameter minimal and
maximal values>
[<_
>	<curve values for parameter minimal and maximal values>];

<edge data representation data 3> = (<2D curve number> <_> ) ^ 2 <continuity order> <_> <surface number> <_> <location number> <_> <curve parameter minimal and maximal values> <\n> <curve values for parameter minimal and maximal values>];

<continuity order> = "C0" | "C1" | "C2" | "C3" | "CN" | "G1" | "G2".

<edge data representation data 4> = <continuity order> (<_> <surface number> <_> <location number>) ^ 2;

<edge data representation data 5> = <3D polygon number> <_> <location number>;

<edge data representation data 6> = <polygon on triangulation number> <_> <triangulation number> <_> <location number>;

<edge data representation data 7> = (<polygon on triangulation number> <_> ) ^ 2 <triangulation number> <_> <location number>;

**Description**

Flags <edge data same parameter flag>, <edge data same range flag> and <edge data degenerated flag> are used in a special way [1].

<edge data representation data 1> describes a 3D curve.

<edge data representation data 2> describes a 2D curve on a surface. <curve values for parameter minimal and maximal values> are used only
in version 2.

<edge data representation data 3> describes a 2D curve on a closed surface. <curve values for parameter minimal and maximal values> are used only in version 2.

<edge data representation data 5> describes a 3D polyline.

<edge data representation data 6> describes a polyline on a triangulation.

<edge data tolerance> $t^*$ describes the maximum distance from the edge $E$ to the set $R$ of edge $E$ representations:

$$\left[ \underset{C \in R}{\max}, \underset{P \in E}{\max}, \underset{Q \in C}{\min} \right] |Q-P| \leq t$$
Face data

BNF-like Definition

```plaintext
<face data> = <face data natural restriction flag> 
  <> <face data tolerance>  <> <surface number> 
  <> <location number>  
  <triangulation number> ];
<face data natural restriction flag> = <flag>;
<face data tolerance> = <real>;
```

Description

<face data> describes a surface $S*$ of face $F*$ and a triangulation $T*$ of face $F$. The surface $S*$ may be empty: <surface number> = 0.

<face data tolerance> $t*$ describes the maximum distance from the face $F*$ to the surface $S$:

$$\left[ \underset{P \in F}{\text{max}}; \underset{Q \in S}{\text{min}} |Q - P| \right] \leq t$$

Flag <face data natural restriction flag> is used in a special way [1].
Appendix

This chapter contains a *.brep file example.

```
DBRep_DrawableShape

CASCADE Topology V1, (c) Matra-Datavision
Locations 3

1

0 0 1
0
1 0 0
0
0 1 0
0

1

1 0 0
4
0 1 0
5
0 0 1

2 1 1 2 1 0
Curve2ds 24
1 0 0 1 0
1 0 0 1 0
1 3 0 0 -1
1 0 0 0 1
1 0 -2 1 0
1 0 0 1 0
1 0 0 0 -1
1 0 0 0 1
1 0 0 1 0
1 0 1 1 0
1 3 0 0 -1
```
Curves 13
1 0 0 0 0 0 1
1 0 0 3 -0 1 0
1 0 2 0 0 0 1
1 0 0 0 -0 1 0
1 1 0 0 0 0 1
1 1 0 3 0 1 0
1 1 2 0 0 0 1
1 1 0 0 -0 1 0
1 0 0 0 1 0 -0
1 0 0 3 1 0 -0
1 0 2 0 1 0 -0
1 0 2 3 1 0 -0
1 1 0 0 1 0 0
Polygon3D 1
2 1
0.1
1 0 0 2 0 0
0 1
PolygonOnTriangulations 24
2 1 2
p 0.1 1 0 3
2 1 4
p 0.1 1 0 3
Surfaces 6
1 0 0 0 1 0 -0 0 0 1 0 -1 0
1 0 0 0 -0 1 0 0 0 1 1 0 -0
1 0 0 3 0 0 1 1 0 -0 -0 1 0
1 0 2 0 -0 1 0 0 0 1 1 0 -0
1 0 0 0 0 0 1 1 0 -0 -0 1 0
1 1 0 0 1 0 -0 0 0 1 0 -1 0
Triangulations 6
4 2 1 0
0 0 0 0 0 3 0 2 3 0 2 0 0 0 3 0 3 -2 0 -2 2 4 3
2 1 4
4 2 1 0
0 0 0 1 0 0 1 0 3 0 0 3 0 0 0 1 3 1 3 0 3 2 1 3
1 4
4 2 1 0
0 0 3 0 2 3 1 2 3 1 0 3 0 0 0 2 1 2 1 0 3 2 1 3
1 4
4 2 1 0
0 2 0 1 2 0 1 2 3 0 2 3 0 0 0 1 3 1 3 0 3 2 1 3
1 4
4 2 1 0
0 0 0 0 2 0 1 2 0 1 0 0 0 0 0 2 1 2 1 0 3 2 1 3
1 4
4 2 1 0
1 0 0 1 0 3 1 2 3 1 2 0 0 0 3 0 3 -2 0 -2 2 4 3
2 1 4
TShapes 39
Ve
1e-007
<table>
<thead>
<tr>
<th>0 0 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
</tr>
</tbody>
</table>

```
0101101
*
Ve
1e-007
0 0 0
0 0
```

```
0101101
*
Ed
1e-007 1 1 0
1 1 0 0 3
2 1 1 0 0 3
2 2 2 0 0 3
6 1 1 0
6 2 2 0
0
```

```
0101000
-39 0 +38 0 *
Ve
1e-007
0 2 3
0 0
```

```
0101101
*
Ed
1e-007 1 1 0
1 2 0 0 2
2 3 1 0 0 2
2 4 3 0 0 2
6 3 1 0
6 4 3 0
```
0

0101000
-36 0 +39 0 *
Ve
1e-007
0 2 0
0 0

0101101
*
Ed
1e-007 1 1 0
1 3 0 0 3
2 5 1 0 0 3
2 6 4 0 0 3
6 5 1 0
6 6 4 0
0

0101000
-36 0 +34 0 *
Ed
1e-007 1 1 0
1 4 0 0 2
2 7 1 0 0 2
2 8 5 0 0 2
6 7 1 0
6 8 5 0
0

0101000
-34 0 +38 0 *
Wi

0101000
-37 0 -35 0 +33 0 +32 0 *
Fa
0  1e-007 1 0
2  1
0101000
+31 0 *
Ve
1e-007
1 0 3
0 0
0101101
*
Ve
1e-007
1 0 0
0 0
0101101
*
Ed
  1e-007 1 1 0
1   5 0 0 3
2   9 6 0 0 3
2   10 2 0 0 3
6   9 6 0
6   10 2 0
0
0101000
-29 0 +28 0 *
Ve
1e-007
1 2 3
0 0
0101101
*
Ed
  1e-007 1 1 0
  1 6 0 0 2
  2 11 6 0 0 2
  2 12 3 0 0 2
  6 11 6 0
  6 12 3 0
  0

0101000
  -26 0 +29 0 *
Ve
  1e-007
  1 2 0
  0 0

0101101
  *
Ed
  1e-007 1 1 0
  1 7 0 0 3
  2 13 6 0 0 3
  2 14 4 0 0 3
  6 13 6 0
  6 14 4 0
  0

0101000
  -26 0 +24 0 *
Ed
  1e-007 1 1 0
  1 8 0 0 2
  2 15 6 0 0 2
  2 16 5 0 0 2
  6 15 6 0
  6 16 5 0
  0
0101000
-24 0 +28 0 *
Wi

0101000
-27 0 -25 0 +23 0 +22 0 *
Fa
0 1e-007 6 0
2 6
0101000
+21 0 *
Ed
1e-007 1 1 0
1 9 0 0 1
2 17 2 0 0 1
2 18 5 0 0 1
6 17 2 0
6 18 5 0
0

0101000
-28 0 +38 0 *
Ed
1e-007 1 1 0
1 10 0 0 1
2 19 2 0 0 1
2 20 3 0 0 1
6 19 2 0
6 20 3 0
0

0101000
-29 0 +39 0 *
Wi

0101000
-19 0 -27 0 +18 0 +37 0 *
Fa
0 1e-007 2 0
2 2
0101000
+17 0 *
Ed
  1e-007 1 1 0
  1 11 0 0 1
  2 21 4 0 0 1
  2 22 5 0 0 1
  6 21 4 0
  6 22 5 0
  0

0101000
-24 0 +34 0 *
Ed
  1e-007 1 1 0
  1 12 0 0 1
  2 23 4 0 0 1
  2 24 3 0 0 1
  6 23 4 0
  6 24 3 0
  0

0101000
-26 0 +36 0 *
Wi

0101000
-15 0 -23 0 +14 0 +33 0 *
Fa
0 1e-007 4 0
2 4
0101000
+13 0 *
Wi

0101000
-32 0 -15 0 +22 0 +19 0 *

Fa
0 1e-007 5 0
2 5
0101000
+11 0 *

Wi

0101000
-35 0 -14 0 +25 0 +18 0 *

Fa
0 1e-007 3 0
2 3
0101000
+9 0 *

Sh

0101100
-30 0 +20 0 -16 0 +12 0 -10 0 +8 0 *

So

0100000
+7 0 *

CS

0101000
+6 3 *

Ve
1e-007
1 0 0
0 0

0101101
*
Modeling Algorithms

Table of Contents

- Introduction
- Geometric Tools
- Intersections
  - Intersection of two curves
  - Intersection of Curves and Surfaces
  - Intersection of two Surfaces
- Interpolations
  - Geom2dAPI_Interpolate
  - GeomAPI_Interpolate
- Lines and Circles from Constraints
  - Types of constraints
  - Available types of lines and circles
  - Types of algorithms
- Curves and Surfaces from Constraints
  - Faired and Minimal Variation 2D Curves
  - Ruled Surfaces
  - Pipe Surfaces
Filling a contour
Plate surfaces
Projections
Projection of a 2D Point on a Curve
Projection of a 3D Point on a Curve
Projection of a Point on a Surface
Switching from 2d and 3d Curves
Topological Tools
Creation of the faces from wireframe model
Classification of the shapes
Orientation of the shapes in the container
Making new shapes
Building PCurves
Checking the validity of the shapes
Taking a point inside the face
Getting normal for the face
The Topology API
Error Handling in the Topology API
Standard Topological Objects
Vertex
Edge
Basic edge construction
method
  ↓ Supplementary edge construction methods
  ↓ Other information and error status
  ↓ Edge 2D
  ↓ Polygon
  ↓ Face
    ↓ Basic face construction method
    ↓ Supplementary face construction methods
    ↓ Error status
  ↓ Wire
  ↓ Shell
  ↓ Solid
  ↓ Object Modification
    ↓ Transformation
    ↓ Duplication
  ↓ Primitives
    ↓ Making Primitives
      ↓ Box
      ↓ Wedge
      ↓ Rotation object
      ↓ Cylinder
      ↓ Cone
      ↓ Sphere
      ↓ Torus
      ↓ Revolution
    ↓ Sweeping: Prism, Revolution and Pipe
      ↓ Sweeping
      ↓ Prism
Rotational Sweep

Boolean Operations
  Input and Result Arguments
  Implementation

Fillets and Chamfers
  Fillets
  Fillet on shape
  Chamfer
  Fillet on a planar face

Offsets, Drafts, Pipes and Evolved shapes
  Offset computation
  Shelling
  Draft Angle
  Pipe Constructor
  Evolved Solid

Sewing
  Introduction
  Sewing Algorithm
  Tolerance Management
  Manifold and Non-manifold Sewing
  Local Sewing

Features
  Form Features
    Prism
    Draft Prism
    Revolution
    Pipe
  Mechanical Features
    Linear Form
    Gluer
    Split Shape
  Hidden Line Removal
Examples
- Meshing
  - Mesh presentations
  - Meshing algorithm
Introduction

This manual explains how to use the Modeling Algorithms. It provides basic documentation on modeling algorithms. For advanced information on Modeling Algorithms, see our E-learning & Training offerings.

The Modeling Algorithms module brings together a wide range of topological algorithms used in modeling. Along with these tools, you will find the geometric algorithms, which they call.
Geometric Tools

Open CASCADE Technology geometric tools provide algorithms to:

- Calculate the intersection of two 2D curves, surfaces, or a 3D curve and a surface;
- Project points onto 2D and 3D curves, points onto surfaces, and 3D curves onto surfaces;
- Construct lines and circles from constraints;
- Construct curves and surfaces from constraints;
- Construct curves and surfaces by interpolation.
Intersections

The Intersections component is used to compute intersections between 2D or 3D geometrical objects:

- the intersections between two 2D curves;
- the self-intersections of a 2D curve;
- the intersection between a 3D curve and a surface;
- the intersection between two surfaces.

The `Geom2dAPI_InterCurveCurve` class allows the evaluation of the intersection points (`gp_Pnt2d`) between two geometric curves (`Geom2d_Curve`) and the evaluation of the points of self-intersection of a curve.

In both cases, the algorithm requires a value for the tolerance (Standard_Real) for the confusion between two points. The default tolerance value used in all constructors is `1.0e-6`. 
The algorithm returns a point in the case of an intersection and a segment in the case of tangent intersection.

**Intersection of two curves**

`Geom2dAPI_InterCurveCurve` class may be instantiated for intersection of curves $C1$ and $C2$.

```
Geom2dAPI_InterCurveCurve Intersector(C1,C2,tolerance);
```

or for self-intersection of curve $C3$.

```
Geom2dAPI_InterCurveCurve Intersector(C3,tolerance);
Standard_Integer N = Intersector.NbPoints();
```

Calls the number of intersection points

To select the desired intersection point, pass an integer index value in argument.

```
gp_Pnt2d P = Intersector.Point(Index);
```

To call the number of intersection segments, use

```
Standard_Integer M = Intersector.NbSegments();
```
To select the desired intersection segment pass integer index values in argument.

    Handle(Geom2d_Curve) Seg1, Seg2;
    Intersector.Segment(Index,Seg1,Seg2);
    // if intersection of 2 curves
    Intersector.Segment(Index,Seg1);
    // if self-intersection of a curve

If you need access to a wider range of functionalities the following method will return the algorithmic object for the calculation of intersections:

    Geom2dInt_GInter& TheIntersector = Intersector.Intersector();

**Intersection of Curves and Surfaces**

The *GeomAPI_IntCS* class is used to compute the intersection points between a curve and a surface.

This class is instantiated as follows:

    GeomAPI_IntCS Intersector(C, S);

To call the number of intersection points, use:

    Standard_Integer nb = Intersector.NbPoints();
    gp_Pnt& P = Intersector.Point(Index);

Where *Index* is an integer between 1 and *nb*, calls the intersection points.

**Intersection of two Surfaces**

The *GeomAPI_IntSS* class is used to compute the intersection of two surfaces from *Geom_Surface* with respect to a given tolerance.

This class is instantiated as follows:
GeomAPI_IntSS Intersector(S1, S2, Tolerance);

Once the GeomAPI_IntSS object has been created, it can be interpreted.

Standard_Integer nb = Intersector. NbLines();

Calls the number of intersection curves.

Handle(Geom_Curve) C = Intersector.Line(Index)

Where Index is an integer between 1 and nb, calls the intersection curves.
Interpolations

The Interpolation Laws component provides definitions of functions: \( y=f(x) \).

In particular, it provides definitions of:

- a linear function,
- an \( S \) function, and
- an interpolation function for a range of values.

Such functions can be used to define, for example, the evolution law of a fillet along the edge of a shape.

The validity of the function built is never checked: the Law package does not know for what application or to what end the function will be used. In particular, if the function is used as the evolution law of a fillet, it is important that the function is always positive. The user must check this.

Geom2dAPI_Interpolate

This class is used to interpolate a BSplineCurve passing through an array of points. If tangency is not requested at the point of interpolation, continuity will be \( C^2 \). If tangency is requested at the point, continuity will be \( C^1 \). If Periodicity is requested, the curve will be closed and the junction will be the first point given. The curve will then have a continuity of \( C^1 \) only. This class may be instantiated as follows:

```
Geom2dAPI_Interpolate
(const Handle_TColgp_HArray1OfPnt2d& Points,
 const Standard_Boolean PeriodicFlag,
 const Standard_Real Tolerance);

Geom2dAPI_Interpolate Interp(Points, Standard_False,
                             Precision::Confusion());
```

It is possible to call the BSpline curve from the object defined above it.
Handle(Geom2d_BSplineCurve) C = Interp.Curve();

Note that the Handle(Geom2d_BSplineCurve) operator has been redefined by the method Curve(). Consequently, it is unnecessary to pass via the construction of an intermediate object of the Geom2dAPI_Interpolate type and the following syntax is correct.

Handle(Geom2d_BSplineCurve) C =
Geom2dAPI_Interpolate(Points,
    Standard_False,
    Precision::Confusion());

GeomAPI_Interpolate

This class may be instantiated as follows:

GeomAPI_Interpolate
    (const Handle_TColgp_HArray1OfPnt& Points,
    const Standard_Boolean PeriodicFlag,
    const Standard_Real Tolerance);

GeomAPI_Interpolate Interp(Points, Standard_False,
    Precision::Confusion());

It is possible to call the BSpline curve from the object defined above it.

Handle(Geom_BSplineCurve) C = Interp.Curve();

Note that the Handle(Geom_BSplineCurve) operator has been redefined by the method Curve(). Thus, it is unnecessary to pass via the construction of an intermediate object of the GeomAPI_Interpolate type and the following syntax is correct.

Handle(Geom_BSplineCurve) C = GeomAPI_Interpolate(Points, Standard_False, 1.0e-7);

Boundary conditions may be imposed with the method Load.
GeomAPI_Interpolate AnInterpolator (Points, Standard_False, 1.0e-5);
AnInterpolator.Load (StartingTangent, EndingTangent);
Lines and Circles from Constraints

Types of constraints

The algorithms for construction of 2D circles or lines can be described with numeric or geometric constraints in relation to other curves.

These constraints can impose the following:

- the radius of a circle,
- the angle that a straight line makes with another straight line,
- the tangency of a straight line or circle in relation to a curve,
- the passage of a straight line or circle through a point,
- the circle with center in a point or curve.

For example, these algorithms enable to easily construct a circle of a given radius, centered on a straight line and tangential to another circle.

The implemented algorithms are more complex than those provided by the Direct Constructions component for building 2D circles or lines.

The expression of a tangency problem generally leads to several results, according to the relative positions of the solution and the circles or straight lines in relation to which the tangency constraints are expressed. For example, consider the following case of a circle of a given radius (a small one) which is tangential to two secant circles C1 and C2:
This diagram clearly shows that there are 8 possible solutions.

In order to limit the number of solutions, we can try to express the relative position of the required solution in relation to the circles to which it is tangential. For example, if we specify that the solution is inside the circle C1 and outside the circle C2, only two solutions referenced 3 and 4 on the diagram respond to the problem posed.

These definitions are very easy to interpret on a circle, where it is easy to identify the interior and exterior sides. In fact, for any kind of curve the interior is defined as the left-hand side of the curve in relation to its orientation.

This technique of qualification of a solution, in relation to the curves to which it is tangential, can be used in all algorithms for constructing a circle or a straight line by geometric constraints. Four qualifiers are used:

- **Enclosing** – the solution(s) must enclose the argument;
- **Enclosed** – the solution(s) must be enclosed by the argument;
- **Outside** – the solution(s) and the argument must be external to one another;
- **Unqualified** – the relative position is not qualified, i.e. all solutions apply.

It is possible to create expressions using the qualifiers, for example:
This expression finds all circles of radius $Rad$, which are tangent to both circle $C1$ and $C2$, while $C1$ is outside and $C2$ is inside.

**Available types of lines and circles**

The following analytic algorithms using value-handled entities for creation of 2D lines or circles with geometric constraints are available:

- circle tangent to three elements (lines, circles, curves, points),
- circle tangent to two elements and having a radius,
- circle tangent to two elements and centered on a third element,
- circle tangent to two elements and centered on a point,
- circle tangent to one element and centered on a second,
- bisector of two points,
- bisector of two lines,
- bisector of two circles,
- bisector of a line and a point,
- bisector of a circle and a point,
- bisector of a line and a circle,
- line tangent to two elements (points, circles, curves),
- line tangent to one element and parallel to a line,
- line tangent to one element and perpendicular to a line,
- line tangent to one element and forming angle with a line.

**Exterior/Interior**

It is not hard to define the interior and exterior of a circle. As is shown in the following diagram, the exterior is indicated by the sense of the binormal, that is to say the right side according to the sense of traversing the circle. The left side is therefore the interior (or "material").
Exterior/Interior of a Circle

By extension, the interior of a line or any open curve is defined as the left side according to the passing direction, as shown in the following diagram:

Exterior/Interior of a Line and a Curve

Orientation of a Line

It is sometimes necessary to define in advance the sense of travel along a line to be created. This sense will be from first to second argument.

The following figure shows a line, which is first tangent to circle C1 which is interior to the line, and then passes through point P1.
Line tangent to two circles

The following four diagrams illustrate four cases of using qualifiers in the creation of a line. The fifth shows the solution if no qualifiers are given.

**Example 1 Case 1**


Syntax:

```cpp
GccAna_Lin2d2Tan
    Solver(GccEnt::Outside(C1),
            GccEnt::Outside(C2),
            Tolerance);
```

**Example 1 Case 2**
Both circles enclosed

Constraints: Tangent and Including C1. Tangent and Including C2.

Syntax:

```cpp
GccAna_Lin2d2Tan
    Solver(GccEnt::Enclosing(C1),
            GccEnt::Enclosing(C2),
            Tolerance);
```

Example 1 Case 3

C1 enclosed and C2 outside


Syntax:

```cpp
GccAna_Lin2d2Tan
    Solver(GccEnt::Enclosing(C1),
            GccEnt::Outside(C2),
            Tolerance);
```
Example 1 Case 4

C1 outside and C2 enclosed


Syntax:

```cpp
GccAna_Lin2d2Tan
    Solver(GccEnt::Outside(C1),
          GccEnt::Enclosing(C2),
          Tolerance);
```

Example 1 Case 5

Without qualifiers

Constraints: Tangent and Undefined with respect to C1. Tangent and Undefined with respect to C2.

Syntax:
Circle of given radius tangent to two circles

The following four diagrams show the four cases in using qualifiers in the creation of a circle.

Example 2 Case 1

![Diagram showing both solutions outside]

Both solutions outside


Syntax:

```cpp
GccAna_Circ2d2TanRad
Solver(GccEnt::Outside(C1),
     GccEnt::Outside(C2),
     Rad, Tolerance);
```

Example 2 Case 2
Constraints: Tangent and Exterior to C1. Tangent and Included by C2.

Syntax:

```cpp
GccAna_Circ2d2TanRad
 Solver(GccEnt::Outside(C1),
        GccEnt::Enclosed(C2), Rad, Tolerance);
```

**Example 2 Case 3**


Syntax:

```cpp
GccAna_Circ2d2TanRad
 Solver(GccEnt::Outside(C1),
        GccEnt::Enclosing(C2), Rad, Tolerance);
```
Example 2 Case 4


Syntax:

```cpp
GccAna_Circ2d2TanRad
    Solver(GccEnt::Enclosing(C1),
           GccEnt::Enclosing(C2), Rad, Tolerance);
```

Example 2 Case 5

The following syntax will give all the circles of radius Rad, which are tangent to C1 and C2 without discrimination of relative position:

```cpp
GccAna_Circ2d2TanRad
    Solver(GccEnt::Unqualified(C1),
           GccEnt::Unqualified(C2), Rad, Tolerance);
```

Types of algorithms

OCCT implements several categories of algorithms:

- **Analytic** algorithms, where solutions are obtained by the resolution of an equation, such algorithms are used when the geometries which are worked on (tangency arguments, position of the center, etc.) are points, lines or circles;
- **Geometric** algorithms, where the solution is generally obtained by
calculating the intersection of parallel or bisecting curves built from geometric arguments;
- **Iterative** algorithms, where the solution is obtained by a process of iteration.

For each kind of geometric construction of a constrained line or circle, OCCT provides two types of access:

- algorithms from the package *Geom2dGcc* automatically select the algorithm best suited to the problem, both in the general case and in all types of specific cases; the used arguments are *Geom2d* objects, while the computed solutions are *gp* objects;
- algorithms from the package *GccAna* resolve the problem analytically, and can only be used when the geometries to be worked on are lines or circles; both the used arguments and the computed solutions are *gp* objects.

The provided algorithms compute all solutions, which correspond to the stated geometric problem, unless the solution is found by an iterative algorithm.

Iterative algorithms compute only one solution, closest to an initial position. They can be used in the following cases:

- to build a circle, when an argument is more complex than a line or a circle, and where the radius is not known or difficult to determine: this is the case for a circle tangential to three geometric elements, or tangential to two geometric elements and centered on a curve;
- to build a line, when a tangency argument is more complex than a line or a circle.

Qualified curves (for tangency arguments) are provided either by:

- the *GccEnt* package, for direct use by *GccAna* algorithms, or
- the *Geom2dGcc* package, for general use by *Geom2dGcc* algorithms.

The *GccEnt* and *Geom2dGcc* packages also provide simple functions for building qualified curves in a very efficient way.

The *GccAna* package also provides algorithms for constructing bisecting
loci between circles, lines or points. Bisecting loci between two geometric objects are such that each of their points is at the same distance from the two geometric objects. They are typically curves, such as circles, lines or conics for GccAna algorithms. Each elementary solution is given as an elementary bisecting locus object (line, circle, ellipse, hyperbola, parabola), described by the GccInt package.

Note: Curves used by GccAna algorithms to define the geometric problem to be solved, are 2D lines or circles from the gp package: they are not explicitly parameterized. However, these lines or circles retain an implicit parameterization, corresponding to that which they induce on equivalent Geom2d objects. This induced parameterization is the one used when returning parameter values on such curves, for instance with the functions Tangency1, Tangency2, Tangency3, Intersection2 and CenterOn3 provided by construction algorithms from the GccAna or Geom2dGcc packages.
Curves and Surfaces from Constraints

The Curves and Surfaces from Constraints component groups together high level functions used in 2D and 3D geometry for:

- creation of faired and minimal variation 2D curves
- construction of ruled surfaces
- construction of pipe surfaces
- filling of surfaces
- construction of plate surfaces
- extension of a 3D curve or surface beyond its original bounds.

OPEN CASCADE company also provides a product known as Surfaces from Scattered Points, which allows constructing surfaces from scattered points. This algorithm accepts or constructs an initial B-Spline surface and looks for its deformation (finite elements method) which would satisfy the constraints. Using optimized computation methods, this algorithm is able to construct a surface from more than 500,000 points.

SSP product is not supplied with Open CASCADE Technology, but can be purchased separately.

Faired and Minimal Variation 2D Curves

Elastic beam curves have their origin in traditional methods of modeling applied in boat-building, where a long thin piece of wood, a lathe, was forced to pass between two sets of nails and in this way, take the form of a curve based on the two points, the directions of the forces applied at those points, and the properties of the wooden lathe itself.

Maintaining these constraints requires both longitudinal and transversal forces to be applied to the beam in order to compensate for its internal elasticity. The longitudinal forces can be a push or a pull and the beam may or may not be allowed to slide over these fixed points.

Batten Curves

The class FairCurve_Batten allows producing faired curves defined on
the basis of one or more constraints on each of the two reference points. These include point, angle of tangency and curvature settings. The following constraint orders are available:

- 0 the curve must pass through a point
- 1 the curve must pass through a point and have a given tangent
- 2 the curve must pass through a point, have a given tangent and a given curvature.

Only 0 and 1 constraint orders are used. The function Curve returns the result as a 2D BSpline curve.

**Minimal Variation Curves**

The class `FairCurve_MinimalVariation` allows producing curves with minimal variation in curvature at each reference point. The following constraint orders are available:

- 0 the curve must pass through a point
- 1 the curve must pass through a point and have a given tangent
- 2 the curve must pass through a point, have a given tangent and a given curvature.

Constraint orders of 0, 1 and 2 can be used. The algorithm minimizes tension, sagging and jerk energy.

The function Curve returns the result as a 2D BSpline curve.

If you want to give a specific length to a batten curve, use:

```java
b.SetSlidingFactor(L / b.SlidingOfReference())
```

where `b` is the name of the batten curve object

Free sliding is generally more aesthetically pleasing than constrained sliding. However, the computation can fail with values such as angles greater than $\pi/2$ because in this case the length is theoretically infinite.

In other cases, when sliding is imposed and the sliding factor is too large, the batten can collapse.
The constructor parameters, *Tolerance* and *NbIterations*, control how precise the computation is, and how long it will take.

**Ruled Surfaces**

A ruled surface is built by ruling a line along the length of two curves.

**Creation of Bezier surfaces**

The class *GeomFill_BezierCurves* allows producing a Bezier surface from contiguous Bezier curves. Note that problems may occur with rational Bezier Curves.

**Creation of BSpline surfaces**

The class *GeomFill_BSplineCurves* allows producing a BSpline surface from contiguous BSpline curves. Note that problems may occur with rational BSplines.

**Pipe Surfaces**

The class *GeomFill_Pipe* allows producing a pipe by sweeping a curve (the section) along another curve (the path). The result is a BSpline surface.

The following types of construction are available:

- pipes with a circular section of constant radius,
- pipes with a constant section,
- pipes with a section evolving between two given curves.

**Filling a contour**

It is often convenient to create a surface from some curves, which will form the boundaries that define the new surface. This is done by the class *GeomFill_ConstrainedFilling*, which allows filling a contour defined by three or four curves as well as by tangency constraints. The resulting surface is a BSpline.
A case in point is the intersection of two fillets at a corner. If the radius of the fillet on one edge is different from that of the fillet on another, it becomes impossible to sew together all the edges of the resulting surfaces. This leaves a gap in the overall surface of the object which you are constructing.

Intersecting filleted edges with differing radiiuses

These algorithms allow you to fill this gap from two, three or four curves. This can be done with or without constraints, and the resulting surface will be either a Bezier or a B-Spline surface in one of a range of filling styles.

Creation of a Boundary

The class GeomFill_SimpleBound allows you defining a boundary for the
surface to be constructed.

Creation of a Boundary with an adjoining surface

The class `GeomFill_BoundWithSurf` allows defining a boundary for the surface to be constructed. This boundary will already be joined to another surface.

Filling styles

The enumerations `FillingStyle` specify the styles used to build the surface. These include:

- *Stretch* – the style with the flattest patches
- *Coons* – a rounded style with less depth than *Curved*
- *Curved* – the style with the most rounded patches.

Intersecting filleted edges with different radii leave a gap filled by a surface

Plate surfaces

In CAD, it is often necessary to generate a surface which has no exact mathematical definition, but which is defined by respective constraints. These can be of a mathematical, a technical or an aesthetic order.

Essentially, a plate surface is constructed by deforming a surface so that it conforms to a given number of curve or point constraints. In the figure below, you can see four segments of the outline of the plane, and a point
which have been used as the curve constraints and the point constraint respectively. The resulting surface can be converted into a BSpline surface by using the function MakeApprox.

The surface is built using a variational spline algorithm. It uses the principle of deformation of a thin plate by localised mechanical forces. If not already given in the input, an initial surface is calculated. This corresponds to the plate prior to deformation. Then, the algorithm is called to calculate the final surface. It looks for a solution satisfying constraints and minimizing energy input.

Surface generated from two curves and a point

The package GeomPlate provides the following services for creating
surfaces respecting curve and point constraints:

**Definition of a Framework**

The class *BuildPlateSurface* allows creating a framework to build surfaces according to curve and point constraints as well as tolerance settings. The result is returned with the function *Surface*.

Note that you do not have to specify an initial surface at the time of construction. It can be added later or, if none is loaded, a surface will be computed automatically.

**Definition of a Curve Constraint**

The class *CurveConstraint* allows defining curves as constraints to the surface, which you want to build.

**Definition of a Point Constraint**

The class *PointConstraint* allows defining points as constraints to the surface, which you want to build.

**Applying Geom_Surface to Plate Surfaces**

The class *Surface* allows describing the characteristics of plate surface objects returned by *BuildPlateSurface::Surface* using the methods of *Geom_Surface*

**Approximating a Plate surface to a BSpline**

The class *MakeApprox* allows converting a *GeomPlate* surface into a *Geom_BSplineSurface*.
Surface generated from four curves and a point

Let us create a Plate surface and approximate it from a polyline as a curve constraint and a point constraint

```plaintext
Standard_Integer NbCurFront=4,
NbPointConstraint=1;
gp_Pnt P1(0.,0.,0.);
gp_Pnt P2(0.,10.,0.);
gp_Pnt P3(0.,10.,10.);
gp_Pnt P4(0.,0.,10.);
gp_Pnt P5(5.,5.,5.);
BRepBuilderAPI_MakePolygon W;
W.Add(P1);
```
W.Add(P2);
W.Add(P3);
W.Add(P4);
W.Add(P1);

// Initialize a BuildPlateSurface
GeomPlate_BuildPlateSurface BPSurf(3,15,2);

// Create the curve constraints
BRepTools_WireExplorer anExp;
for(anExp.Init(W); anExp.More(); anExp.Next())
{
  TopoDS_Edge E = anExp.Current();
  Handle(BRepAdaptor_HCurve) C = new BRepAdaptor_HCurve();
  C.ChangeCurve().Initialize(E);
  Handle(BRepFill_CurveConstraint) Cont = new BRepFill_CurveConstraint(C,0);
  BPSurf.Add(Cont);
}

// Point constraint
Handle(GeomPlate_PointConstraint) PCont = new GeomPlate_PointConstraint(P5,0);
BPSurf.Add(PCont);

// Compute the Plate surface
BPSurf.Perform();

// Approximation of the Plate surface
Standard_Integer MaxSeg=9;
Standard_Integer MaxDegree=8;
Standard_Integer CritOrder=0;
Standard_Real dmax,Tol;
Handle(GeomPlate_Surface) PSurf = BPSurf.Surface();
dmax = Max(0.0001,10*BPSurf.G0Error());
Tol=0.0001;
GeomPlate_MakeApprox Mapp(PSurf,Tol,MaxSeg,MaxDegree,dmax,CritOrder);
Handle (Geom_Surface) Surf (Mapp.Surface());

// create a face corresponding to the approximated Plate
Surface
Standard_Real Umin, Umax, Vmin, Vmax;
PSurf->Bounds( Umin, Umax, Vmin, Vmax);
BRepBuilderAPI_MakeFace MF(Surf,Umin, Umax, Vmin, Vmax);
Projections

Projections provide for computing the following:

- the projections of a 2D point onto a 2D curve
- the projections of a 3D point onto a 3D curve or surface
- the projection of a 3D curve onto a surface.
- the planar curve transposition from the 3D to the 2D parametric space of an underlying plane and v. s.
- the positioning of a 2D gp object in the 3D geometric space.

Projection of a 2D Point on a Curve

`Geom2dAPI_ProjectPointOnCurve` allows calculation of all normals projected from a point (`gp_Pnt2d`) onto a geometric curve (`Geom2d_Curve`). The calculation may be restricted to a given domain.

The curve does not have to be a `Geom2d_TrimmedCurve`. The algorithm will function with any class inheriting `Geom2d_Curve`. 
The class `Geom2dAPI_ProjectPointOnCurve` may be instantiated as in the following example:

```cpp
gp_Pnt2d P;
Handle(Geom2d_BezierCurve) C = new Geom2d_BezierCurve(args);
Geom2dAPI_ProjectPointOnCurve Projector (P, C);
```

To restrict the search for normals to a given domain \([U1,U2]\), use the following constructor:

```cpp
Geom2dAPI_ProjectPointOnCurve Projector (P, C, U1, U2);
```

Having thus created the `Geom2dAPI_ProjectPointOnCurve` object, we can now interrogate it.

**Calling the number of solution points**

```cpp
Standard_Integer NumSolutions = Projector.NbPoints();
```

**Calling the location of a solution point**

The solutions are indexed in a range from 1 to `Projector.NbPoints()`. The point, which corresponds to a given `Index` may be found:

```cpp
gp_Pnt2d Pn = Projector.Point(Index);
```

**Calling the parameter of a solution point**

For a given point corresponding to a given `Index`:

```cpp
Standard_Real U = Projector.Parameter(Index);
```

This can also be programmed as:

```cpp
Standard_Real U;
Projector.Parameter(Index, U);
```
Calling the distance between the start and end points

We can find the distance between the initial point and a point, which corresponds to the given \textit{Index}:

\begin{verbatim}
Standard_Real D = Projector.Dist\_ace(Index);
\end{verbatim}

Calling the nearest solution point

This class offers a method to return the closest solution point to the starting point. This solution is accessed as follows:

\begin{verbatim}
gp\_Pnt2d P1 = Projector.NearestPoint();
\end{verbatim}

Calling the parameter of the nearest solution point

\begin{verbatim}
Standard_Real U = Projector.LowerDistanceParameter();
\end{verbatim}

Calling the minimum distance from the point to the curve

\begin{verbatim}
Standard_Real D = Projector.LowerDistance();
\end{verbatim}

Redefined operators

Some operators have been redefined to find the closest solution.

\textit{Standard\_Real()} returns the minimum distance from the point to the curve.

\begin{verbatim}
Standard\_Real D = Geom2dAPI\_ProjectPointOnCurve(P,C);
\end{verbatim}

\textit{Standard\_Integer()} returns the number of solutions.

\begin{verbatim}
Standard\_Integer N = Geom2dAPI\_ProjectPointOnCurve (P,C);
\end{verbatim}

\textit{gp\_Pnt2d()} returns the nearest solution point.
Using these operators makes coding easier when you only need the nearest point. Thus:

```cpp
Geom2dAPI_ProjectPointOnCurve Projector (P, C);
gp_Pnt2d P1 = Projector.NearestPoint();
```

can be written more concisely as:

```cpp
gp_Pnt2d P1 = Geom2dAPI_ProjectPointOnCurve (P,C);
```

However, note that in this second case no intermediate `Geom2dAPI_ProjectPointOnCurve` object is created, and thus it is impossible to have access to other solution points.

**Access to lower-level functionalities**

If you want to use the wider range of functionalities available from the `Extrema` package, a call to the `Extrema()` method will return the algorithmic object for calculating extrema. For example:

```cpp
Extrema_ExtPC2d& TheExtrema = Projector.Extrema();
```

**Projection of a 3D Point on a Curve**

The class `GeomAPI_ProjectPointOnCurve` is instantiated as in the following example:

```cpp
gp_Pnt P;
Handle(Geom_BezierCurve) C =
    new Geom_BezierCurve(args);
GeomAPI_ProjectPointOnCurve Projector (P, C);
```

If you wish to restrict the search for normals to the given domain `[U1,U2]`, use the following constructor:

```cpp
GeomAPI_ProjectPointOnCurve Projector (P, C, U1, U2);
```
Having thus created the `GeomAPI_ProjectPointOnCurve` object, you can now interrogate it.

### Calling the number of solution points

```cpp
Standard_Integer NumSolutions = Projector.NbPoints();
```

### Calling the location of a solution point

The solutions are indexed in a range from 1 to `Projector.NbPoints()`. The point, which corresponds to a given index, may be found:

```cpp
gp_Pnt Pn = Projector.Point(Index);
```

### Calling the parameter of a solution point

For a given point corresponding to a given index:

```cpp
Standard_Real U = Projector.Parameter(Index);
```

This can also be programmed as:

```cpp
Standard_Real U;
Projector.Parameter(Index, U);
```

### Calling the distance between the start and end point

The distance between the initial point and a point, which corresponds to a given index, may be found:

```cpp
Standard_Real D = Projector.Distance(Index);
```

### Calling the nearest solution point

This class offers a method to return the closest solution point to the starting point. This solution is accessed as follows:

```cpp
gp_Pnt P1 = Projector.NearestPoint();
```
Calling the parameter of the nearest solution point

```
Standard_Real U = Projector.LowerDistanceParameter();
```

Calling the minimum distance from the point to the curve

```
Standard_Real D = Projector.LowerDistance();
```

Redefined operators

Some operators have been redefined to find the nearest solution.

`Standard_Real()` returns the minimum distance from the point to the curve.

```
Standard_Real D = GeomAPI_ProjectPointOnCurve (P,C);
```

`Standard_Integer()` returns the number of solutions.

```
Standard_Integer N = GeomAPI_ProjectPointOnCurve (P,C);
```

`gp_Pnt2d()` returns the nearest solution point.

```
gp_Pnt P1 = GeomAPI_ProjectPointOnCurve (P,C);
```

Using these operators makes coding easier when you only need the nearest point. In this way,

```
GeomAPI_ProjectPointOnCurve Projector (P, C);
gp_Pnt P1 = Projector.NearestPoint();
```

can be written more concisely as:

```
gp_Pnt P1 = GeomAPI_ProjectPointOnCurve (P,C);
```

In the second case, however, no intermediate `GeomAPI_ProjectPointOnCurve` object is created, and it is impossible to access other solutions points.
Access to lower-level functionalities

If you want to use the wider range of functionalities available from the Extrema package, a call to the `Extrema()` method will return the algorithmic object for calculating the extrema. For example:

```
Extrema_ExtPC& TheExtrema = Projector.Extrema();
```

Projection of a Point on a Surface

The class `GeomAPI_ProjectPointOnSurf` allows calculation of all normals projected from a point from `gp_Pnt` onto a geometric surface from `Geom_Surface`.

Note that the surface does not have to be of `Geom_RectangularTrimmedSurface` type. The algorithm will function with any class inheriting `Geom_Surface`.

`GeomAPI_ProjectPointOnSurf` is instantiated as in the following example:

```
gp_Pnt P;
Handle (Geom_Surface) S = new
    Geom_BezierSurface(args);
GeomAPI_ProjectPointOnSurf Proj (P, S);
```
To restrict the search for normals within the given rectangular domain 
\([U_1, U_2, V_1, V_2]\), use the constructor \(\text{GeomAPI\_ProjectPointOnSurf} \text{ Proj} (P, S, U_1, U_2, V_1, V_2)\)

The values of \(U_1, U_2, V_1\) and \(V_2\) lie at or within their maximum and 
minimum limits, i.e.:

\[
\begin{align*}
U_{\text{min}} & \leq U_1 < U_2 \leq U_{\text{max}} \\
V_{\text{min}} & \leq V_1 < V_2 \leq V_{\text{max}}
\end{align*}
\]

Having thus created the \(\text{GeomAPI\_ProjectPointOnSurf}\) object, you can interrogate it.

**Calling the number of solution points**

\[
\text{Standard\_Integer \ NumSolutions} = \text{Proj.NbPoints();}
\]

**Calling the location of a solution point**

The solutions are indexed in a range from 1 to \(\text{Proj.NbPoints()}\). The point 
corresponding to the given index may be found:

\[
\text{gp\_Pnt \ Pn} = \text{Proj.Point(\text{Index});}
\]

**Calling the parameters of a solution point**

For a given point corresponding to the given index:

\[
\begin{align*}
\text{Standard\_Real \ U, V;}
\text{Proj.Parameters(\text{Index, U, V});}
\end{align*}
\]

**Calling the distance between the start and end point**

The distance between the initial point and a point corresponding to the 
given index may be found:

\[
\text{Standard\_Real \ D} = \text{Projector.Distance(\text{Index});}
\]
Calling the nearest solution point

This class offers a method, which returns the closest solution point to the starting point. This solution is accessed as follows:

```
gp_Pnt P1 = Proj.NearestPoint();
```

Calling the parameters of the nearest solution point

```
Standard_Real U,V;
Proj.LowerDistanceParameters (U, V);
```

Calling the minimum distance from a point to the surface

```
Standard_Real D = Proj.LowerDistance();
```

Redefined operators

Some operators have been redefined to help you find the nearest solution.

`Standard_Real()` returns the minimum distance from the point to the surface.

```
Standard_Real D = GeomAPI_ProjectPointOnSurf (P,S);
```

`Standard_Integer()` returns the number of solutions.

```
Standard_Integer N = GeomAPI_ProjectPointOnSurf (P,S);
```

`gp_Pnt2d()` returns the nearest solution point.

```
gp_Pnt P1 = GeomAPI_ProjectPointOnSurf (P,S);
```

Using these operators makes coding easier when you only need the nearest point. In this way,
GeomAPI_ProjectPointOnSurface Proj (P, S);
gp_Pnt P1 = Proj.NearestPoint();

can be written more concisely as:

gp_Pnt P1 = GeomAPI_ProjectPointOnSurface (P,S);

In the second case, however, no intermediate
GeomAPI_ProjectPointOnSurf object is created, and it is impossible to
access other solution points.

Access to lower-level functionalities

If you want to use the wider range of functionalities available from the
Extrema package, a call to the Extrema() method will return the
algorithmic object for calculating the extrema as follows:

Extrema_ExtPS& TheExtrema = Proj.Extrema();

Switching from 2d and 3d Curves

The To2d and To3d methods are used to:

- build a 2d curve from a 3d Geom_Curve lying on a gp_Pln plane
- build a 3d curve from a Geom2d_Curve and a gp_Pln plane.

These methods are called as follows:

Handle(Geom2d_Curve) C2d = GeomAPI::To2d(C3d, Pln);
Handle(Geom_Curve) C3d = GeomAPI::To3d(C2d, Pln);
Topological Tools

Open CASCADE Technology topological tools provide algorithms to

- Create wires from edges;
- Create faces from wires;
- Compute state of the shape relatively other shape;
- Orient shapes in container;
- Create new shapes from the existing ones;
- Build PCurves of edges on the faces;
- Check the validity of the shapes;
- Take the point in the face;
- Get the normal direction for the face.
Creation of the faces from wireframe model

It is possible to create the planar faces from the arbitrary set of planar edges randomly located in 3D space. This feature might be useful if you need for instance to restore the shape from the wireframe model:

To make the faces from edges it is, firstly, necessary to create planar wires from the given edges and than create planar faces from each wire. The static methods BOPAlgo_Tools::EdgesToWires and BOPAlgo_Tools::WiresToFaces can be used for that:

```cpp
TopoDS_Shape anEdges = ...; /* The input edges */
Standard_Real anAngTol = 1.e-8; /* The angular
tolerance for distinguishing the planes in which
the wires are located */
Standard_Boolean bShared = Standard_False; /* Defines
whether the edges are shared or not */

TopoDS_Shape aWires; /* resulting wires */
Standard_Integer iErr =
    BOPAlgo_Tools::EdgesToWires(anEdges, aWires,
    bShared, anAngTol);
```
if (iErr) {
    cout << "Error: Unable to build wires from given edges\n";
    return;
}

//
TopoDS_Shape aFaces; /* resulting faces */
Standard_Boolean bDone =
    BOPAlgo_Tools::WiresToFaces(aWires, aFaces,
anAngTol);
if (!bDone) {
    cout << "Error: Unable to build faces from wires\n";
    return;
}

These methods can also be used separately:

- **BOPAlgo_Tools::EdgesToWires** allows creating planar wires from edges. The input edges may be not shared, but the output wires will be sharing the coinciding vertices and edges. For this the intersection of the edges is performed. Although, it is possible to skip the intersection stage (if the input edges are already shared) by passing the corresponding flag into the method. The input edges are expected to be planar, but the method does not check it. Thus, if the input edges are not planar, the output wires will also be not planar. In general, the output wires are non-manifold and may contain free vertices, as well as multi-connected vertices.

- **BOPAlgo_Tools::WiresToFaces** allows creating planar faces from the planar wires. In general, the input wires are non-manifold and may be not closed, but should share the coinciding parts. The wires located in the same plane and completely included into other wires will create holes in the faces built from outer wires:
Wireframe model

Two faces (red face has a hole)
Classification of the shapes

The following methods allow classifying the different shapes relatively other shapes:

- The variety of the `BOPTools_AlgoTools::ComputState` methods classify the vertex/edge/face relatively solid;
- `BOPTools_AlgoTools::IsHole` classifies wire relatively face;
- `IntTools_Tools::ClassifyPointByFace` classifies point relatively face.
Orientation of the shapes in the container

The following methods allow reorienting shapes in the containers:

- `BOPTools_AlgTools::OrientEdgesOnWire` correctly orients edges on the wire;
- `BOPTools_AlgTools::OrientFacesOnShell` correctly orients faces on the shell.
Making new shapes

The following methods allow creating new shapes from the existing ones:

- The variety of the `BOPTools_AlgoTools::MakeNewVertex` creates the new vertices from other vertices and edges;
- `BOPTools_AlgoTools::MakeSplitEdge` splits the edge by the given parameters.
Building PCurves

The following methods allow building PCurves of edges on faces:

- `BOPTools_AlgoTools::BuildPCurveForEdgeOnFace` computes PCurve for the edge on the face;
- `BOPTools_AlgoTools::BuildPCurveForEdgeOnPlane` and `BOPTools_AlgoTools::BuildPCurveForEdgesOnPlane` allow building PCurves for edges on the planar face;
- `BOPTools_AlgoTools::AttachExistingPCurve` takes PCurve on the face from one edge and attach this PCurve to other edge coinciding with the first one.
Checking the validity of the shapes

The following methods allow checking the validity of the shapes:

- `BOPTools_AlgoTools::IsMicroEdge` detects the small edges;
- `BOPTools_AlgoTools::ComputeTolerance` computes the correct tolerance of the edge on the face;
- `BOPTools_AlgoTools::CorrectShapeTolerances` and `BOPTools_AlgoTools::CorrectTolerances` allow correcting the tolerances of the sub-shapes.
- `BRepLib::FindValidRange` finds a range of 3d curve of the edge not covered by tolerance spheres of vertices.
Taking a point inside the face

The following methods allow taking a point located inside the face:

- The variety of the `BOPTools_AlgoTools3D::PointNearEdge` allows getting a point inside the face located near the edge;
- `BOPTools_AlgoTools3D::PointInFace` allows getting a point inside the face.
Getting normal for the face

The following methods allow getting the normal direction for the face/surface:

- *BOPTools_AlgTools3D::GetNormalToSurface* computes the normal direction for the surface in the given point defined by UV parameters;
- *BOPTools_AlgTools3D::GetNormalToFaceOnEdge* computes the normal direction for the face in the point located on the edge of the face;
- *BOPTools_AlgTools3D::GetApproxNormalToFaceOnEdge* computes the normal direction for the face in the point located near the edge of the face.
The Topology API

The Topology API of Open CASCADE Technology (OCCT) includes the following six packages:

- BRepAlgoAPI
- BRepBuilderAPI
- BRepFilletAPI
- BRepFeat
- BRepOffsetAPI
- BRepPrimAPI

The classes provided by the API have the following features:

- The constructors of classes provide different construction methods;
- The class retains different tools used to build objects as fields;
- The class provides a casting method to obtain the result automatically with a function-like call.

Let us use the class BRepBuilderAPI_MakeEdge to create a linear edge from two points.

```cpp
gp_Pnt P1(10,0,0), P2(20,0,0);
TopoDS_Edge E = BRepBuilderAPI_MakeEdge(P1,P2);
```

This is the simplest way to create edge E from two points P1, P2, but the developer can test for errors when he is not as confident of the data as in the previous example.

```cpp
#include <gp_Pnt.hxx>
#include <TopoDS_Edge.hxx>
#include <BRepBuilderAPI_MakeEdge.hxx>

void EdgeTest()
{
    gp_Pnt P1;
    gp_Pnt P2;
    BRepBuilderAPI_MakeEdge ME(P1,P2);
    if (!ME.IsDone())
```
In this example an intermediary object ME has been introduced. This can be tested for the completion of the function before accessing the result. More information on error handling in the topology programming interface can be found in the next section.

`BRepBuilderAPI_MakeEdge` provides valuable information. For example, when creating an edge from two points, two vertices have to be created from the points. Sometimes you may be interested in getting these vertices quickly without exploring the new edge. Such information can be provided when using a class. The following example shows a function creating an edge and two vertices from two points.

```cpp
void MakeEdgeAndVertices(const gp_Pnt& P1, const gp_Pnt& P2, TopoDS_Edge& E, TopoDS_Vertex& V1, TopoDS_Vertex& V2) {
    BRepBuilderAPI_MakeEdge ME(P1, P2);
    if (!ME.IsDone()) {
        Standard_DomainError::Raise
        ("MakeEdgeAndVertices::Failed to create an edge");
    }
    E = ME;
    V1 = ME.Vextex1();
    V2 = ME.Vertex2();
}
```

The class `BRepBuilderAPI_MakeEdge` provides two methods `Vertex1` and `Vertex2`, which return two vertices used to create the edge.
How can `BRepBuilderAPI_MakeEdge` be both a function and a class? It can do this because it uses the casting capabilities of C++. The `BRepBuilderAPI_MakeEdge` class has a method called `Edge`; in the previous example the line `E = ME` could have been written.

```plaintext
E = ME.Edge();
```

This instruction tells the C++ compiler that there is an **implicit casting** of a `BRepBuilderAPI_MakeEdge` into a `TopoDS_Edge` using the `Edge` method. It means this method is automatically called when a `BRepBuilderAPI_MakeEdge` is found where a `TopoDS_Edge` is required.

This feature allows you to provide classes, which have the simplicity of function calls when required and the power of classes when advanced processing is necessary. All the benefits of this approach are explained when describing the topology programming interface classes.
Error Handling in the Topology API

A method can report an error in the two following situations:

- The data or arguments of the method are incorrect, i.e. they do not respect the restrictions specified by the methods in its specifications. Typical example: creating a linear edge from two identical points is likely to lead to a zero divide when computing the direction of the line.
- Something unexpected happened. This situation covers every error not included in the first category. Including: interruption, programming errors in the method or in another method called by the first method, bad specifications of the arguments (i.e. a set of arguments that was not expected to fail).

The second situation is supposed to become increasingly exceptional as a system is debugged and it is handled by the exception mechanism. Using exceptions avoids handling error statuses in the call to a method: a very cumbersome style of programming.

In the first situation, an exception is also supposed to be raised because the calling method should have verified the arguments and if it did not do so, there is a bug. For example, if before calling MakeEdge you are not sure that the two points are non-identical, this situation must be tested.

Making those validity checks on the arguments can be tedious to program and frustrating as you have probably correctly surmised that the method will perform the test twice. It does not trust you. As the test involves a great deal of computation, performing it twice is also time-consuming.

Consequently, you might be tempted to adopt the highly inadvisable style of programming illustrated in the following example:

```cpp
#include <Standard_ErrorHandler.hxx>
try {
  TopoDS_Edge E = BRepBuilderAPI_MakeEdge(P1,P2);
  // go on with the edge
}
```
catch {
    // process the error.
}

To help the user, the Topology API classes only raise the exception `StdFail_NotDone`. Any other exception means that something happened which was unforeseen in the design of this API.

The `NotDone` exception is only raised when the user tries to access the result of the computation and the original data is corrupted. At the construction of the class instance, if the algorithm cannot be completed, the internal flag `NotDone` is set. This flag can be tested and in some situations a more complete description of the error can be queried. If the user ignores the `NotDone` status and tries to access the result, an exception is raised.

```cpp
BRepBuilderAPI_MakeEdge ME(P1,P2);
if (!ME.IsDone()) {
    // doing ME.Edge() or E = ME here
    // would raise StdFail_NotDone
    Standard_DomainError::Raise
    ("ProcessPoints::Failed to create an edge");
}
TopoDS_Edge E = ME;
```
Standard Topological Objects

The following standard topological objects can be created:

- Vertices
- Edges
- Faces
- Wires
- Polygonal wires
- Shells
- Solids.

There are two root classes for their construction and modification:

- The deferred class `BRepBuilderAPI_MakeShape` is the root of all `BRepBuilderAPI` classes, which build shapes. It inherits from the class `BRepBuilderAPI_Command` and provides a field to store the constructed shape.
- The deferred class `BRepBuilderAPI_ModifyShape` is used as a root for the shape modifications. It inherits `BRepBuilderAPI_MakeShape` and implements the methods used to trace the history of all sub-shapes.
**Vertex**

`BRepBuilderAPI_MakeVertex` creates a new vertex from a 3D point from `gp`.

```cpp
gp_Pnt P(0, 0, 10);
TopoDS_Vertex V = BRepBuilderAPI_MakeVertex(P);
```

This class always creates a new vertex and has no other methods.
**Edge**

**Basic edge construction method**

Use `BRepBuilderAPI_MakeEdge` to create from a curve and vertices. The basic method constructs an edge from a curve, two vertices, and two parameters.

```
Handle(Geom_Curve) C = ...; // a curve
TopoDS_Vertex V1 = ..., V2 = ...; // two Vertices
Standard_Real p1 = ..., p2 = ..; // two parameters
TopoDS_Edge E =
    BRepBuilderAPI_MakeEdge(C, V1, V2, p1, p2);
```

where `C` is the domain of the edge; `V1` is the first vertex oriented FORWARD; `V2` is the second vertex oriented REVERSED; `p1` and `p2` are the parameters for the vertices `V1` and `V2` on the curve. The default tolerance is associated with this edge.

---

**The curve**

- Must not be a Null Handle.
- If the curve is a trimmed curve, the basis curve is used.
The vertices

- Can be null shapes. When V1 or V2 is Null the edge is open in the corresponding direction and the corresponding parameter p1 or p2 must be infinite (i.e. p1 is RealFirst(), p2 is RealLast()).
- Must be different vertices if they have different 3d locations and identical vertices if they have the same 3d location (identical vertices are used when the curve is closed).

The parameters

- Must be increasing and in the range of the curve, i.e.:

\[ \text{C->FirstParameter()} \leq p1 < p2 \leq \text{C->LastParameter()} \]

- If the parameters are decreasing, the Vertices are switched, i.e. V2 becomes V1 and V1 becomes V2.
- On a periodic curve the parameters p1 and p2 are adjusted by adding or subtracting the period to obtain p1 in the range of the curve and p2 in the range p1 < p2 <= p1 + Period. So on a parametric curve p2 can be greater than the second parameter, see the figure below.
- Can be infinite but the corresponding vertex must be Null (see above).
- The distance between the Vertex 3d location and the point evaluated on the curve with the parameter must be lower than the default precision.

The figure below illustrates two special cases, a semi-infinite edge and an edge on a periodic curve.
Supplementary edge construction methods

There exist supplementary edge construction methods derived from the basic one.

*BRepBuilderAPI_MakeEdge* class provides methods, which are all simplified calls of the previous one:

- The parameters can be omitted. They are computed by projecting the vertices on the curve.
- 3d points (Pnt from gp) can be given in place of vertices. Vertices are created from the points. Giving vertices is useful when creating connected vertices.
- The vertices or points can be omitted if the parameters are given. The points are computed by evaluating the parameters on the curve.
- The vertices or points and the parameters can be omitted. The first and the last parameters of the curve are used.

The five following methods are thus derived from the basic construction:

```
Handle(Geom_Curve) C = ...; // a curve
TopoDS_Vertex V1 = ..., V2 = ...; // two Vertices
Standard_Real p1 = ..., p2 = ..; // two parameters
gp_Pnt P1 = ..., P2 = ...; // two points
TopoDS_Edge E;
// project the vertices on the curve
```
E = BRepBuilderAPI_MakeEdge(C,V1,V2);
// Make vertices from points
E = BRepBuilderAPI_MakeEdge(C,P1,P2,p1,p2);
// Make vertices from points and project them
E = BRepBuilderAPI_MakeEdge(C,P1,P2);
// Computes the points from the parameters
E = BRepBuilderAPI_MakeEdge(C,p1,p2);
// Make an edge from the whole curve
E = BRepBuilderAPI_MakeEdge(C);

Six methods (the five above and the basic method) are also provided for curves from the gp package in place of Curve from Geom. The methods create the corresponding Curve from Geom and are implemented for the following classes:

- `gp_Lin` creates a `Geom_Line`
- `gp_Circ` creates a `Geom_Circle`
- `gp_Elips` creates a `Geom_Ellipse`
- `gp_Hypr` creates a `Geom_Hyperbola`
- `gp_Parab` creates a `Geom_Parabola`

There are also two methods to construct edges from two vertices or two points. These methods assume that the curve is a line; the vertices or points must have different locations.

```
TopoDS_Vertex V1 = ..., V2 = ...; // two Vertices
gp_Pnt P1 = ..., P2 = ...; // two points
TopoDS_Edge E;

// linear edge from two vertices
E = BRepBuilderAPI_MakeEdge(V1,V2);

// linear edge from two points
E = BRepBuilderAPI_MakeEdge(P1,P2);
```

**Other information and error status**

The class `BRepBuilderAPI_MakeEdge` can provide extra information and return an error status.

If `BRepBuilderAPI_MakeEdge` is used as a class, it can provide two
vertices. This is useful when the vertices were not provided as arguments, for example when the edge was constructed from a curve and parameters. The two methods `Vertex1` and `Vertex2` return the vertices. Note that the returned vertices can be null if the edge is open in the corresponding direction.

The `Error` method returns a term of the `BRepBuilderAPI_EdgeError` enumeration. It can be used to analyze the error when `IsDone` method returns False. The terms are:

- **EdgeDone** – No error occurred, `IsDone` returns True.
- **PointProjectionFailed** – No parameters were given, but the projection of the 3D points on the curve failed. This happens if the point distance to the curve is greater than the precision.
- **ParameterOutOfRange** – The given parameters are not in the range `C->FirstParameter()`, `C->LastParameter()`
- **DifferentPointsOnClosedCurve** – The two vertices or points have different locations but they are the extremities of a closed curve.
- **PointWithInfiniteParameter** – A finite coordinate point was associated with an infinite parameter (see the Precision package for a definition of infinite values).
- **DifferentsPointAndParameter** – The distance of the 3D point and the point evaluated on the curve with the parameter is greater than the precision.
- **LineThroughIdenticPoints** – Two identical points were given to define a line (construction of an edge without curve), `gp::Resolution` is used to test confusion.

The following example creates a rectangle centered on the origin of dimensions H, L with fillets of radius R. The edges and the vertices are stored in the arrays `theEdges` and `theVertices`. We use class `Array1OfShape` (i.e. not arrays of edges or vertices). See the image below.
Creating a Wire

```cpp
#include <BRepBuilderAPI_MakeEdge.hxx>
#include <TopoDS_Shape.hxx>
#include <gp_Circ.hxx>
#include <gp.hxx>
#include <TopoDS_Wire.hxx>
#include <TopTools_Array1OfShape.hxx>
#include <BRepBuilderAPI_MakeWire.hxx>

// Use MakeArc method to make an edge and two vertices
void MakeArc(Standard_Real x, Standard_Real y, Standard_Real R, Standard_Real ang, TopoDS_Shape& E, TopoDS_Shape& V1, TopoDS_Shape& V2) {
    gp_Ax2 Origin = gp::XOY();
    gp_Vec Offset(x, y, 0.);
    Origin.Translate(Offset);
    BRepBuilderAPI_MakeEdge
```
ME(gp_Circ(Origin,R), ang, ang+PI/2);
E = ME;
V1 = ME.Vertex1();
V2 = ME.Vertex2();
}

TopoDS_Wire MakeFilletedRectangle(const Standard_Real H,
const Standard_Real L,
const Standard_Real R)
{
TopTools_Array1OfShape theEdges(1,8);
TopTools_Array1OfShape theVertices(1,8);

// First create the circular edges and the vertices
// using the MakeArc function described above.
void MakeArc(Standard_Real, Standard_Real, Standard_Real, Standard_Real,
TopoDS_Shape&, TopoDS_Shape&, TopoDS_Shape&);

Standard_Real x = L/2 - R, y = H/2 - R;
MakeArc(x,-y,R,3.*PI/2.,theEdges(2),theVertices(2),
theVertices(3));
MakeArc(x,y,R,0.,theEdges(4),theVertices(4),
theVertices(5));
MakeArc(-x,y,R,PI/2.,theEdges(6),theVertices(6),
theVertices(7));
MakeArc(-x,-y,R,PI,theEdges(8),theVertices(8),
theVertices(1));

// Create the linear edges
for (Standard_Integer i = 1; i <= 7; i += 2)
{
theEdges(i) = BRepBuilderAPI_MakeEdge
(TopoDS::Vertex(theVertices(i)),TopoDS::Vertex
(theVertices(i+1)));
}

// Create the wire using the BRepBuilderAPI_MakeWire
BRepBuilderAPI_MakeWire MW;
for (i = 1; i <= 8; i++)
{
    MW.Add(TopoDS::Edge(theEdges(i)));
}
return MW.Wire();
Edge 2D

Use `BRepBuilderAPI_MakeEdge2d` class to make edges on a working plane from 2d curves. The working plane is a default value of the `BRepBuilderAPI` package (see the `Plane` methods).

`BRepBuilderAPI_MakeEdge2d` class is strictly similar to `BRepBuilderAPI_MakeEdge`, but it uses 2D geometry from gp and Geom2d instead of 3D geometry.
**Polygon**

*BRepBuilderAPI_MakePolygon* class is used to build polygonal wires from vertices or points. Points are automatically changed to vertices as in *BRepBuilderAPI_MakeEdge*.

The basic usage of *BRepBuilderAPI_MakePolygon* is to create a wire by adding vertices or points using the Add method. At any moment, the current wire can be extracted. The close method can be used to close the current wire. In the following example, a closed wire is created from an array of points.

```cpp
#include <TopoDS_Wire.hxx>
#include <BRepBuilderAPI_MakePolygon.hxx>
#include <TColgp_Array1OfPnt.hxx>

TopoDS_Wire ClosedPolygon(const TColgp_Array1OfPnt& Points) {
    BRepBuilderAPI_MakePolygon MP;
    for(Standard_Integer i=Points.Lower();i=Points.Upper();i++) {
        MP.Add(Points(i));
    }
    MP.Close();
    return MP;
}
```

Short-cuts are provided for 2, 3, or 4 points or vertices. Those methods have a Boolean last argument to tell if the polygon is closed. The default value is False.

**Two examples:**

Example of a closed triangle from three vertices:

```cpp
TopoDS_Wire W =
```
Example of an open polygon from four points:

```cpp
TopoDS_Wire W =
    BRepBuilderAPI_MakePolygon(P1,P2,P3,P4);
```

`BRepBuilderAPI_MakePolygon` class maintains a current wire. The current wire can be extracted at any moment and the construction can proceed to a longer wire. After each point insertion, the class maintains the last created edge and vertex, which are returned by the methods `Edge`, `FirstVertex` and `LastVertex`.

When the added point or vertex has the same location as the previous one it is not added to the current wire but the most recently created edge becomes Null. The `Added` method can be used to test this condition. The `MakePolygon` class never raises an error. If no vertex has been added, the `Wire` is `Null`. If two vertices are at the same location, no edge is created.
Face

Use `BRepBuilderAPI_MakeFace` class to create a face from a surface and wires. An underlying surface is constructed from a surface and optional parametric values. Wires can be added to the surface. A planar surface can be constructed from a wire. An error status can be returned after face construction.

**Basic face construction method**

A face can be constructed from a surface and four parameters to determine a limitation of the UV space. The parameters are optional, if they are omitted the natural bounds of the surface are used. Up to four edges and vertices are created with a wire. No edge is created when the parameter is infinite.

```cpp
Handle(Geom_Surface) S = ...; // a surface
Standard_Real umin,umax,vmin,vmax; // parameters
TopoDS_Face F =
    BRepBuilderAPI_MakeFace(S,umin,umax,vmin,vmax);
```
To make a face from the natural boundary of a surface, the parameters are not required:

```
Handle(Geom_Surface) S = ...; // a surface
TopoDS_Face F = BRepBuilderAPI_MakeFace(S);
```

Constraints on the parameters are similar to the constraints in `BRepBuilderAPI_MakeEdge`.

- `umin,umax (vmin,vmax)` must be in the range of the surface and must be increasing.
- On a $U (V)$ periodic surface $umin$ and $umax (vmin,vmax)$ are adjusted.
- $umin, umax, vmin, vmax$ can be infinite. There will be no edge in the corresponding direction.
Supplementary face construction methods

The two basic constructions (from a surface and from a surface and parameters) are implemented for all gp package surfaces, which are transformed in the corresponding Surface from Geom.

<table>
<thead>
<tr>
<th>gp package surface</th>
<th>Geom package surface</th>
</tr>
</thead>
<tbody>
<tr>
<td>gp_Pln</td>
<td>Geom_Plane</td>
</tr>
<tr>
<td>gp_Cylinder</td>
<td>Geom_CylindricalSurface</td>
</tr>
<tr>
<td>gp_Cone</td>
<td>creates a</td>
</tr>
<tr>
<td></td>
<td>Geom_ConicalSurface</td>
</tr>
<tr>
<td>gp_Sphere</td>
<td>Geom_SphericalSurface</td>
</tr>
<tr>
<td>gp_Torus</td>
<td>Geom_ToroidalSurface</td>
</tr>
</tbody>
</table>

Once a face has been created, a wire can be added using the Add method. For example, the following code creates a cylindrical surface and adds a wire.

```cpp
gp_Cylinder C = ..; // a cylinder
TopoDS_Wire W = ...; // a wire
BRepBuilderAPI_MakeFace MF(C);
MF.Add(W);
TopoDS_Face F = MF;
```

More than one wire can be added to a face, provided that they do not cross each other and they define only one area on the surface. (Note that this is not checked). The edges on a Face must have a parametric curve description.

If there is no parametric curve for an edge of the wire on the Face it is computed by projection.

For one wire, a simple syntax is provided to construct the face from the surface and the wire. The above lines could be written:

```cpp
TopoDS_Face F = BRepBuilderAPI_MakeFace(C,W);
```

A planar face can be created from only a wire, provided this wire defines a plane. For example, to create a planar face from a set of points you can
use `BRepBuilderAPI_MakePolygon` and `BRepBuilderAPI_MakeFace`.

```cpp
#include <TopoDS_Face.hxx>
#include <TColgp_Array1OfPnt.hxx>
#include <BRepBuilderAPI_MakePolygon.hxx>
#include <BRepBuilderAPI_MakeFace.hxx>

TopoDS_Face PolygonalFace(const TColgp_Array1OfPnt& thePnts)
{
    BRepBuilderAPI_MakePolygon MP;
    for(Standard_Integer i=thePnts.Lower(); i<=thePnts.Upper(); i++)
    {
        MP.Add(thePnts(i));
    }
    MP.Close();
    TopoDS_Face F = BRepBuilderAPI_MakeFace(MP.Wire());
    return F;
}
```

The last use of `MakeFace` is to copy an existing face to add new wires. For example, the following code adds a new wire to a face:

```cpp
TopoDS_Face F = ...; // a face
TopoDS_Wire W = ...; // a wire
F = BRepBuilderAPI_MakeFace(F,W);
```

To add more than one wire an instance of the `BRepBuilderAPI_MakeFace` class can be created with the face and the first wire and the new wires inserted with the `Add` method.

**Error status**

The `Error` method returns an error status, which is a term from the `BRepBuilderAPI_FaceError` enumeration.

- `FaceDone` – no error occurred.
- `NoFace` – no initialization of the algorithm; an empty constructor was
used.

- **NotPlanar** – no surface was given and the wire was not planar.
- **CurveProjectionFailed** – no curve was found in the parametric space of the surface for an edge.
- **ParametersOutOfRange** – the parameters $umin$, $umax$, $vmin$, $vmax$ are out of the surface.
**Wire**

The wire is a composite shape built not from a geometry, but by the assembly of edges. `BRepBuilderAPI_MakeWire` class can build a wire from one or more edges or connect new edges to an existing wire.

Up to four edges can be used directly, for example:

```cpp
TopoDS_Wire W = BRepBuilderAPI_MakeWire(E1,E2,E3,E4);
```

For a higher or unknown number of edges the Add method must be used; for example, to build a wire from an array of shapes (to be edges).

```cpp
TopTools_Array1OfShapes theEdges;
BRepBuilderAPI_MakeWire MW;
for (Standard_Integer i = theEdge.Lower();
i <= theEdges.Upper(); i++)
MW.Add(TopoDS::Edge(theEdges(i));
TopoDS_Wire W = MW;
```

The class can be constructed with a wire. A wire can also be added. In this case, all the edges of the wires are added. For example to merge two wires:

```cpp
#include <TopoDS_Wire.hxx>
#include <BRepBuilderAPI_MakeWire.hxx>

TopoDS_Wire MergeWires (const TopoDS_Wire& W1,
const TopoDS_Wire& W2)
{
BRepBuilderAPI_MakeWire MW(W1);
MW.Add(W2);
return MW;
}
```

`BRepBuilderAPI_MakeWire` class connects the edges to the wire. When a new edge is added if one of its vertices is shared with the wire it is considered as connected to the wire. If there is no shared vertex, the
algorithm searches for a vertex of the edge and a vertex of the wire, which are at the same location (the tolerances of the vertices are used to test if they have the same location). If such a pair of vertices is found, the edge is copied with the vertex of the wire in place of the original vertex. All the vertices of the edge can be exchanged for vertices from the wire. If no connection is found the wire is considered to be disconnected. This is an error.

BRepBuilderAPI_MakeWire class can return the last edge added to the wire (Edge method). This edge can be different from the original edge if it was copied.

The Error method returns a term of the BRepBuilderAPI_WireError enumeration: WireDone – no error occurred. EmptyWire – no initialization of the algorithm, an empty constructor was used. DisconnectedWire – the last added edge was not connected to the wire. NonManifoldWire – the wire with some singularity.
Shell

The shell is a composite shape built not from a geometry, but by the assembly of faces. Use BRepBuilderAPI_MakeShell class to build a Shell from a set of Faces. What may be important is that each face should have the required continuity. That is why an initial surface is broken up into faces.
Solid

The solid is a composite shape built not from a geometry, but by the assembly of shells. Use `BRepBuilderAPI_MakeSolid` class to build a Solid from a set of Shells. Its use is similar to the use of the MakeWire class: shells are added to the solid in the same way that edges are added to the wire in MakeWire.
Object Modification

Transformation

*BRepBuilderAPI_Transform* class can be used to apply a transformation to a shape (see class *gp_Trsf*). The methods have a boolean argument to copy or share the original shape, as long as the transformation allows (it is only possible for direct isometric transformations). By default, the original shape is shared.

The following example deals with the rotation of shapes.

```cpp
TopoDS_Shape myShape1 = ...;
// The original shape 1
TopoDS_Shape myShape2 = ...;
// The original shape2
gp_Trsf T;
T.SetRotation(gp_Ax1(gp_Pnt(0.,0.,0.),gp_Vec(0.,0.,1.)),
2.*PI/5.);
BRepBuilderAPI_Transformation theTrsf(T);
theTrsf.Perform(myShape1);
TopoDS_Shape myNewShape1 = theTrsf.Shape();
theTrsf.Perform(myShape2,Standard_True); // Here duplication is forced
TopoDS_Shape myNewShape2 = theTrsf.Shape()
```
Duplication

Use the `BRepBuilderAPI_Copy` class to duplicate a shape. A new shape is thus created. In the following example, a solid is copied:

```cpp
TopoDS_Solid MySolid;
....// Creates a solid

TopoDS_Solid myCopy = BRepBuilderAPI_Copy(mySolid);
```
Primitives

The *BRepPrimAPI* package provides an API (Application Programming Interface) for construction of primitives such as:

- Boxes;
- Cones;
- Cylinders;
- Prisms.

It is possible to create partial solids, such as a sphere limited by longitude. In real models, primitives can be used for easy creation of specific sub-parts.

- Construction by sweeping along a profile:
  - Linear;
  - Rotational (through an angle of rotation).

Sweeps are objects obtained by sweeping a profile along a path. The profile can be any topology and the path is usually a curve or a wire. The profile generates objects according to the following rules:

- Vertices generate Edges
- Edges generate Faces.
- Wires generate Shells.
- Faces generate Solids.
- Shells generate Composite Solids.

It is not allowed to sweep Solids and Composite Solids. Swept constructions along complex profiles such as BSpline curves also available in the *BRepOffsetAPI* package. This API provides simple, high level calls for the most common operations.
Making Primitives

Box

The class `BRepPrimAPI_MakeBox` allows building a parallelepiped box. The result is either a Shell or a Solid. There are four ways to build a box:

- From three dimensions $dx$, $dy$ and $dz$. The box is parallel to the axes and extends for $[0,dx] [0,dy] [0,dz]$.
- From a point and three dimensions. The same as above but the point is the new origin.
- From two points, the box is parallel to the axes and extends on the intervals defined by the coordinates of the two points.
- From a system of axes `gp_Ax2` and three dimensions. Same as the first way but the box is parallel to the given system of axes.

An error is raised if the box is flat in any dimension using the default precision. The following code shows how to create a box:

```cpp
TopoDS_Solid theBox =
    BRepPrimAPI_MakeBox(10.,20.,30.);
```

The four methods to build a box are shown in the figure:
Wedge

BRepPrimAPI_MakeWedge class allows building a wedge, which is a slanted box, i.e. a box with angles. The wedge is constructed in much the same way as a box i.e. from three dimensions dx,dy,dz plus arguments or from an axis system, three dimensions, and arguments.

The following figure shows two ways to build wedges. One is to add a dimension ltx, which is the length in x of the face at dy. The second is to add xmin, xmax, zmin and zmax to describe the face at dy.
The first method is a particular case of the second with $x_{min} = 0$, $x_{max} = ltx$, $z_{min} = 0$, $z_{max} = dz$. To make a centered pyramid you can use $x_{min} = x_{max} = dx / 2$, $z_{min} = z_{max} = dz / 2$.

**Making Wedges**

**Rotation object**

`BRepPrimAPI_MakeOneAxis` is a deferred class used as a root class for all classes constructing rotational primitives. Rotational primitives are created by rotating a curve around an axis. They cover the cylinder, the cone, the sphere, the torus, and the revolution, which provides all other curves.

The particular constructions of these primitives are described, but they all have some common arguments, which are:

- A system of coordinates, where the Z axis is the rotation axis.
- An angle in the range $[0,2\pi]$. 
- A \( \text{vmin}, \text{vmax} \) parameter range on the curve.

The result of the \text{OneAxis} construction is a \text{Solid}, a \text{Shell}, or a \text{Face}. The face is the face covering the rotational surface. Remember that you will not use the \text{OneAxis} directly but one of the derived classes, which provide improved constructions. The following figure illustrates the \text{OneAxis} arguments.

![MakeOneAxis arguments](image)

### Cylinder

\textit{BRepPrimAPI\_MakeCylinder} class allows creating cylindrical primitives. A cylinder is created either in the default coordinate system or in a given coordinate system \text{gp\_Ax2}. There are two constructions:

- Radius and height, to build a full cylinder.
- Radius, height and angle to build a portion of a cylinder.

The following code builds the cylindrical face of the figure, which is a quarter of cylinder along the \( Y \) axis with the origin at \( X,Y,Z \) the length of \( DY \) and radius \( R \).

```cpp
Standard_Real X = 20, Y = 10, Z = 15, R = 10, DY = 30;
// Make the system of coordinates
gp_Ax2 axes = gp::ZOX();
```
axes.Translate(gp_Vec(X,Y,Z));
TopoDS_Face F = BRepPrimAPI_MakeCylinder(axes,R,DY,PI/2.);

Cone

The following code builds the solid cone of the figure, which is located in the default system with radii $R1$ and $R2$ and height $H$.

```cpp
Standard_Real R1 = 30, R2 = 10, H = 15;
TopoDS_Solid S = BRepPrimAPI_MakeCone(R1,R2,H);
```
**Sphere**

`BRepPrimAPI_MakeSphere` class allows creating spherical primitives. Like a cylinder, a sphere is created either in the default coordinate system or in a given coordinate system `gp_Ax2`. There are four constructions:

- From a radius – builds a full sphere.
- From a radius and an angle – builds a lune (digon).
- From a radius and two angles – builds a wraparound spherical segment between two latitudes. The angles \(a1\) and \(a2\) must follow the relation: \(\pi/2 \leq a1 < a2 \leq \pi/2\).
- From a radius and three angles – a combination of two previous methods builds a portion of spherical segment.

The following code builds four spheres from a radius and three angles.

```cpp
Standard_Real R = 30, ang = PI/2, a1 = -PI/2.3, a2 = PI/4;
TopoDS_Solid S1 = BRepPrimAPI_MakeSphere(R);
TopoDS_Solid S2 = BRepPrimAPI_MakeSphere(R, ang);
TopoDS_Solid S3 = BRepPrimAPI_MakeSphere(R, a1, a2);
TopoDS_Solid S4 = BRepPrimAPI_MakeSphere(R, a1, a2, ang);
```
Note that we could equally well choose to create Shells instead of Solids.

Examples of Spheres

**Torus**

*BRepPrimAPI_MakeTorus* class allows creating toroidal primitives. Like the other primitives, a torus is created either in the default coordinate system or in a given coordinate system *gp_Ax2*. There are four constructions similar to the sphere constructions:

- Two radii – builds a full torus.
- Two radii and an angle – builds an angular torus segment.
- Two radii and two angles – builds a wraparound torus segment between two radial planes. The angles $a_1$, $a_2$ must follow the relation $0 < a_2 - a_1 < 2\pi$. 
- Two radii and three angles – a combination of two previous methods builds a portion of torus segment.

The following code builds four toroidal shells from two radii and three angles.

```plaintext
Standard_Real R1 = 30, R2 = 10, ang = PI, a1 = 0, a2 = PI/2;
TopoDS_Shell S1 = BRepPrimAPI_MakeTorus(R1,R2);
TopoDS_Shell S2 = BRepPrimAPI_MakeTorus(R1,R2,ang);
TopoDS_Shell S3 = BRepPrimAPI_MakeTorus(R1,R2,a1,a2);
TopoDS_Shell S4 = BRepPrimAPI_MakeTorus(R1,R2,a1,a2,ang);
```

Note that we could equally well choose to create Solids instead of Shells.

**Revolution**
**BRepPrimAPI_MakeRevolution** class allows building a uniaxial primitive from a curve. As other uniaxial primitives it can be created in the default coordinate system or in a given coordinate system.

The curve can be any **Geom_Curve**, provided it is planar and lies in the same plane as the Z-axis of local coordinate system. There are four modes of construction:

- From a curve, use the full curve and make a full rotation.
- From a curve and an angle of rotation.
- From a curve and two parameters to trim the curve. The two parameters must be growing and within the curve range.
- From a curve, two parameters, and an angle. The two parameters must be growing and within the curve range.
Sweeping: Prism, Revolution and Pipe

Sweeping

Sweeps are the objects you obtain by sweeping a profile along a path. The profile can be of any topology. The path is usually a curve or a wire. The profile generates objects according to the following rules:

- Vertices generate Edges
- Edges generate Faces.
- Wires generate Shells.
- Faces generate Solids.
- Shells generate Composite Solids

It is forbidden to sweep Solids and Composite Solids. A Compound generates a Compound with the sweep of all its elements.

Generating a sweep

BRepPrimAPI_MakeSweep class is a deferred class used as a root of the following sweep classes:

- BRepPrimAPI_MakePrism – produces a linear sweep
- BRepPrimAPI_MakeRevol – produces a rotational sweep
- BRepPrimAPI_MakePipe – produces a general sweep.
Prism

*BRepPrimAPI_MakePrism* class allows creating a linear *prism* from a shape and a vector or a direction.

- A vector allows creating a finite prism;
- A direction allows creating an infinite or semi-infinite prism. The semi-infinite or infinite prism is toggled by a Boolean argument. All constructors have a boolean argument to copy the original shape or share it (by default).

The following code creates a finite, an infinite and a semi-infinite solid using a face, a direction and a length.

```plaintext
TopoDS_Face F = ..; // The swept face
gp_Dir direc(0,0,1);
Standard_Real l = 10;
// create a vector from the direction and the length
gp_Vec v = direc;
v *= l;
TopoDS_Solid P1 = BRepPrimAPI_MakePrism(F,v);
// finite
TopoDS_Solid P2 = BRepPrimAPI_MakePrism(F,direc);
// infinite
TopoDS_Solid P3 =
    BRepPrimAPI_MakePrism(F,direc,Standard_False);
// semi-infinite
```
Rotational Sweep

*BRepPrimAPI_MakeRevol* class allows creating a rotational sweep from a shape, an axis (gp_Ax1), and an angle. The angle has a default value of 2*PI* which means a closed revolution.

*BRepPrimAPI_MakeRevol* constructors have a last argument to copy or share the original shape. The following code creates a full and a partial rotation using a face, an axis and an angle.

```plaintext
TopoDS_Face F = ...; // the profile
gp_Ax1 axis(gp_Pnt(0,0,0),gp_Dir(0,0,1));
Standard_Real ang = PI/3;
TopoDS_Solid R1 = BRepPrimAPI_MakeRevol(F,axis);
```
// Full revol
TopoDS_Solid R2 = BRepPrimAPI_MakeRevol(F,axis,ang);
Boolean Operations

Boolean operations are used to create new shapes from the combinations of two shapes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuse</td>
<td>all points in S1 or S2</td>
</tr>
<tr>
<td>Common</td>
<td>all points in S1 and S2</td>
</tr>
<tr>
<td>Cut S1 by S2</td>
<td>all points in S1 and not in S2</td>
</tr>
</tbody>
</table>

From the viewpoint of Topology these are topological operations followed by blending (putting fillets onto edges created after the topological operation).
Topological operations are the most convenient way to create real industrial parts. As most industrial parts consist of several simple elements such as gear wheels, arms, holes, ribs, tubes and pipes. It is usually easy to create those elements separately and then to combine them by Boolean operations in the whole final part.

See **Boolean Operations** for detailed documentation.
Input and Result Arguments

Boolean Operations have the following types of the arguments and produce the following results:

- For arguments having the same shape type (e.g. SOLID / SOLID) the type of the resulting shape will be a COMPOUND, containing shapes of this type;
- For arguments having different shape types (e.g. SHELL / SOLID) the type of the resulting shape will be a COMPOUND, containing shapes of the type that is the same as that of the low type of the argument. Example: For SHELL/SOLID the result is a COMPOUND of SHELLs.
- For arguments with different shape types some of Boolean Operations can not be done using the default implementation, because of a non-manifold type of the result. Example: the FUSE operation for SHELL and SOLID can not be done, but the CUT operation can be done, where SHELL is the object and SOLID is the tool.
- It is possible to perform Boolean Operations on arguments of the COMPOUND shape type. In this case each compound must not be heterogeneous, i.e. it must contain equidimensional shapes (EDGES or/and WIREs, FACEs or/and SHELLs, SOLIDs). SOLIDs inside the COMPOUND must not contact (intersect or touch) each other. The same condition should be respected for SHELLs or FACEs, WIREs or EDGES.
- Boolean Operations for COMPSOLID type of shape are not supported.
Implementation

*BRepAlgoAPI_BooleanOperation* class is the deferred root class for Boolean operations.

**Fuse**

*BRepAlgoAPI_Fuse* performs the Fuse operation.

```
TopoDS_Shape A = ..., B = ...;
TopoDS_Shape S = BRepAlgoAPI_Fuse(A, B);
```

**Common**

*BRepAlgoAPI_Common* performs the Common operation.

```
TopoDS_Shape A = ..., B = ...;
TopoDS_Shape S = BRepAlgoAPI_Common(A, B);
```

**Cut**

*BRepAlgoAPI_Cut* performs the Cut operation.

```
TopoDS_Shape A = ..., B = ...;
TopoDS_Shape S = BRepAlgoAPI_Cut(A, B);
```

**Section**

*BRepAlgoAPI_Section* performs the section, described as a *TopoDS_Compound* made of *TopoDS_Edge*.
Section operation

TopoDS_Shape A = ..., TopoDS_ShapeB = ...;
TopoDS_Shape S = BRepAlgoAPI_Section(A,B);
Fillets and Chamfers

This library provides algorithms to make fillets and chamfers on shape edges. The following cases are addressed:

- Corners and apexes with different radii;
- Corners and apexes with different concavity.

If there is a concavity, both surfaces that need to be extended and those, which do not, are processed.
Fillets

Fillet on shape

A fillet is a smooth face replacing a sharp edge.

*BRepFilletAPI_MakeFillet* class allows filleting a shape.

To produce a fillet, it is necessary to define the filleted shape at the construction of the class and add fillet descriptions using the *Add* method.

A fillet description contains an edge and a radius. The edge must be shared by two faces. The fillet is automatically extended to all edges in a smooth continuity with the original edge. It is not an error to add a fillet twice, the last description holds.

![Filleting two edges using radii r1 and r2.](image)

In the following example a filleted box with dimensions a,b,c and radius r is created.

Constant radius

```cpp
#include <TopoDS_Shape.hxx>
#include <TopoDS.hxx>
```
```cpp
#include <BRepPrimAPI_MakeBox.hxx>
#include <TopoDS_Solid.hxx>
#include <BRepFilletAPI_MakeFillet.hxx>
#include <TopExp_Explorer.hxx>

TopoDS_Shape FilletedBox(const Standard_Real a, const Standard_Real b, const Standard_Real c, const Standard_Real r)
{
    TopoDS_Solid Box = BRepPrimAPI_MakeBox(a, b, c);
    BRepFilletAPI_MakeFillet MF(Box);

    // add all the edges to fillet
    TopExp_Explorer ex(Box, TopAbs_EDGE);
    while (ex.More())
    {
        MF.Add(r, TopoDS::Edge(ex.Current()));
        ex.Next();
    }
    return MF.Shape();
}
```

Fillet with constant radius

Changing radius
void CSampleTopologicalOperationsDoc::OnEvolvedblend1()
{
    TopoDS_Shape theBox = BRepPrimAPI_MakeBox(200, 200, 200);

    BRepFilletAPI_MakeFillet Rake(theBox);
    ChFi3d_FilletShape FSh = ChFi3d_Rational;
    Rake.SetFilletShape(FSh);

    TColgp_Array1OfPnt2d ParAndRad(1, 6);
    ParAndRad(1).SetCoord(0., 10.);
    ParAndRad(1).SetCoord(50., 20.);
    ParAndRad(1).SetCoord(70., 20.);
    ParAndRad(1).SetCoord(130., 60.);
    ParAndRad(1).SetCoord(160., 30.);
    ParAndRad(1).SetCoord(200., 20.);

    TopExp_Explorer ex(theBox, TopAbs_EDGE);
    Rake.Add(ParAndRad, TopoDS::Edge(ex.Current()));
    TopoDS_Shape evolvedBox = Rake.Shape();
}
Chamfer

A chamfer is a rectilinear edge replacing a sharp vertex of the face.

The use of `BRepFilletAPI_MakeChamfer` class is similar to the use of `BRepFilletAPI_MakeFillet`, except for the following:

- The surfaces created are ruled and not smooth.
- The `Add` syntax for selecting edges requires one or two distances, one edge and one face (contiguous to the edge).

\[
\text{Add}(\text{dist}, \ E, \ F) \\
\text{Add}(\text{d1, d2, E, F}) \text{ with d1 on the face F.}
\]
Fillets on a planar face

*BRepFilletAPI_MakeFillet2d* class allows constructing fillets and chamfers on planar faces. To create a fillet on planar face: define it, indicate, which vertex is to be deleted, and give the fillet radius with *AddFillet* method.

A chamfer can be calculated with *AddChamfer* method. It can be described by

- two edges and two distances
- one edge, one vertex, one distance and one angle. Fillets and chamfers are calculated when addition is complete.

If face F2 is created by 2D fillet and chamfer builder from face F1, the builder can be rebuilt (the builder recovers the status it had before deletion). To do so, use the following syntax:

```cpp
BRepFilletAPI_MakeFillet2d builder;
builder.Init(F1,F2);
```
#include "BRepPrimAPI_MakeBox.hxx"
#include "TopoDS_Shape.hxx"
#include "TopExp_Explorer.hxx"
#include "BRepFilletAPI_MakeFillet2d.hxx"
#include "TopoDS.hxx"
#include "TopoDS_Solid.hxx"

TopoDS_Shape FilletFace(const Standard_Real a,
                        const Standard_Real b,
                        const Standard_Real c,
                        const Standard_Real r)
{
    TopoDS_Solid Box = BRepPrimAPI_MakeBox (a,b,c);
    TopExp_Explorer ex1(Box,TopAbs_FACE);

    const TopoDS_Face& F =
        TopoDS::Face(ex1.Current());
    BRepFilletAPI_MakeFillet2d MF(F);
    TopExp_Explorer ex2(F, TopAbs_VERTEX);
    while (ex2.More())
    {
        MF.AddFillet(TopoDS::Vertex(ex2.Current()),r);
        ex2.Next();
    }
    // while...
    return MF.Shape();
}
Offsets, Drafts, Pipes and Evolved shapes

These classes provide the following services:

- Creation of offset shapes and their variants such as:
  - Hollowing;
  - Shelling;
  - Lofting;
- Creation of tapered shapes using draft angles;
- Creation of sweeps.
Offset computation

Offset computation can be performed using BRepOffsetAPI_MakeOffsetShape. This class provides API to the two different offset algorithms:

Offset algorithm based on computation of the analytical continuation. Meaning of the parameters can be found in BRepOffsetAPI_MakeOffsetShape::PerformByJoin method description. The list below demonstrates principal scheme of this algorithm:

- At the first step, the offsets are computed.
- After this, the analytical continuations are computed for each offset.
- Pairwise intersection is computed according to the original topological information (sharing, number of neighbors, etc.).
- The offset shape is assembled.

The second algorithm is based on the fact that the offset computation for a single face without continuation can always be built. The list below shows simple offset algorithm:

- Each surface is mapped to its geometric offset surface.
- For each edge, pcurves are mapped to the same pcurves on offset surfaces.
- For each edge, 3d curve is constructed by re-approximation of pcurve on the first offset face.
- Position of each vertex in a result shell is computed as average point of all ends of edges sharing that vertex.
- Tolerances are updated according to the resulting geometry. The possible drawback of the simple algorithm is that it leads, in general case, to tolerance increasing. The tolerances have to grow in order to cover the gaps between the neighbor faces in the output. It should be noted that the actual tolerance growth depends on the offset distance and the quality of joints between the input faces. Anyway the good input shell (smooth connections between adjacent faces) will lead to good result.

The snippets below show usage examples:
BRepOffsetAPI_MakeOffsetShape OffsetMaker1;
// Computes offset shape using analytical
   continuation mechanism.
OffsetMaker1.PerformByJoin(Shape, OffsetValue, Tolerance);
if (OffsetMaker1.IsDone())
   NewShape = OffsetMaker1.Shape();

BRepOffsetAPI_MakeOffsetShape OffsetMaker2;
// Computes offset shape using simple algorithm.
OffsetMaker2.PerformBySimple(Shape, OffsetValue);
if (OffsetMaker2.IsDone())
   NewShape = OffsetMaker2.Shape();
Shelling

Shelling is used to offset given faces of a solid by a specific value. It rounds or intersects adjacent faces along its edges depending on the convexity of the edge. The MakeThickSolidByJoin method of the BRepOffsetAPI_MakeThickSolid takes the solid, the list of faces to remove and an offset value as input.

```cpp
TopoDS_Solid SolidInitial = ...;
Standard_Real Of = ...;
TopTools_ListOfShape LCF;
TopoDS_Shape Result;
Standard_Real Tol = Precision::Confusion();

for (Standard_Integer i = 1 ; i <= n; i++) {
    TopoDS_Face SF = ...;  // a face from SolidInitial
    LCF.Append(SF);
}

BRepOffsetAPI_MakeThickSolid SolidMaker;
SolidMaker.MakeThickSolidByJoin(SolidInitial, LCF, Of, Tol);
if (SolidMaker.IsDone())
    Result = SolidMaker.Shape();
```
Also it is possible to create solid between shell, offset shell. This functionality can be called using 
\textit{BRepOffsetAPI\_MakeThickSolid::MakeThickSolidBySimple} method. The code below shows usage example:

```cpp
BRepOffsetAPI\_MakeThickSolid SolidMaker;
SolidMaker\_\_\_\_MakeThickSolidBySimple(Shell,
    OffsetValue);
if (myDone\_IsDone())
    Solid = SolidMaker\_\_\_\_Shape();
```
Draft Angle

*BRepOffsetAPI_DraftAngle* class allows modifying a shape by applying draft angles to its planar, cylindrical and conical faces.

The class is created or initialized from a shape, then faces to be modified are added; for each face, three arguments are used:

- Direction: the direction with which the draft angle is measured
- Angle: value of the angle
- Neutral plane: intersection between the face and the neutral plane is invariant.

The following code places a draft angle on several faces of a shape; the same direction, angle and neutral plane are used for each face:

```cpp
topoDS_shape myShape = ...  // The original shape
topoTools_listofShape ListOfFace;  // Creation of the list of faces to be modified...

gp_dir Direc(0.,0.,1.);  // Z direction
Standard_Real Angle = 5.*PI/180.;  // 5 degree angle
gp_pln Neutral(gp_pnt(0.,0.,5.), Direc);  // Neutral plane Z=5
BRepOffsetAPI_DraftAngle theDraft(myShape);
topoTools_listIteratorOflistofShape itl;
for (itl.Initialize(ListOfFace); itl.More();
    itl.Next()) {
    theDraft.Add(topoDS::Face(itl.Value()), Direc, Angle, Neutral);
    if (!theDraft.AddDone()) {
        // An error has occurred. The faulty face is
```
given by // ProblematicShape
break;
}
}
if (!theDraft.AddDone()) {
    // An error has occurred
    TopoDS_Face guilty =
    theDraft.ProblematicShape();
    ...
}
theDraft.Build();
if (!theDraft.IsDone()) {
    // Problem encountered during reconstruction
    ...
}
else {
    TopoDS_Shape myResult = theDraft.Shape();
    ...
}
Pipe Constructor

`BRepOffsetAPI_MakePipe` class allows creating a pipe from a Spine, which is a Wire and a Profile which is a Shape. This implementation is limited to spines with smooth transitions, sharp transitions are processed by `BRepOffsetAPI_MakePipeShell`. To be more precise the continuity must be G1, which means that the tangent must have the same direction, though not necessarily the same magnitude, at neighboring edges.

The angle between the spine and the profile is preserved throughout the pipe.

```plaintext
TopoDS_Wire Spine = ...;
TopoDS_Shape Profile = ...;
TopoDS_Shape Pipe =
    BRepOffsetAPI_MakePipe(Spine,Profile);
```

Example of a Pipe
Evolved Solid

*BRepOffsetAPI_MakeEvolved* class allows creating an evolved solid from a Spine (planar face or wire) and a profile (wire).

The evolved solid is an unlooped sweep generated by the spine and the profile.

The evolved solid is created by sweeping the profile’s reference axes on the spine. The origin of the axes moves to the spine, the X axis and the local tangent coincide and the Z axis is normal to the face.

The reference axes of the profile can be defined following two distinct modes:

- The reference axes of the profile are the origin axes.
- The references axes of the profile are calculated as follows:
  - the origin is given by the point on the spine which is the closest to the profile
  - the X axis is given by the tangent to the spine at the point defined above
  - the Z axis is the normal to the plane which contains the spine.

```plaintext
TopoDS_Face Spine = ...;
TopoDS_Wire Profile = ...;
TopoDS_Shape Evol = BRepOffsetAPI_MakeEvolved(Spine,Profile);
```
Sewing

Introduction

Sewing allows creation of connected topology (shells and wires) from a set of separate topological elements (faces and edges). For example, Sewing can be used to create of shell from a compound of separate faces.
Shapes with partially shared edges

It is important to distinguish between sewing and other procedures, which modify the geometry, such as filling holes or gaps, gluing, bending curves and surfaces, etc.

Sewing does not change geometrical representation of the shapes. Sewing applies to topological elements (faces, edges) which are not connected but can be connected because they are geometrically coincident: it adds the information about topological connectivity. Already connected elements are left untouched in case of manifold sewing.
Let us define several terms:

- **Floating edges** do not belong to any face;
- **Free boundaries** belong to one face only;
- **Shared edges** belong to several faces, (i.e. two faces in a manifold topology).
- **Sewn faces** should have edges shared with each other.
- **Sewn edges** should have vertices shared with each other.
Sewing Algorithm

The sewing algorithm is one of the basic algorithms used for shape processing, therefore its quality is very important.

Sewing algorithm is implemented in the class `BRepBuilder_Sewing`. This class provides the following methods:

- loading initial data for global or local sewing;
- setting customization parameters, such as special operation modes, tolerances and output results;
- applying analysis methods that can be used to obtain connectivity data required by external algorithms;
- sewing of the loaded shapes.

Sewing supports working mode with big value tolerance. It is not necessary to repeat sewing step by step while smoothly increasing tolerance.

It is also possible to sew edges to wire and to sew locally separate faces and edges from a shape.

The Sewing algorithm can be subdivided into several independent stages, some of which can be turned on or off using Boolean or other flags.

In brief, the algorithm should find a set of merge candidates for each free boundary, filter them according to certain criteria, and finally merge the found candidates and build the resulting sewn shape.

Each stage of the algorithm or the whole algorithm can be adjusted with the following parameters:

- **Working tolerance** defines the maximal distance between topological elements which can be sewn. It is not ultimate that such elements will be actually sewn as many other criteria are applied to make the final decision.
- **Minimal tolerance** defines the size of the smallest element (edge) in the resulting shape. It is declared that no edges with size less than
this value are created after sewing. If encountered, such topology becomes degenerated.

- **Non-manifold mode** enables sewing of non-manifold topology.

**Example**

To connect a set of $n$ contiguous but independent faces, do the following:

```cpp
BRepBuilderAPI_Sewing Sew;
Sew.Add(Face1);
Sew.Add(Face2);
...
Sew.Add(Facen);
Sew.Perform();
TopoDS_Shape result = Sew.SewedShape();
```

If all faces have been sewn correctly, the result is a shell. Otherwise, it is a compound. After a successful sewing operation all faces have a coherent orientation.
Tolerance Management

To produce a closed shell, Sewing allows specifying the value of working tolerance, exceeding the size of small faces belonging to the shape.

However, if we produce an open shell, it is possible to get incorrect sewing results if the value of working tolerance is too large (i.e. it exceeds the size of faces lying on an open boundary).

The following recommendations can be proposed for tuning-up the sewing process:

- Use as small working tolerance as possible. This will reduce the sewing time and, consequently, the number of incorrectly sewn edges for shells with free boundaries.
- Use as large minimal tolerance as possible. This will reduce the number of small geometry in the shape, both original and appearing after cutting.
- If it is expected to obtain a shell with holes (free boundaries) as a result of sewing, the working tolerance should be set to a value not greater than the size of the smallest element (edge) or smallest distance between elements of such free boundary. Otherwise the free boundary may be sewn only partially.
- It should be mentioned that the Sewing algorithm is unable to understand which small (less than working tolerance) free boundary should be kept and which should be sewn.
Manifold and Non-manifold Sewing

To create one or several shells from a set of faces, sewing merges edges, which belong to different faces or one closed face.

Face sewing supports manifold and non manifold modes. Manifold mode can produce only a manifold shell. Sewing should be used in the non manifold mode to create non manifold shells.

Manifold sewing of faces merges only two nearest edges belonging to different faces or one closed face with each other. Non manifold sewing of faces merges all edges at a distance less than the specified tolerance.

For a complex topology it is advisable to apply first the manifold sewing and then the non manifold sewing a minimum possible working tolerance. However, this is not necessary for a easy topology.

Giving a large tolerance value to non manifold sewing will cause a lot of incorrectness since all nearby geometry will be sewn.
Local Sewing

If a shape still has some non-sewn faces or edges after sewing, it is possible to use local sewing with a greater tolerance.

Local sewing is especially good for open shells. It allows sewing an unwanted hole in one part of the shape and keeping a required hole, which is smaller than the working tolerance specified for the local sewing in the other part of the shape. Local sewing is much faster than sewing on the whole shape.

All preexisting connections of the whole shape are kept after local sewing.

For example, if you want to sew two open shells having coincided free edges using local sewing, it is necessary to create a compound from two shells then load the full compound using method `BRepBuilderAPI_Sewing::Load()`. After that it is necessary to add local sub-shapes, which should be sewn using method `BRepBuilderAPI_Sewing::Add()`. The result of sewing can be obtained using method `BRepBuilderAPI_Sewing::SewedShape()`.

See the example:

```cpp
//initial sewn shapes
TopoDS_Shape aS1, aS2; // these shapes are expected to be well sewn shells
TopoDS_Shape aComp;
BRep_Builder aB;
aB.MakeCompound(aComp);
aB.Add(aComp, aS1);
aB.Add(aComp, aS2);
..........................
aSewing.Load(aComp);

//sub shapes which should be locally sewed
aSewing.Add(aF1);
aSewing.Add(aF2);
```
// performing sewing
aSewing.Perform();
// result shape
TopoDS_Shape aRes = aSewing.SewedShape();
Features

This library, contained in $BRepFeat$ package, is necessary for creation and manipulation of form and mechanical features that go beyond the classical boundary representation of shapes. In that sense, $BRepFeat$ is an extension of $BRepBuilderAPI$ package.
Form Features

The form features are depressions or protrusions including the following types:

- Cylinder;
- Draft Prism;
- Prism;
- Revolved feature;
- Pipe.

Depending on whether you wish to make a depression or a protrusion, you can choose either to remove matter (Boolean cut: Fuse equal to 0) or to add it (Boolean fusion: Fuse equal to 1).

The semantics of form feature creation is based on the construction of shapes:

- for a certain length in a certain direction;
- up to the limiting face;
- from the limiting face at a height;
- above and/or below a plane.

The shape defining the construction of a feature can be either a supporting edge or a concerned area of a face.

In case of supporting edge, this contour can be attached to a face of the basis shape by binding. When the contour is bound to this face, the information that the contour will slide on the face becomes available to the relevant class methods. In case of the concerned area of a face, you can, for example, cut it out and move it at a different height, which defines the limiting face of a protrusion or depression.

Topological definition with local operations of this sort makes calculations simpler and faster than a global operation. The latter would entail a second phase of removing unwanted matter to get the same result.

The Form from BRepFeat package is a deferred class used as a root for form features. It inherits MakeShape from BRepBuilderAPI and provides
implementation of methods keep track of all sub-shapes.

**Prism**

The class `BRepFeat_MakePrism` is used to build a prism interacting with a shape. It is created or initialized from

- a shape (the basic shape),
- the base of the prism,
- a face (the face of sketch on which the base has been defined and used to determine whether the base has been defined on the basic shape or not),
- a direction,
- a Boolean indicating the type of operation (fusion=protrusion or cut=depression) on the basic shape,
- another Boolean indicating if the self-intersections have to be found (not used in every case).

There are six Perform methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Perform(Height)</code></td>
<td>The resulting prism is of the given length.</td>
</tr>
<tr>
<td><code>Perform(Until)</code></td>
<td>The prism is defined between the position of the base and the given face.</td>
</tr>
<tr>
<td><code>Perform(From, Until)</code></td>
<td>The prism is defined between the two faces From and Until.</td>
</tr>
<tr>
<td><code>PerformUntilEnd()</code></td>
<td>The prism is semi-infinite, limited by the actual position of the base.</td>
</tr>
<tr>
<td><code>PerformFromEnd(Until)</code></td>
<td>The prism is semi-infinite, limited by the face Until.</td>
</tr>
<tr>
<td><code>PerformThruAll()</code></td>
<td>The prism is infinite. In the case of a depression, the result is similar to a cut with an infinite prism. In the case of a protrusion, infinite parts are not kept in the result.</td>
</tr>
</tbody>
</table>

**Note** that `Add` method can be used before `Perform` methods to indicate that a face generated by an edge slides onto a face of the base shape.
In the following sequence, a protrusion is performed, i.e., a face of the shape is changed into a prism.

```cpp
TopoDS_Shape Sbase = ...; // an initial shape
TopoDS_Face Fbase = ....; // a base of prism

gp_Dir Extrusion (.,.,.);

// An empty face is given as the sketch face
BRepFeat_MakePrism thePrism(Sbase, Fbase,
  TopoDS_Face(), Extrusion, Standard_True,
  Standard_True);

thePrism, Perform(100.);
if (thePrism.IsDone()) {
  TopoDS_Shape theResult = thePrism;
  ...
}
```

Fusion with MakePrism
Draft Prism

The class `BRepFeat_MakeDPrism` is used to build draft prism topologies interacting with a basis shape. These can be depressions or protrusions. A class object is created or initialized from:

- a shape (basic shape),
- the base of the prism,
- a face (face of sketch on which the base has been defined and used to determine whether the base has been defined on the basic shape or not),
- an angle,
- a Boolean indicating the type of operation (fusion=protrusion or cut=depression) on the basic shape,
- another Boolean indicating if self-intersections have to be found (not used in every case).

Evidently the input data for `MakeDPrism` are the same as for `MakePrism` except for a new parameter `Angle` and a missing parameter `Direction`: the direction of the prism generation is determined automatically as the normal to the base of the prism. The semantics of draft prism feature
creation is based on the construction of shapes:

- along a length
- up to a limiting face
- from a limiting face to a height.

The shape defining construction of the draft prism feature can be either the supporting edge or the concerned area of a face.

In case of the supporting edge, this contour can be attached to a face of the basis shape by binding. When the contour is bound to this face, the information that the contour will slide on the face becomes available to the relevant class methods. In case of the concerned area of a face, it is possible to cut it out and move it to a different height, which will define the limiting face of a protrusion or depression direction.

The *Perform* methods are the same as for *MakePrism*.

```plaintext
TopoDS_Shape S = BRepPrimAPI_MakeBox(400., 250., 300.);
TopExp_Explorer Ex;
Ex.Init(S, TopAbs_FACE);
Ex.Next();
Ex.Next();
Ex.Next();
Ex.Next();
Ex.Next();
TopoDS_Face F = TopoDS::Face(Ex.Current());
Handle(Geom_Surface) surf = BRep_Tool::Surface(F);
gp_Circ2d c(gp_Ax2d(gp_Pnt2d(200., 130.), gp_Dir2d(1., 0.)), 50.);
BRepBuilderAPI_MakeWire MW;
Handle(Geom2d_Curve) aline = new Geom2d_Circle(c);
MW.Add(BRepBuilderAPI_MakeEdge(aline, surf, 0., PI));
MW.Add(BRepBuilderAPI_MakeEdge(aline, surf, PI, 2.*PI));
BRepBuilderAPI_MakeFace MKF;
MKF.Init(surf, Standard_False);
MKF.Add(MW.Wire());
TopoDS_Face FP = MKF.Face();
```
BRepLib::BuildCurves3d(FP);
BRepFeat_MakeDPrism MKDP
    (S,FP,F,10*PI180,Standard_True,
     Standard_True);
MKDP.Perform(200);
TopoDS_Shape res1 = MKDP.Shape();

A tapered prism

Revolution

The class BRepFeat_MakeRevol is used to build a revolution interacting with a shape. It is created or initialized from:

- a shape (the basic shape),
- the base of the revolution,
- a face (the face of sketch on which the base has been defined and used to determine whether the base has been defined on the basic shape or not),
• an axis of revolution,
• a boolean indicating the type of operation (fusion=protrusion or cut=depression) on the basic shape,
• another boolean indicating whether the self-intersections have to be found (not used in every case).

There are four Perform methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform(Angle)</td>
<td>The resulting revolution is of the given magnitude.</td>
</tr>
<tr>
<td>Perform(Until)</td>
<td>The revolution is defined between the actual position of the base and the given face.</td>
</tr>
<tr>
<td>Perform(From, Until)</td>
<td>The revolution is defined between the two faces, From and Until.</td>
</tr>
<tr>
<td>PerformThruAll()</td>
<td>The result is similar to Perform(2*PI).</td>
</tr>
</tbody>
</table>

Note that Add method can be used before Perform methods to indicate that a face generated by an edge slides onto a face of the base shape.

In the following sequence, a face is revolved and the revolution is limited by a face of the base shape.

```plaintext
TopoDS_Shape Sbase = ...; // an initial shape
TopoDS_Face Frevol = ....; // a base of prism
TopoDS_Face FUntil = ....; // face limiting the revol

gp.Dir RevolDir (.,.,.);
gp_Ax1 RevolAx(gp_Pnt(.,.,.), RevolDir);

// An empty face is given as the sketch face

BRepFeat.MakeRevol theRevol(Sbase, Frevol,
                TopoDS_Face(), RevolAx, Standard_True,
                Standard_True);

theRevol.Perform(FUntil);
if (theRevol.IsDone()) {
    TopoDS_Shape theResult = theRevol;
```
Pipe

The class `BRepFeat_MakePipe` constructs compound shapes with pipe features: depressions or protrusions. A class object is created or initialized from:

- a shape (basic shape),
- a base face (profile of the pipe)
- a face (face of sketch on which the base has been defined and used to determine whether the base has been defined on the basic shape or not),
- a spine wire
- a Boolean indicating the type of operation (fusion=protrusion or cut=depression) on the basic shape,
- another Boolean indicating if self-intersections have to be found (not used in every case).

There are three Perform methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Perform()</code></td>
<td>The pipe is defined along the entire path (spine wire)</td>
</tr>
<tr>
<td><code>Perform(Until)</code></td>
<td>The pipe is defined along the path until a given face</td>
</tr>
<tr>
<td><code>Perform(From, Until)</code></td>
<td>The pipe is defined between the two faces From and Until</td>
</tr>
</tbody>
</table>

Let us have a look at the example:

```cpp
TopoDS_Shape S = BRepPrimAPI_MakeBox(400., 250., 300.);
TopExp_Explorer Ex;
Ex.Init(S, TopAbs_FACE);
Ex.Next();
Ex.Next();
TopoDS_Face F1 = TopoDS::Face(Ex.Current());
Handle(Geom_Surface) surf = BRep_Tool::Surface(F1);
```
BRepBuilderAPI_MakeWire MW1;
gp_Pnt2d p1, p2;
p1 = gp_Pnt2d(100.,100.);
p2 = gp_Pnt2d(200.,100.);
Handle(Geom2d_Line) aline =
    GCE2d_MakeLine(p1,p2).Value();
MW1.Add(BRepBuilderAPI_MakeEdge(aline,surf,0.,p1.Distance(p2)));
p1 = p2;
p2 = gp_Pnt2d(150.,200.);
aline = GCE2d_MakeLine(p1,p2).Value();
MW1.Add(BRepBuilderAPI_MakeEdge(aline,surf,0.,p1.Distance(p2)));
p1 = p2;
p2 = gp_Pnt2d(100.,100.);
aline = GCE2d_MakeLine(p1,p2).Value();
MW1.Add(BRepBuilderAPI_MakeEdge(aline,surf,0.,p1.Distance(p2)));
BRepBuilderAPI_MakeFace MKF1;
MKF1.Init(surf,Standard_False);
MKF1.Add(MW1.Wire());
TopoDS_Face FP = MKF1.Face();
BRepLib::BuildCurves3d(FP);
TColgp_Array1OfPnt CurvePoles(1,3);
gp_Pnt pt = gp_Pnt(150.,0.,150.);
CurvePoles(1) = pt;
pt = gp_Pnt(200.,100.,150.);
CurvePoles(2) = pt;
pt = gp_Pnt(150.,200.,150.);
CurvePoles(3) = pt;
Handle(Geom_BezierCurve) curve = new Geom_BezierCurve(CurvePoles);
TopoDS_Edge E = BRepBuilderAPI_MakeEdge(curve);
TopoDS_Wire W = BRepBuilderAPI_MakeWire(E);
BRepFeat_MakePipe MKPipe (S, FP, F1, W, Standard_False, Standard_True);
MKPipe.Perform();
TopoDS_Shape res1 = MKPipe.Shape();
Mechanical Features

Mechanical features include ribs, protrusions and grooves (or slots), depressions along planar (linear) surfaces or revolution surfaces.

The semantics of mechanical features is built around giving thickness to a contour. This thickness can either be symmetrical – on one side of the contour – or dissymmetrical – on both sides. As in the semantics of form features, the thickness is defined by construction of shapes in specific contexts.

The development contexts differ, however, in the case of mechanical features. Here they include extrusion:

- to a limiting face of the basis shape;
- to or from a limiting plane;
- to a height.

A class object is created or initialized from

- a shape (basic shape);
- a wire (base of rib or groove);
- a plane (plane of the wire);
- direction1 (a vector along which thickness will be built up);
- direction2 (vector opposite to the previous one along which thickness will be built up, may be null);
- a Boolean indicating the type of operation (fusion=rib or cut=groove) on the basic shape;
- another Boolean indicating if self-intersections have to be found (not used in every case).

Linear Form

Linear form is implemented in MakeLinearForm class, which creates a rib or a groove along a planar surface. There is one Perform() method, which performs a prism from the wire along the direction1 and direction2 interacting with base shape Sbase. The height of the prism is Magnitude(Direction1)+Magnitude(direction2).
BRepBuilderAPI_MakeWire mkw;
p_Pnt p1 = gp_Pnt(0.,0.,0.);
p_Pnt p2 = gp_Pnt(200.,0.,0.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
p2 = gp_Pnt(200.,0.,50.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
p2 = gp_Pnt(50.,0.,50.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
p2 = gp_Pnt(50.,0.,200.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
p2 = gp_Pnt(0.,0.,200.);
mkw.Add(BRepBuilderAPI_MakeEdge(p1,p2));
p1 = p2;
mkw.Add(BRepBuilderAPI_MakeEdge(p2,gp_Pnt(0.,0.,0.)));

TopoDS_Shape S = BRepBuilderAPI_MakePrism(BRepBuilderAPI_MakeFace(mkw.Wire()),gp_Vec(gp_Pnt(0.,0.,0.),gp_Pnt(0.,100.,0.)));

TopoDS_Wire W = BRepBuilderAPI_MakeWire(BRepBuilderAPI_MakeEdge(gp_Pnt(50.,45.,100.), gp_Pnt(100.,45.,50.)));

Handle(Geom_Plane) aplane = new Geom_Plane(gp_Pnt(0.,45.,0.), gp_Vec(0.,1.,0.));

BRepFeat_MakeLinearForm aform(S, W, aplane, gp_Dir(0.,5.,0.), gp_Dir(0.,-3.,0.), 1, Standard_True);
aform.Perform();
TopoDS_Shape res = aform.Shape();
Creating a rib

**Gluer**

The class `BRepFeat_Gluer` allows gluing two solids along faces. The contact faces of the glued shape must not have parts outside the contact faces of the basic shape. Upon completion the algorithm gives the glued shape with cut out parts of faces inside the shape.

The class is created or initialized from two shapes: the “glued” shape and the basic shape (on which the other shape is glued). Two `Bind` methods are used to bind a face of the glued shape to a face of the basic shape and an edge of the glued shape to an edge of the basic shape.

**Note** that every face and edge has to be bounded, if two edges of two glued faces are coincident they must be explicitly bounded.

```cpp
TopoDS_Shape Sbase = ...; // the basic shape
TopoDS_Shape Sglued = ...; // the glued shape
TopTools_ListOfShape Lfbase;
```
TopTools_ListOfShape Lfglued;
// Determination of the glued faces
...

BRepFeat_Gluer theGlue(Sglue, Sbase);
TopTools_ListIteratorOfListOfShape itlb(Lfbase);
TopTools_ListIteratorOfListOfShape itlg(Lfglued);
for (; itlb.More(); itlb.Next(), itlg.Next()) {
    const TopoDS_Face& f1 = TopoDS::Face(itlg.Value());
    const TopoDS_Face& f2 = TopoDS::Face(itlb.Value());
    theGlue.Bind(f1, f2);
    // for example, use the class FindEdges from LocOpe
    to
    // determine coincident edges
    LocOpe_FindEdge fined(f1, f2);
    for (fined.InitIterator(); fined.More(); fined.Next()) {
        theGlue.Bind(fined.EdgeFrom(), fined.EdgeTo());
    }
}
theGlue.Build();
if (theGlue.IsDone()) {
    TopoDS_Shape theResult = theGlue;
    ...
}

Split Shape

The class `BRepFeat_SplitShape` is used to split faces of a shape into wires or edges. The shape containing the new entities is rebuilt, sharing the unmodified ones.

The class is created or initialized from a shape (the basic shape). Three Add methods are available:

- `Add(Wire, Face)` – adds a new wire on a face of the basic shape.
- `Add(Edge, Face)` – adds a new edge on a face of the basic shape.
- `Add(EdgeNew, EdgeOld)` – adds a new edge on an existing one (the
old edge must contain the new edge).

**Note** The added wires and edges must define closed wires on faces or wires located between two existing edges. Existing edges must not be intersected.

```
TopoDS_Shape Sbase = ...; // basic shape
TopoDS_Face Fsplat = ...; // face of Sbase
TopoDS_Wire Wsplit = ...; // new wire contained in Fsplat
BRepFeat_SplitShape Spls(Sbase);
Spls.Add(Wsplit, Fsplat);
TopoDS_Shape theResult = Spls;
...```
Hidden Line Removal

To provide the precision required in industrial design, drawings need to offer the possibility of removing lines, which are hidden in a given projection.

For this the Hidden Line Removal component provides two algorithms: HLRBRep_Algo and HLRBRep_PolyAlgo.

These algorithms are based on the principle of comparing each edge of the shape to be visualized with each of its faces, and calculating the visible and the hidden parts of each edge. Note that these are not the algorithms used in generating shading, which calculate the visible and hidden parts of each face in a shape to be visualized by comparing each face in the shape with every other face in the same shape. These algorithms operate on a shape and remove or indicate edges hidden by faces. For a given projection, they calculate a set of lines characteristic of the object being represented. They are also used in conjunction with extraction utilities, which reconstruct a new, simplified shape from a selection of the results of the calculation. This new shape is made up of edges, which represent the shape visualized in the projection.

HLRBRep_Algo allows working with the shape itself, whereas HLRBRep_PolyAlgo works with a polyhedral simplification of the shape. When you use HLRBRep_Algo, you obtain an exact result, whereas, when you use HLRBRep_PolyAlgo, you reduce the computation time, but obtain polygonal segments.

No smoothing algorithm is provided. Consequently, a polyhedron will be treated as such and the algorithms will give the results in form of line segments conforming to the mathematical definition of the polyhedron. This is always the case with HLRBRep_PolyAlgo.

HLRBRep_Algo and HLRBRep_PolyAlgo can deal with any kind of object, for example, assemblies of volumes, surfaces, and lines, as long as there are no unfinished objects or points within it.

However, there some restrictions in HLR use:
Points are not processed;
Infinite faces or lines are not processed.

"Sharp, smooth and sewn edges in a simple screw shape",

Outline edges and isoparameters in the same shape
A simple screw shape seen with shading

An extraction showing hidden sharp edges

The following services are related to Hidden Lines Removal:

**Loading Shapes**

To pass a `TopoDS_Shape` to an `HLRBRep_Algo` object, use `HLRBRep_Algo::Add`. With an `HLRBRep_PolyAlgo` object, use `HLRBRep_PolyAlgo::Load`. If you wish to add several shapes, use `Add` or `Load` as often as necessary.
Setting view parameters

*HLRBRep_Algo::Projector* and *HLRBRep_PolyAlgo::Projector* set a projector object which defines the parameters of the view. This object is an *HLRAlgo_Projector*.

Computing the projections

*HLRBRep_PolyAlgo::Update* launches the calculation of outlines of the shape visualized by the *HLRBRep_PolyAlgo* framework.

In the case of *HLRBRep_Algo*, use *HLRBRep_Algo::Update*. With this algorithm, you must also call the method *HLRBRep_Algo::Hide* to calculate visible and hidden lines of the shape to be visualized. With an *HLRBRep_PolyAlgo* object, visible and hidden lines are computed by *HLRBRep_PolyHLRToShape*.

Extracting edges

The classes *HLRBRep_HLRToShape* and *HLRBRep_PolyHLRToShape* present a range of extraction filters for an *HLRBRep_Algo* object and an *HLRBRep_PolyAlgo* object, respectively. They highlight the type of edge from the results calculated by the algorithm on a shape. With both extraction classes, you can highlight the following types of output:

- visible/hidden sharp edges;
- visible/hidden smooth edges;
- visible/hidden sewn edges;
- visible/hidden outline edges.

To perform extraction on an *HLRBRep_PolyHLRToShape* object, use *HLRBRep_PolyHLRToShape::Update* function.

For an *HLRBRep_HLRToShape* object built from an *HLRBRepAlgo* object you can also highlight:

- visible isoparameters and
- hidden isoparameters.
Examples

HLRBRep_Algo

// Build The algorithm object
myAlgo = new HLRBRep_Algo();

// Add Shapes into the algorithm
TopTools_ListIteratorOfListOfShape
    anIterator(myListOfShape);
for (;anIterator.More();anIterator.Next())
    myAlgo-Add(anIterator.Value(),myNbIsos);

// Set The Projector (myProjector is a
    HLRAalgo_Projector)
myAlgo-Projector(myProjector);

// Build HLR
myAlgo->Update();

// Set The Edge Status
myAlgo->Hide();

// Build the extraction object :
HLRBRep_HLRToShape aHLRToShape(myAlgo);

// extract the results :
TopoDS_Shape VCompound =
    aHLRToShape.VCompound();
TopoDS_Shape Rg1LineVCompound =
    aHLRToShape.Rg1LineVCompound();
TopoDS_Shape RgNLineVCompound =
    aHLRToShape.RgNLineVCompound();
TopoDS_Shape OutLineVCompound
aHLRToShape.OutLineVCompound();
TopoDS_Shape IsoLineVCompound = aHLRToShape.IsoLineVCompound();
TopoDS_Shape HCompound = aHLRToShape.HCompound();
TopoDS_Shape Rg1LineHCompound = aHLRToShape.Rg1LineHCompound();
TopoDS_Shape RgNLineHCompound = aHLRToShape.RgNLineHCompound();
TopoDS_Shape OutLineHCompound = aHLRToShape.OutLineHCompound();
TopoDS_Shape IsoLineHCompound = aHLRToShape.IsoLineHCompound();

HLRBRep_PolyAlgo

// Build The algorithm object
myPolyAlgo = new HLRBRep_PolyAlgo();

// Add Shapes into the algorithm
TopTools_ListIteratorOfListOfShape anIterator(myListOfShape);
for (;anIterator.More();anIterator.Next())
myPolyAlgo->Load(anIterator.Value());

// Set The Projector (myProjector is a HLRAlgo_Projector)
myPolyAlgo->Projector(myProjector);

// Build HLR
myPolyAlgo->Update();
// Build the extraction object:
HLRBRep_PolyHLRToShape aPolyHLRToShape;
aPolyHLRToShape.Update(myPolyAlgo);

// extract the results:
TopoDS_Shape VCompound =
aPolyHLRToShape.VCompound();
TopoDS_Shape Rg1LineVCompound =
aPolyHLRToShape.Rg1LineVCompound();
TopoDS_Shape RgNLineVCompound =
aPolyHLRToShape.RgNLineVCompound();
TopoDS_Shape OutLineVCompound =
aPolyHLRToShape.OutLineVCompound();
TopoDS_Shape HCompound =
aPolyHLRToShape.HCompound();
TopoDS_Shape Rg1LineHCompound =
aPolyHLRToShape.Rg1LineHCompound();
TopoDS_Shape RgNLineHCompound =
aPolyHLRToShape.RgNLineHCompound();
TopoDS_Shape OutLineHCompound =
aPolyHLRToShape.OutLineHCompound();
Meshing

Mesh presentations

In addition to support of exact geometrical representation of 3D objects Open CASCADE Technology provides functionality to work with tessellated representations of objects in form of meshes.

Open CASCADE Technology mesh functionality provides:

- data structures to store surface mesh data associated to shapes, and some basic algorithms to handle these data
- data structures and algorithms to build surface triangular mesh from BRep objects (shapes).
- tools to extend 3D visualization capabilities of Open CASCADE Technology with displaying meshes along with associated pre- and post-processor data.

Open CASCADE Technology includes two mesh converters:

- VRML converter translates Open CASCADE shapes to VRML 1.0 files (Virtual Reality Modeling Language). Open CASCADE shapes may be translated in two representations: shaded or wireframe. A shaded representation present shapes as sets of triangles computed by a mesh algorithm while a wireframe representation present shapes as sets of curves.
- STL converter translates Open CASCADE shapes to STL files. STL (STereolitography) format is widely used for rapid prototyping.

Open CASCADE SAS also offers Advanced Mesh Products:

- Open CASCADE Mesh Framework (OMF)
- Express Mesh

Besides, we can efficiently help you in the fields of surface and volume meshing algorithms, mesh optimization algorithms etc. If you require a qualified advice about meshing algorithms, do not hesitate to benefit from the expertise of our team in that domain.
The projects dealing with numerical simulation can benefit from using SALOME - an Open Source Framework for CAE with CAD data interfaces, generic Pre- and Post- F.E. processors and API for integrating F.E. solvers.

Learn more about SALOME platform on http://www.salome-platform.org
Meshing algorithm

The algorithm of shape triangulation is provided by the functionality of `BRepMesh_IncrementalMesh` class, which adds a triangulation of the shape to its topological data structure. This triangulation is used to visualize the shape in shaded mode.

```cpp
const Standard_Real aRadius = 10.0;
const Standard_Real aHeight = 25.0;
BRepPrimAPI_MakeCylinder aCylinder(aRadius, aHeight);
TopoDS_Shape aShape = aCylinder.Shape();

const Standard_Real aLinearDeflection = 0.01;
const Standard_Real anAngularDeflection = 0.5;

BRepMesh_IncrementalMesh aMesh(aShape,
    aLinearDeflection, Standard_False,
    anAngularDeflection);
```

The default meshing algorithm `BRepMesh_IncrementalMesh` has two major options to define triangulation – linear and angular deflections.

At the first step all edges from a face are discretized according to the specified parameters.

At the second step, the faces are tessellated. Linear deflection limits the distance between a curve and its tessellation, whereas angular deflection limits the angle between subsequent segments in a polyline.

![Deflection parameters of BRepMesh_IncrementalMesh algorithm](image)
Linear deflection limits the distance between triangles and the face interior.

Note that if a given value of linear deflection is less than shape tolerance then the algorithm will skip this value and will take into account the shape tolerance.

The application should provide deflection parameters to compute a satisfactory mesh. Angular deflection is relatively simple and allows using a default value (12-20 degrees). Linear deflection has an absolute meaning and the application should provide the correct value for its models. Giving small values may result in a too huge mesh (consuming a lot of memory, which results in a long computation time and slow rendering) while big values result in an ugly mesh.

For an application working in dimensions known in advance it can be reasonable to use the absolute linear deflection for all models. This provides meshes according to metrics and precision used in the application (for example, it it is known that the model will be stored in meters, 0.004 m is enough for most tasks).

However, an application that imports models created in other applications may not use the same deflection for all models. Note that actually this is an abnormal situation and this application is probably just a viewer for CAD models with dimensions varying by an order of magnitude. This
problem can be solved by introducing the concept of a relative linear deflection with some LOD (level of detail). The level of detail is a scale factor for absolute deflection, which is applied to model dimensions.

Meshing covers a shape with a triangular mesh. Other than hidden line removal, you can use meshing to transfer the shape to another tool: a manufacturing tool, a shading algorithm, a finite element algorithm, or a collision algorithm.

You can obtain information on the shape by first exploring it. To access triangulation of a face in the shape later, use `BRepTool::Triangulation`. To access a polygon, which is the approximation of an edge of the face, use `BRepTool::PolygonOnTriangulation`. 
Boolean Operations

Table of Contents

¶ Introduction
¶ Overview
¶ Operators
  ¶ Boolean operator
  ¶ General Fuse operator
  ¶ Splitter operator
  ¶ Section operator
¶ Parts of algorithms
¶ Terms and Definitions
¶ Interferences
  ¶ Vertex/Vertex interference
  ¶ Vertex/Edge interference
  ¶ Vertex/Face interference
  ¶ Edge/Edge interference
  ¶ Edge/Face interference
  ¶ Face/Face interference
  ¶ Face/Solid interference
  ¶ Vertex/Solid interference
  ¶ Edge/Solid interference
  ¶ Face/Solid interference
Solid/Soild
Interference
Computation
Order
Results
Paves
Pave Blocks
Shrunk Range
Common Blocks
FacInfo
Data Structure
Arguments
Shapes
Interferences
Pave, PaveBlock and
CommonBlock
Points and Curves
FacInfo
Root Classes
Class
BOPAlgo_Options
Class BOPAlgo_Alg
Intersection Part
Initialization
Compute
Vertex/Vertex
Interferences
Compute Vertex/Edge
Interferences
Update Pave Blocks
Compute Edge/Edge
Interferences
Compute Vertex/Face
Interferences
Compute Edge/Face
Interferences
Build Split Edges
Compute Face/Face
Interferences
Build Section Edges
Build P-Curves
Process Degenerated Edges
General description of the Building Part
General Fuse Algorithm
Arguments
Results
Examples
Case 1: Three edges intersecting at a point
Case 2: Two wires and an edge
Case 3: An edge intersecting with a face
Case 4: An edge lying on a face
Case 5: An edge and a shell
Case 6: A wire and a shell
Case 7: Three faces
Case 8: A face and a shell
Case 9: A shell and a solid
Case 10: A compound and a solid
Class BOPAlgo_Builder
Fields
Initialization
Build Images for Vertices
Build Result of Type Vertex
Build Images for Edges
Build Result of Type Edge
Build Images for Wires
Build Result of Type Wire
Build Images for Faces
Build Result of Type Face
Build Images for Shells
Build Result of Type Shell
Build Images for Solids
Build Result of Type Solid
Build Images for Type CompSolid
Build Result of Type Compsolid
Build Images for Compounds
Build Result of Type Compound
Post-Processing

Splitter Algorithm
Arguments
Results
Usage
API
DRAW
Examples
Example 1
Example 2
Example 3

Boolean Operations Algorithm

Arguments

Results. General Rules

Examples

Case 1: Two Vertices

Case 2: A Vertex and an Edge

Case 3: A Vertex and a Face

Case 4: A Vertex and a Solid

Case 5: Two edges intersecting at one point

Case 6: Two edges having a common block

Case 7: An Edge and a Face intersecting at a point

Case 8: A Face and an Edge that have a common block

Case 9: An Edge and a Solid intersecting at a point

Case 10: An Edge and a Solid that have a common block

Case 11: Two intersecting faces

Case 12: Two faces that have
Case 23: A Shell and a Wire cut
Case 24: Two Wires that have overlapping edges.

Class BOPAlgo_BOP
Building Draft Result
Building the Result
Section Algorithm
Arguments
Results and general rules
Examples
Case 1: Two Vertices
Case 1: Case 2: A Vertex and an Edge
Case 1: Case 2: A Vertex and a Face
Case 4: A Vertex and a Solid
Case 5: Two edges intersecting at one point
Case 6: Two edges having a common block
Case 7: An Edge and a Face intersecting at a point
Case 8: A Face and an Edge that have a common block
Case 9: An Edge and a Solid intersecting at a point
Case 10: An
Edge and a Solid that have a common block

Case 11: Two intersecting faces

Case 12: Two faces that have a common part

Case 13: Two faces that have overlapping edges

Case 14: Two faces that have overlapping vertices

Case 15: A Face and a Solid that have an intersection curve

Case 16: A Face and a Solid that have overlapping faces.

Case 17: A Face and a Solid that have overlapping edges.

Case 18: A Face and a Solid that have overlapping vertices.

Case 19: Two intersecting Solids

Case 20: Two Solids that have overlapping faces

Case 21: Two
Solids that have overlapping edges

Case 22: Two Solids that have overlapping vertices

Class BOPAlgo_Section

Building the Result

Volume Maker Algorithm

Usage

Examples

Cells Builder algorithm

Usage

Examples

Algorithm Limitations

Arguments

Common requirements

Pure self-interference

Self-interferences due to tolerances

Parametric representation

Using tolerances of vertices to fix gaps

Intersection problems

Pure intersections and common zones

Tolerances and inaccuracies

Acquired Self-interferences
Advanced Options

- Fuzzy Boolean Operation
  - Examples
- Gluing Operation
  - Usage
  - Examples
- Safe processing mode
  - Usage

Errors and warnings reporting system

Usage

- Package BRepAlgoAPI
- Package BOPTest
  - Case 1. General Fuse operation
  - Case 2. Splitting operation
  - Case 3. Common operation
  - Case 4. Fuse operation
  - Case 5. Cut operation
  - Case 6. Section operation
Introduction

This document provides a comprehensive description of the Boolean Operation Algorithm (BOA) as it is implemented in Open CASCADE Technology. The Boolean Component contains:

- General Fuse Operator (GFA),
- Boolean Operator (BOA),
- Section Operator (SA),
- Splitter Operator (SPA).

GFA is the base algorithm for BOA, SPA, SA.

GFA has a history-based architecture designed to allow using OCAF naming functionality. The architecture of GFA is expandable, that allows creating new algorithms basing on it.
Overview

Operators

Boolean operator

The Boolean operator provides the operations (Common, Fuse, Cut) between two groups: Objects and Tools. Each group consists of an arbitrary number of arguments in terms of TopoDS_Shape.

The operator can be represented as:

\[ R_B = B_j (G_1, G_2), \]

where:

- \( R_B \) – result of the operation;
- \( B_j \) – operation of type \( j \) (Common, Fuse, Cut);
- \( G_1 = \{ S_{11}, S_{12} \ldots S_{1n_1} \} \) group of arguments (Objects);
- \( G_2 = \{ S_{21}, S_{22} \ldots S_{2n_2} \} \) group of arguments (Tools);
- \( n_1 \) – Number of arguments in Objects group;
- \( n_2 \) – Number of arguments in Tools group.

Note There is an operation \( \text{Cut21} \), which is an extension for forward Cut operation, i.e \( \text{Cut21}=\text{Cut}(G_2, G_1) \).

For more details see Boolean Operations Algorithm section.

General Fuse operator

The General fuse operator can be applied to an arbitrary number of arguments in terms of TopoDS_Shape.

The GFA operator can be represented as:

\[ R_{GF} = GF (S_1, S_2 \ldots S_n), \]
where

- $R_{GF}$ – result of the operation,
- $S_1$, $S_2$ ... $S_n$ – arguments of the operation,
- $n$ – number of arguments.

The result of the Boolean operator, $R_B$, can be obtained from $R_{GF}$.

For example, for two arguments $S_1$ and $S_2$ the result $R_{GF}$ is

$$R_{GF} = GF (S_1, S_2) = S_{p1} + S_{p2} + S_{p12}$$

This Figure shows that

- $B_{common} (S_1, S_2) = S_{p12}$;
- $B_{cut12} (S_1, S_2) = S_{p1}$;
- $B_{cut21} (S_1, S_2) = S_{p2}$;
- $B_{fuse} (S_1, S_2) = S_{p1} + S_{p2} + S_{p12}$

$$R_{GF}=GF (S_1, S_2) = B_{fuse} = B_{common} + B_{cut12} + B_{cut21}.$$ 

The fact that $R_{GF}$ contains the components of $R_B$ allows considering GFA as the general case of BOA. So it is possible to implement BOA as a subclass of GFA.

For more details see General Fuse Algorithm section.
**Splitter operator**

The Splitter operator can be applied to an arbitrary number of arguments in terms of *TopoDS_Shape*. The arguments are divided into two groups: *Objects* and *Tools*. The result of *SPA* contains all parts that belong to the *Objects* but does not contain the parts that belong to the *Tools*.

The *SPA* operator can be represented as follows:

\[ R_{SPA} = SPA(G_1, G_2) , \]

where:

- \( R_{SPA} \) – is the result of the operation;
- \( G_1 = \{S_{11}, S_{12} \ldots S_{1n_1}\} \) group of arguments (*Objects*);
- \( G_2 = \{S_{21}, S_{22} \ldots S_{2n_2}\} \) group of arguments (*Tools*);
- \( n_1 \) – Number of arguments in *Objects* group;
- \( n_2 \) – Number of arguments in *Tools* group.

The result \( R_{SPA} \) can be obtained from \( R_{GF} \).

For example, for two arguments \( S_1 \) and \( S_2 \) the result \( R_{SPA} \) is

\[ R_{SPA} = SPA(S_1, S_2) = S_{p1} + S_{p12} . \]

In case when all arguments of the *SPA* are *Objects* and there are no *Tools*, the result of *SPA* is equivalent to the result of *GFA*.

For example, when \( G_1 \) consists of shapes \( S_1 \) and \( S_2 \) the result of *SPA* is

\[ R_{SPA} = SPA(S_1, S_2) = S_{p1} + S_{p2} + S_{p12} = GF(S_1, S_2) \]

The fact that the \( R_{GF} \) contains the components of \( R_{SPA} \) allows considering *GFA* as the general case of *SPA*. Thus, it is possible to implement *SPA* as a subclass of *GFA*.

For more details see [Splitter Algorithm](#) section.

**Section operator**

The Section operator *SA* can be applied to arbitrary number of
arguments in terms of TopoDS.Shape. The result of SA contains vertices and edges in accordance with interferences between the arguments. The SA operator can be represented as follows: $R_{SA} = SA(S_1, S_2... S_n)$, where

- $R_{SA}$ – the operation result;
- $S_1, S_2 ... S_n$ – the operation arguments;
- $n$ – the number of arguments.

For more details see Section Algorithm section.
Parts of algorithms

GFA, BOA, SPA and SA have the same Data Structure (DS). The main goal of the Data Structure is to store all necessary information for input data and intermediate results.

The operators consist of two main parts:

- Intersection Part (IP). The main goal of IP is to compute the interferences between sub-shapes of arguments. The IP uses DS to retrieve input data and store the results of intersections.
- Building Part (BP). The main goal of BP is to build required result of an operation. This part also uses DS to retrieve data and store the results.

As it follows from the definition of operator results, the main differences between GFA, BOA, SPA and SA are in the Building Part. The Intersection Part is the same for the algorithms.
Terms and Definitions

This chapter provides the background terms and definitions that are necessary to understand how the algorithms work.
Interferences

There are two groups of interferences.

At first, each shape having a boundary representation (vertex, edge, face) has an internal value of geometrical tolerance. The shapes interfere with each other in terms of their tolerances. The shapes that have a boundary representation interfere when there is a part of 3D space where the distance between the underlying geometry of shapes is less or equal to the sum of tolerances of the shapes. Three types of shapes: vertex, edge and face – produce six types of **BRep interferences**:

- Vertex/Vertex,
- Vertex/Edge,
- Vertex/Face,
- Edge/Edge,
- Edge/Face and
- Face/Face.

At second, there are interferences that occur between a solid $Z_1$ and a shape $S_2$ when $Z_1$ and $S_2$ have no BRep interferences but $S_2$ is completely inside of $Z_1$. These interferences are **Non-BRep interferences**. There are four possible cases:

- Vertex/Solid,
- Edge/Solid,
- Face/Solid and
- Solid/Solid.

**Vertex/Vertex interference**

For two vertices $Vi$ and $Vj$, the distance between their corresponding 3D points is less than the sum of their tolerances $Tol(Vi)$ and $Tol(Vj)$. 
The result is a new vertex \( V_n \) with 3D point \( P_n \) and tolerance value \( Tol(V_n) \).

The coordinates of \( P_n \) and the value \( Tol(V_n) \) are computed as the center and the radius of the sphere enclosing the tolerance spheres of the source vertices \( (V_1, V_2) \).

**Vertex/Edge interference**

For a vertex \( V_i \) and an edge \( E_j \), the distance \( D \) between 3D point of the vertex and its projection on the 3D curve of edge \( E_j \) is less or equal than sum of tolerances of vertex \( Tol(V_i) \) and edge \( Tol(E_j) \).

The result is vertex \( V_i \) with the corresponding tolerance value \( Tol(V_i) = \text{Max}(Tol(V_i), D+Tol(E_j)) \), where \( D = \text{distance} (P_i, PP_i) \); and parameter \( t_i \) of the projected point \( PP_i \) on 3D curve \( C_j \) of edge \( E_j \).
For a vertex $V_i$ and a face $F_j$ the distance $D$ between 3D point of the vertex and its projection on the surface of the face is less or equal than sum of tolerances of the vertex $Tol(V_i)$ and the face $Tol(F_j)$.

$2 \cdot Tol(F_j)$

**Vertex/face interference**

The result is vertex $V_i$ with the corresponding tolerance value $Tol(V_i)=\text{Max}(Tol(V_i), D+Tol(F_j))$, where $D = \text{distance } (P_i, PPi)$ and parameters $u_i, v_i$ of the projected point $PPi$ on surface $S_j$ of face $F_j$.

**Edge/Edge interference**

For two edges $E_i$ and $E_j$ (with the corresponding 3D curves $C_i$ and $C_j$) there are some places where the distance between the curves is less than (or equal to) sum of tolerances of the edges.

Let us examine two cases:

In the first case two edges have one or several common parts of 3D curves in terms of tolerance.
Edge/edge interference: common parts

The results are:

- Parametric range $[t_{i1}, t_{i2}]$ for 3D curve $Ci$ of edge $Ei$.
- Parametric range $[t_{j1}, t_{j2}]$ for 3D curve $Cj$ of edge $Ej$.

In the second case two edges have one or several common points in terms of tolerance.
Edge/edge interference: common points

The result is a new vertex $V_n$ with 3D point $P_n$ and tolerance value $Tol(V_n)$.

The coordinates of $P_n$ and the value $Tol(V_n)$ are computed as the center and the radius of the sphere enclosing the tolerance spheres of the corresponding nearest points $P_i, P_j$ of 3D curves $C_i, C_j$ of source edges $E_i, E_j$.

- Parameter $t_i$ of $P_i$ for the 3D curve $C_i$.
- Parameter $t_j$ of $P_j$ for the 3D curve $C_j$.

Edge/Face interference

For an edge $E_i$ (with the corresponding 3D curve $C_i$) and a face $F_j$ (with the corresponding 3D surface $S_j$) there are some places in 3D space, where the distance between $C_i$ and surface $S_j$ is less than (or equal to) the sum of tolerances of edge $E_i$ and face $F_j$.

Let us examine two cases:

In the first case Edge $E_i$ and Face $F_j$ have one or several common parts in terms of tolerance.
**Edge/face interference: common parts**

The result is a parametric range \([t_{i1}, t_{i2}]\) for the 3D curve \(Ci\) of the edge \(Ei\).

In the second case Edge \(Ei\) and Face \(Fj\) have one or several common points in terms of tolerance.
**Edge/face interference: common points**

The result is a new vertex $V_n$ with 3D point $P_n$ and tolerance value $Tol(V_n)$.

The coordinates of $P_n$ and the value $Tol(V_n)$ are computed as the center and the radius of the sphere enclosing the tolerance spheres of the corresponding nearest points $P_i, P_j$ of 3D curve $C_i$ and surface $S_j$ of source edges $E_i, F_j$.

- Parameter $t_i$ of $P_i$ for the 3D curve $C_i$.
- Parameters $u_i$ and $v_i$ of the projected point $PP_i$ on the surface $S_j$ of the face $F_j$.

**Face/Face Interference**

For a face $F_i$ and a face $F_j$ (with the corresponding surfaces $S_i$ and $S_j$) there are some places in 3D space, where the distance between the surfaces is less than (or equal to) sum of tolerances of the faces.
Face/face interference: common curves

In the first case the result contains intersection curves $C_{ijk}$ ($k = 0, 1, 2… k_N$), where $k_N$ is the number of intersection curves with corresponding values of tolerances $Tol(C_{ijk})$.

Face/face interference: common points

In the second case Face $Fi$ and face $Fj$ have one or several new vertices $V_{ijm}$, where $m=0,1,2, … mN$, $mN$ is the number of intersection points.

The coordinates of a 3D point $P_{ijm}$ and the value $Tol(V_{ijm})$ are computed as the center and the radius of the sphere enclosing the tolerance spheres of the corresponding nearest points $Pi, Pj$ of the surface $Si, Sj$ of source shapes $Fi, Fj$.

- Parameters $u_j, v_j$ belong to point $PPj$ projected on surface $Sj$ of face
$F_j$.
- Parameters $u_i$ and $v_i$ belong to point $PP_i$ projected on surface $S_i$ of face $F_i$.

**Vertex/Solid Interference**

For a vertex $V_i$ and a solid $Z_j$ there is Vertex/Solid interference if the vertex $V_i$ has no BRep interferences with any sub-shape of $Z_j$ and $V_i$ is completely inside the solid $Z_j$.

**Edge/Solid Interference**

For an edge $E_i$ and a solid $Z_j$ there is Edge/Solid interference if the edge $E_i$ and its sub-shapes have no BRep interferences with any sub-shape of $Z_j$ and $E_i$ is completely inside the solid $Z_j$. 
Face/Solid Interference

For a face $F_i$ and a solid $Z_j$ there is Face/Solid interference if the face $F_i$ and its sub-shapes have no BRep interferences with any sub-shape of $Z_j$ and $F_i$ is completely inside the solid $Z_j$. 
Solid/Soild Interference

For a solid $Z_i$ and a solid $Z_j$ there is Solid/Solid interference if the solid $Z_i$ and its sub-shapes have no BRep interferences with any sub-shape of $Z_j$ and $Z_i$ is completely inside the solid $Z_j$.

Computation Order

The interferences between shapes are computed on the basis of increasing of the dimension value of the shape in the following order:

- Vertex/Vertex,
- Vertex/Edge,
- Edge/Edge,
- Vertex/Face,
- Edge/Face,
- Face/Face,
- Vertex/Solid,
- Edge/Solid,
- Face/Solid,
- Solid/Solid.

This order allows avoiding the computation of redundant interferences.
between upper-level shapes $S_i$ and $S_j$ when there are interferences between lower sub-shapes $S_{ik}$ and $S_{jm}$.

**Results**

- The result of the interference is a shape that can be either interfered shape itself (or its part) or a new shape.
- The result of the interference is a shape with the dimension value that is less or equal to the minimal dimension value of interfered shapes. For example, the result of Vertex/Edge interference is a vertex, but not an edge.
- The result of the interference splits the source shapes on the parts each time as it can do that.
**Paves**

The result of interferences of the type Vertex/Edge, Edge/Edge and Edge/Face in most cases is a vertex (new or old) lying on an edge.

The result of interferences of the type Face/Face in most cases is intersection curves, which go through some vertices lying on the faces.

The position of vertex $V_i$ on curve $C$ can be defined by a value of parameter $t_i$ of the 3D point of the vertex on the curve. Pave $PV_i$ on curve $C$ is a structure containing the vertex $V_i$ and correspondent value of the parameter $t_i$ of the 3D point of the vertex on the curve. Curve $C$ can be a 3D or a 2D curve.

Two paves $PV1$ and $PV2$ on the same curve $C$ can be compared using the parameter value

$PV1 > PV2$ if $t1 > t2$

The usage of paves allows binding of the vertex to the curve (or any structure that contains a curve: edge, intersection curve).
Pave Blocks

A set of paves $PVi \; (i=1, 2...nPV)$, where $nPV$ is the number of paves] of curve $C$ can be sorted in the increasing order using the value of parameter $t$ on curve $C$.

A pave block $PBi$ is a part of the object (edge, intersection curve) between neighboring paves.

Any finite source edge $E$ has at least one pave block that contains two paves $Pb$ and $Pe$:

- Pave $Pb$ corresponds to the vertex $Vb$ with minimal parameter $t_b$ on the curve of the edge.
- Pave $Pe$ corresponds to the vertex $Ve$ with maximal parameter $t_e$ on the curve of the edge.
Shrunken Range

Pave block $PV$ of curve $C$ is bounded by vertices $V1$ and $V2$ with tolerance values $Tol(V1)$ and $Tol(V2)$. Curve $C$ has its own tolerance value $Tol(C)$:

- In case of edge, the tolerance value is the tolerance of the edge.
- In case of intersection curve, the tolerance value is obtained from an intersection algorithm.

The theoretical parametric range of the pave block is $[t1C, t2C]$.

The positions of the vertices $V1$ and $V2$ of the pave block can be different. The positions are determined by the following conditions:

\[
\begin{align*}
\text{Distance (P1, P1c) is equal or less than Tol(V1) + Tol(C)} \\
\text{Distance (P2, P2c) is equal or less than Tol(V2) + Tol(C)}
\end{align*}
\]

The Figure shows that each tolerance sphere of a vertex can reduce the parametric range of the pave block to a range $[t1S, t2S]$. The range $[t1S, t2S]$ is the shrunk range of the pave block.

The shrunk range of the pave block is the part of 3D curve that can interfere with other shapes.
Common Blocks

The interferences of the type Edge/Edge, Edge/Face produce results as common parts.

In case of Edge/Edge interference the common parts are pave blocks that have different base edges.

Common Blocks: Edge/Edge interference

If the pave blocks $PB_1, PB_2, \ldots PB_{NbPB}$, where $NbPB$ is the number of pave blocks have the same bounding vertices and geometrically coincide, the pave blocks form common block $CB$.

In case of Edge/Face interference the common parts are pave blocks lying on a face(s).

Common Blocks: Edge/Face interference

If the pave blocks $PBi$ geometrically coincide with a face $Fj$, the pave blocks form common block $CB$. 
In general case a common block $CB$ contains:

- Pave blocks $PB_i \ (i=0,1,2, \ldots \ \text{NbPB})$.
- A set of faces $F_j \ (j=0,1,\ldots \ \text{NbF})$, $\text{NbF}$ – number of faces.
The structure *FaceInfo* contains the following information:

- Pave blocks that have state **In** for the face;
- Vertices that have state **In** for the face;
- Pave blocks that have state **On** for the face;
- Vertices that have state **On** for the face;
- Pave blocks built up from intersection curves for the face;
- Vertices built up from intersection points for the face.

In the figure, for face *F1*:

- Pave blocks that have state **In** for the face: *PB_{in1}.*
- Vertices that have state **In** for the face: *V_{in1}.*
- Pave blocks that have state **On** for the face: *PB_{on11}, PB_{on12}, PB_{on2}, PB_{on31}, PB_{on32}, PB_{on4}.*
- Vertices that have state **On** for the face: *V1, V2, V3, V4, V5, V6.*
- Pave blocks built up from intersection curves for the face: *PB_{sc1}.*
- Vertices built up from intersection points for the face: none
Data Structure

Data Structure (DS) is used to:

- Store information about input data and intermediate results;
- Provide the access to the information;
- Provide the links between the chunks of information.

This information includes:

- Arguments;
- Shapes;
- Interferences;
- Pave Blocks;
- Common Blocks.

Data Structure is implemented in the class \textit{BOPDS\_DS}. 
Arguments

The arguments are shapes (in terms of `TopoDS_Shape`):

- Number of arguments is unlimited.
- Each argument is a valid shape (in terms of `BRepCheck_Analyzer`).
- Each argument can be of one of the following types (see the Table):

<table>
<thead>
<tr>
<th>No</th>
<th>Type</th>
<th>Index of Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>COMPOUND</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>COMPSOLID</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>SOLID</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>SHELL</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>FACE</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>WIRE</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>EDGE</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>VERTEX</td>
<td>7</td>
</tr>
</tbody>
</table>

- The argument of type 0 (`COMPOUND`) can include any number of shapes of an arbitrary type (0, 1...7).
- The argument should not be self-interfered, i.e. all sub-shapes of the argument that have geometrical coincidence through any topological entities (vertices, edges, faces) must share these entities.
- There are no restrictions on the type of underlying geometry of the shapes. The faces or edges of arguments $S_i$ can have underlying geometry of any type supported by Open CASCADE Technology modeling algorithms (in terms of `GeomAbs_CurveType` and `GeomAbs_SurfaceType`).
- The faces or edges of the arguments should have underlying geometry with continuity that is not less than C1.
Shapes

The information about Shapes is stored in structure `BOPDS_ShapeInfo`. The objects of type `BOPDS_ShapeInfo` are stored in the container of array type. The array allows getting the access to the information by an index (DS index). The structure `BOPDS_ShapeInfo` has the following contents:

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>myShape</code></td>
<td>Shape itself</td>
</tr>
<tr>
<td><code>myType</code></td>
<td>Type of shape</td>
</tr>
<tr>
<td><code>myBox</code></td>
<td>3D bounding box of the shape</td>
</tr>
<tr>
<td><code>mySubShapes</code></td>
<td>List of DS indices of sub-shapes</td>
</tr>
<tr>
<td><code>myReference</code></td>
<td>Storage for some auxiliary information</td>
</tr>
<tr>
<td><code>myFlag</code></td>
<td>Storage for some auxiliary information</td>
</tr>
</tbody>
</table>
## Interferences

The information about interferences is stored in the instances of classes that are inherited from class *BOPDS_Interf*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOPDS_Interf</td>
<td>Root class for interference</td>
</tr>
<tr>
<td>Index1</td>
<td>DS index of the shape 1</td>
</tr>
<tr>
<td>Index2</td>
<td>DS index of the shape 2</td>
</tr>
<tr>
<td>BOPDS_InterfVV</td>
<td>Storage for Vertex/Vertex interference</td>
</tr>
<tr>
<td>BOPDS_InterfVE</td>
<td>Storage for Vertex/Edge interference</td>
</tr>
<tr>
<td>myParam</td>
<td>The value of parameter of the point of the vertex on the curve of the edge</td>
</tr>
<tr>
<td>BOPDS_InterfVF</td>
<td>Storage for Vertex/Face interference</td>
</tr>
<tr>
<td>myU, myV</td>
<td>The value of parameters of the point of the vertex on the surface of the face</td>
</tr>
<tr>
<td>BOPDS_InterfEE</td>
<td>Storage for Edge/Edge interference</td>
</tr>
<tr>
<td>myCommonPart</td>
<td>Common part (in terms of IntTools_CommonPart)</td>
</tr>
<tr>
<td>BOPDS_InterfEF</td>
<td>Storage for Edge/Face interference</td>
</tr>
<tr>
<td>myCommonPart</td>
<td>Common part (in terms of IntTools_CommonPart)</td>
</tr>
<tr>
<td>BOPDS_InterfFF</td>
<td>Storage for Face/Face interference</td>
</tr>
<tr>
<td>myTolR3D, myTolR2D</td>
<td>The value of tolerances of curves (points) reached in 3D and 2D</td>
</tr>
<tr>
<td>myCurves</td>
<td>Intersection Curves (in terms of BOPDS_Curve)</td>
</tr>
<tr>
<td>myPoints</td>
<td>Intersection Points (in terms of BOPDS_Point)</td>
</tr>
<tr>
<td>BOPDS_InterfVZ</td>
<td>Storage for Vertex/Solid interference</td>
</tr>
<tr>
<td>BOPDS_InterfEZ</td>
<td>Storage for Edge/Solid interference</td>
</tr>
<tr>
<td>BOPDS_InterfFZ</td>
<td>Storage for Face/Solid interference</td>
</tr>
<tr>
<td>BOPDS_InterfZZ</td>
<td>Storage for Solid/Solid interference</td>
</tr>
</tbody>
</table>

The Figure shows inheritance diagram for *BOPDS_Interf* classes.
Pave, PaveBlock and CommonBlock

The information about the pave is stored in objects of type \textit{BOPDS\_Pave}.

\begin{tabular}{|l|l|}
\hline
\textbf{Name} & \textbf{Contents} \\
\hline
\textit{BOPDS\_Pave} & \\
\textit{myIndex} & DS index of the vertex \\
\textit{myParam} & Value of the parameter of the 3D point of vertex on curve. \\
\hline
\end{tabular}

The information about pave blocks is stored in objects of type \textit{BOPDS\_PaveBlock}.

\begin{tabular}{|l|l|}
\hline
\textbf{Name} & \textbf{Contents} \\
\hline
\textit{BOPDS\_PaveBlock} & \\
\textit{myEdge} & DS index of the edge produced from the pave block \\
\textit{myOriginalEdge} & DS index of the source edge \\
\textit{myPave1} & Pave 1 (in terms of \textit{BOPDS\_Pave}) \\
\textit{myPave2} & Pave 2 (in terms of \textit{BOPDS\_Pave}) \\
\textit{myExtPaves} & The list of paves (in terms of \textit{BOPDS\_Pave}) that is used to store paves lying inside the pave block during intersection process \\
\textit{myCommonBlock} & The reference to common block (in terms of \textit{BOPDS\_CommonBlock}) if the pave block is a common block \\
\textit{myShrunkData} & The shrunk range of the pave block \\
\hline
\end{tabular}

- To be bound to an edge (or intersection curve) the structures of type \textit{BOPDS\_PaveBlock} are stored in one container of list type \textit{(BOPDS\_ListOfPaveBlock)}.
- In case of edge, all the lists of pave blocks above are stored in one container of array type. The array allows getting the access to the information by index of the list of pave blocks for the edge. This
index (if exists) is stored in the field *myReference*.

The information about common block is stored in objects of type *BOPDS_CommonBlock*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>BOPDS_CommonBlock</em></td>
<td></td>
</tr>
<tr>
<td><em>myPaveBlocks</em></td>
<td>The list of pave blocks that are common in terms of <strong>Common Blocks</strong></td>
</tr>
<tr>
<td><em>myFaces</em></td>
<td>The list of DS indices of the faces, on which the pave blocks lie.</td>
</tr>
</tbody>
</table>
Points and Curves

The information about intersection point is stored in objects of type `BOPDS_Point`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BOPDS_Point</code></td>
<td></td>
</tr>
<tr>
<td><code>myPnt</code></td>
<td>3D point</td>
</tr>
<tr>
<td><code>myPnt2D1</code></td>
<td>2D point on the face1</td>
</tr>
<tr>
<td><code>myPnt2D2</code></td>
<td>2D point on the face2</td>
</tr>
</tbody>
</table>

The information about intersection curve is stored in objects of type `BOPDS_Curve`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BOPDS_Curve</code></td>
<td></td>
</tr>
<tr>
<td><code>myCurve</code></td>
<td>The intersection curve (in terms of <code>IntTools_Curve</code> )</td>
</tr>
<tr>
<td><code>myPaveBlocks</code></td>
<td>The list of pave blocks that belong to the curve</td>
</tr>
<tr>
<td><code>myBox</code></td>
<td>The bounding box of the curve (in terms of <code>Bnd_Box</code> )</td>
</tr>
</tbody>
</table>
# FaceInfo

The information about `FaceInfo` is stored in a structure `BOPDS_FaceInfo`. The structure `BOPDS_FaceInfo` has the following contents.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BOPDS_FaceInfo</code></td>
<td></td>
</tr>
<tr>
<td><code>myPaveBlocksIn</code></td>
<td>Pave blocks that have state In for the face</td>
</tr>
<tr>
<td><code>myVerticesIn</code></td>
<td>Vertices that have state In for the face</td>
</tr>
<tr>
<td><code>myPaveBlocksOn</code></td>
<td>Pave blocks that have state On for the face</td>
</tr>
<tr>
<td><code>myVerticesOn</code></td>
<td>Vertices that have state On for the face</td>
</tr>
<tr>
<td><code>myPaveBlocksSc</code></td>
<td>Pave blocks built up from intersection curves for the face</td>
</tr>
<tr>
<td><code>myVerticesSc</code></td>
<td>Vertices built up from intersection points for the face</td>
</tr>
</tbody>
</table>

The objects of type `BOPDS_FaceInfo` are stored in one container of array type. The array allows getting the access to the information by index. This index (if exists) is stored in the field `myReference`. 
Root Classes

Class BOPAlgo_Options

The class *BOPAlgo_Options* provides the following options for the algorithms:

- Set the appropriate memory allocator;
- Check the presence of the Errors and Warnings;
- Turn on/off the parallel processing;
- Set the additional tolerance for the operation;
- Break the operations by user request.
Class BOPAlgo_Algo

The class *BOPAlgo_Algo* provides the base interface for all algorithms:

- Perform the operation;
- Check the input data;
- Check the result.
Intersection Part

Intersection Part (IP) is used to

- Initialize the Data Structure;
- Compute interferences between the arguments (or their sub-shapes);
- Compute same domain vertices, edges;
- Build split edges;
- Build section edges;
- Build p-curves;
- Store all obtained information in DS.

IP is implemented in the class `BOPAlgo_PaveFiller`.

The description provided in the next paragraphs is coherent with the implementation of the method `BOPAlgo_PaveFiller::Perform()`.
## Initialization

The input data for the step is the Arguments. The description of initialization step is shown in the Table.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialization the array of shapes (in terms of \textbf{Shapes}). Filling the array of shapes.</td>
<td>\textit{BOPDS_DS::Init()}</td>
</tr>
<tr>
<td>2</td>
<td>Initialization the array pave blocks (in terms of \textbf{Pave, PaveBlock, CommonBlock})</td>
<td>\textit{BOPDS_DS::Init()}</td>
</tr>
<tr>
<td>3</td>
<td>Initialization of intersection Iterator. The intersection Iterator is the object that computes intersections between sub-shapes of the arguments in terms of bounding boxes. The intersection Iterator provides approximate number of the interferences for given type (in terms of \textbf{Interferences})</td>
<td>\textit{BOPDS_Iterator}</td>
</tr>
<tr>
<td>4</td>
<td>Initialization of intersection Context. The intersection Context is an object that contains geometrical and topological toolkit (classifiers, projectors, etc). The intersection Context is used to cache the tools to increase the algorithm performance.</td>
<td>\textit{IntTools_Context}</td>
</tr>
</tbody>
</table>
Compute Vertex/Vertex Interferences

The input data for this step is the DS after the **Initialization**. The description of this step is shown in the table:

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize array of Vertex/Vertex interferences.</td>
<td><code>BOPAlgo_PaveFiller::PerformVV()</code></td>
</tr>
<tr>
<td>2</td>
<td>Access to the pairs of interfered shapes ((nVi, nVj)k, k=0, 1...nk), where (nVi) and (nVj) are DS indices of vertices (Vi) and (Vj) and (nk) is the number of pairs.</td>
<td><code>BOPDS_Iterator</code></td>
</tr>
<tr>
<td>3</td>
<td>Compute the connexity chains of interfered vertices (nV1C, nV2C... nVnC)k, C=0, 1...nCs), where (nCs) is the number of the connexity chains</td>
<td><code>BOPAlgo_Tools::MakeBlocksCnx()</code></td>
</tr>
<tr>
<td>4</td>
<td>Build new vertices from the chains (VNC). (C=0, 1...nCs).</td>
<td><code>BOPAlgo_PaveFiller::PerformVV()</code></td>
</tr>
<tr>
<td>5</td>
<td>Append new vertices in DS.</td>
<td><code>BOPDS_DS::Append()</code></td>
</tr>
<tr>
<td>6</td>
<td>Append same domain vertices in DS.</td>
<td><code>BOPDS_DS::AddShapeSD()</code></td>
</tr>
<tr>
<td>7</td>
<td>Append Vertex/Vertex interferences in DS.</td>
<td><code>BOPDS_DS::AddInterf()</code></td>
</tr>
</tbody>
</table>

- The pairs of interfered vertices are: \((nV11, nV12), (nV11, nV13), (nV12, nV13), (nV13, nV15), (nV13, nV14), (nV14, nV15), (nV21, nV22), (nV21, nV23), (nV22, nV23);\)
- These pairs produce two chains: \((nV11, nV12, nV13, nV14, nV15)\) and \((nV21, nV22, nV23);\)
- Each chain is used to create a new vertex, \(VN1\) and \(VN2\), correspondingly.
The example of connexity chains of interfered vertices is given in the image:

Connexity chains of interfered vertices
Compute Vertex/Edge Interferences

The input data for this step is the DS after computing Vertex/Vertex interferences.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize array of Vertex/Edge interferences</td>
<td>BOPAlgo_PaveFiller::PerformVE()</td>
</tr>
<tr>
<td>2</td>
<td>Access to the pairs of interfered shapes ((nVi, nEj)k), (k=0, 1...nk), where (nVi) is DS index of vertex (Vi), (nEj) is DS index of edge (Ej) and (nk) is the number of pairs.</td>
<td>BOPDS_Iterator</td>
</tr>
<tr>
<td>3</td>
<td>Compute paves. See Vertex/Edge Interference</td>
<td>BOPInt_Context::ComputeVE()</td>
</tr>
<tr>
<td>4</td>
<td>Initialize pave blocks for the edges (Ej) involved in the interference</td>
<td>BOPDS_DS::ChangePaveBlocks()</td>
</tr>
<tr>
<td>5</td>
<td>Append the paves into the pave blocks in terms of Pave, PaveBlock and CommonBlock</td>
<td>BOPDS_PaveBlock::AppendExtPave()</td>
</tr>
<tr>
<td>6</td>
<td>Append Vertex/Edge interferences in DS</td>
<td>BOPDS_DS::AddInterf()</td>
</tr>
</tbody>
</table>
Update Pave Blocks

The input data for this step is the DS after computing Vertex/Edge Interferences.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Each pave block PB containing internal paves is split by internal paves into new pave blocks $PBN1$, $PBN2$… $PBNn$. PB is replaced by new pave blocks $PBN1$, $PBN2$… $PBNn$ in the DS.</td>
<td>$BOPDS_{DS}::$ $UpdatePaveBlocks()$</td>
</tr>
</tbody>
</table>
## Compute Edge/Edge Interferences

The input data for this step is the DS after updating Pave Blocks.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize array of Edge/Edge interferences</td>
<td><code>BOPAlgo_PaveFiller::PerformEE()</code></td>
</tr>
<tr>
<td>2</td>
<td>Access to the pairs of interfered shapes ((nE_i, nE_j)k, k=0, 1… nk), where (nE_i) is DS index of the edge (E_i), (nE_j) is DS index of the edge (E_j) and (nk) is the number of pairs.</td>
<td><code>BOPDS_Iterator</code></td>
</tr>
<tr>
<td>3</td>
<td>Initialize pave blocks for the edges involved in the interference, if it is necessary.</td>
<td><code>BOPDS_DS::ChangePaveBlocks()</code></td>
</tr>
<tr>
<td>4</td>
<td>Access to the pave blocks of interfered shapes: ((PB_{i1}, PB_{i2}… PB_{iNi})) for edge (E_i) and ((PB_{j1}, PB_{j2}… PB_{jNj})) for edge (E_j)</td>
<td><code>BOPAlgo_PaveFiller::PerformEE()</code></td>
</tr>
<tr>
<td>5</td>
<td>Compute shrunk data for pave blocks in terms of Pave, PaveBlock and CommonBlock, if it is necessary.</td>
<td><code>BOPAlgo_PaveFiller::FillShrunkData()</code></td>
</tr>
<tr>
<td>6</td>
<td>Compute Edge/Edge interference for pave blocks (PB_{ix}) and (PB_{iy}). The result of the computation is a set</td>
<td><code>IntTools_EdgeEdge</code></td>
</tr>
</tbody>
</table>
For each CommonPart of type VERTEX: Create new vertices $V_{Ni}$ ($i = 1, 2..., NbVN$), where $NbVN$ is the number of new vertices. Intersect the vertices $V_{Ni}$ using the steps Initialization and compute Vertex/Vertex interferences as follows: a) create a new object $PFn$ of type $BOPAlgo_PaveFiller$ with its own DS; b) use new vertices $V_{Ni}$ ($i=1, 2..., NbVN$), $NbVN$ as arguments (in terms of $TopoDs_Shape$) of $PFn$; c) invoke method $Perform()$ for $PFn$. The resulting vertices $V_{NXi}$ ($i=1, 2..., NbVNX$), where $NbVNX$ is the number of vertices, are obtained via mapping between $V_{Ni}$ and the results of $PVn$.

For each CommonPart of type EDGE: Compute the coinciding connexity chains of pave blocks
<table>
<thead>
<tr>
<th></th>
<th>(PB1C, PB2C...PNNnC)k, C=0, 1...nCs, where nCs is the number of the connexity chains. Create common blocks (CBc. C=0, 1...nCs) from the chains. Attach the common blocks to the pave blocks.</th>
<th>BOPAlgo_Tools::PerformCommonBlocks()</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>Post-processing. Append the paves of VNXi into the corresponding pave blocks in terms of Pave, PaveBlock and CommonBlock</td>
<td>BOPDS_PaveBlock:: AppendExtPave()</td>
</tr>
<tr>
<td>8</td>
<td>Split common blocks CBc by the paves.</td>
<td>BOPDS_DS:: UpdateCommonBlock()</td>
</tr>
<tr>
<td>9</td>
<td>Append Edge/Edge interferences in the DS.</td>
<td>BOPDS_DS::AddInterf()</td>
</tr>
</tbody>
</table>

The example of coinciding chains of pave blocks is given in the image:
The pairs of coincided pave blocks are: (PB11, PB12), (PB11, PB13), (PB12, PB13), (PB21, PB22), (PB21, PB23), (PB22, PB23).
The pairs produce two chains: (PB11, PB12, PB13) and (PB21, PB22, PB23).
Compute Vertex/Face Interferences

The input data for this step is the DS after computing Edge/Edge interferences.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize array of Vertex/Face interferences</td>
<td><code>BOPAlgo_PaveFiller::PerformVF()</code></td>
</tr>
<tr>
<td>2</td>
<td>Access to the pairs of interfered shapes <code>(nVi, nFj)k, k=0, 1…nk</code>, where</td>
<td><code>BOPDS_Iterator</code></td>
</tr>
<tr>
<td></td>
<td><code>nVi</code> is DS index of the vertex <code>Vi</code>, <code>nFj</code> is DS index of the edge <code>Fj</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>and <code>nk</code> is the number of pairs.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Compute interference</td>
<td><code>BOPInt_Context::ComputeVF()</code></td>
</tr>
<tr>
<td>4</td>
<td>Append Vertex/Face interferences in the DS</td>
<td><code>BOPDS_DS::AddInterf()</code></td>
</tr>
<tr>
<td>5</td>
<td>Repeat steps 2-4 for each new vertex <code>VNXi</code> (<code>i=1, 2…, NbVNX</code>), where</td>
<td><code>BOPAlgo_PaveFiller::TreatVerticesEE()</code></td>
</tr>
<tr>
<td></td>
<td><code>NbVNX</code> is the number of vertices.</td>
<td></td>
</tr>
</tbody>
</table>
## Compute Edge/Face Interferences

The input data for this step is the DS after computing Vertex/Face Interferences.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize array of Edge/Face interferences</td>
<td><code>BOPAlgo_PaveFiller::PerformEF()</code></td>
</tr>
<tr>
<td>2</td>
<td>Access to the pairs of interfered shapes ((nE_i, nF_j))(k, k=0, 1... nk), where (nE_i) is DS index of edge (E_i), (nF_j) is DS index of face (F_j) and (nk) is the number of pairs.</td>
<td><code>BOPDS_Iterator</code></td>
</tr>
<tr>
<td>3</td>
<td>Initialize pave blocks for the edges involved in the interference, if it is necessary.</td>
<td><code>BOPDS_DS::ChangePaveBlocks()</code></td>
</tr>
<tr>
<td>4</td>
<td>Access to the pave blocks of interfered edge ((PB_{i1}, PB_{i2}... PB_{iN})) for edge (E_i)</td>
<td><code>BOPAlgo_PaveFiller::PerformEF()</code></td>
</tr>
<tr>
<td>5</td>
<td>Compute shrunk data for pave blocks (in terms of Pave, PaveBlock and CommonBlock) if it is necessary.</td>
<td><code>BOPAlgo_PaveFiller::FillShrunkData()</code></td>
</tr>
<tr>
<td>6</td>
<td>Compute Edge/Face interference for pave block (PB_{ix}), and face (nF_j). The result of the computation is a set of objects of type</td>
<td><code>IntTools_EdgeFace</code></td>
</tr>
</tbody>
</table>
For each CommonPart of type VERTEX: Create new vertices $V_{Ni}$ ($i=1, 2..., NbVN$), where $NbVN$ is the number of new vertices. Merge vertices $V_{Ni}$ as follows: a) create new object $PFn$ of type $BOPAlgo\_PaveFiller$ with its own DS; b) use new vertices $V_{Ni}$ ($i=1, 2..., NbVN$), $NbVN$ as arguments (in terms of $TopoDs\_Shape$) of $PFn$; c) invoke method $Perform()$ for $PFn$. The resulting vertices $V_{NXi}$ ($i=1, 2..., NbVNX$), where $NbVNX$ is the number of vertices, are obtained via mapping between $V_{Ni}$ and the results of $PVn$.

For each CommonPart of type EDGE: Create common blocks ($CBc. C=0, 1...nCs$) from pave blocks that lie on the faces. Attach the common blocks to the pave blocks.

Post-processing. Append the paves of $V_{NXi}$ into the
<table>
<thead>
<tr>
<th>8</th>
<th>corresponding pave blocks in terms of \textbf{Pave, PaveBlock and CommonBlock}.</th>
<th>\textit{BOPDS_PaveBlock:: AppendExtPave(()}</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Split pave blocks and common blocks \textit{CBc} by the paves.</td>
<td>\texttt{BOPAlgo_PaveFiller::PerformVertices1()}, \texttt{BOPDS_DS:: UpdatePaveBlock()} and \texttt{BOPDS_DS:: UpdateCommonBlock()}</td>
</tr>
<tr>
<td>10</td>
<td>Append Edge/\textit{Face} interferences in the DS</td>
<td>\texttt{BOPDS_DS::AddInterf()}</td>
</tr>
<tr>
<td>11</td>
<td>Update \textit{FaceInfo} for all faces having \textit{EF} common parts.</td>
<td>\texttt{BOPDS_DS:: UpdateFaceInfoIn()}</td>
</tr>
</tbody>
</table>
# Build Split Edges

The input data for this step is the DS after computing Edge/Face Interferences.

For each pave block $PB$ take the following steps:

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Get the real pave block $PBR$, which is equal to $PB$ if $PB$ is not a common block and to $PB_1$ if $PB$ is a common block. $PB_1$ is the first pave block in the pave blocks list of the common block. See Pave, PaveBlock and CommonBlock.</td>
<td>$BOPAlgo_PaveFiller::MakeSplitEdges()$</td>
</tr>
<tr>
<td>2</td>
<td>Build the split edge $Esp$ using the information from $DS$ and $PBR$.</td>
<td>$BOPTools_Tools::MakeSplitEdge()$</td>
</tr>
<tr>
<td>3</td>
<td>Compute $BOPDS_ShapeInfo$ contents for $Esp$</td>
<td>$BOPAlgo_PaveFiller::MakeSplitEdges()$</td>
</tr>
<tr>
<td>4</td>
<td>Append $BOPDS_ShapeInfo$ contents to the DS</td>
<td>$BOPDS_DS::Append()$</td>
</tr>
</tbody>
</table>
Compute Face/Face Interferences

The input data for this step is DS after building Split Edges.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize array of Face/Face interferences</td>
<td>BOPAlgo_PaveFiller::PerformFF()</td>
</tr>
<tr>
<td>2</td>
<td>Access to the pairs of interfered shapes ((nFi, nFj)k, k=0, 1...nk), where (nFi) is DS index of edge Fi, (nFj) is DS index of face Fj and (nk) is the number of pairs.</td>
<td>BOPDS_Iterator</td>
</tr>
<tr>
<td>3</td>
<td>Compute Face/Face interference</td>
<td>IntTools_FaceFace</td>
</tr>
<tr>
<td>4</td>
<td>Append Face/Face interferences in the DS.</td>
<td>BOPDS_DS::AddInterf()</td>
</tr>
</tbody>
</table>
# Build Section Edges

The input data for this step is the DS after computing Face/Face interferences.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For each Face/Face interference (nFi, nFj), retrieve <strong>FacelInfo</strong>. Create draft vertices from intersection points (VPk) ((k=1, 2\ldots, NbVP)), where (NbVP) is the number of new vertices, and the draft vertex (VPk) is created from an intersection point if (VPk \neq Vm) ((m = 0, 1, 2\ldots NbVm)), where (Vm) is an existing vertex for the faces (nFi) and (nFj) (On or In in terms of TopoDs_Shape), (NbVm) is the number of vertices existing on faces (nFi) and (nFj) and (\neq) – means non-coincidence in terms of Vertex/Vertex interference.</td>
<td><strong>BOPAlgo_PaveFiller::MakeBlocks()</strong></td>
</tr>
<tr>
<td>2</td>
<td>For each intersection curve (Cijk)</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Create paves PVc for the curve using existing vertices, i.e. vertices On or In (in terms of FaceInfo) for faces $nFi$ and $nFj$. Append the paves PVc</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Create technological vertices $Vt$, which are the bounding points of an intersection curve (with the value of tolerance $Tol(Cijk)$). Each vertex $Vt$ with parameter $Tt$ on curve $Cijk$ forms pave $PVt$ on curve $Cijk$. Append technological paves.</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Create pave blocks $PBk$ for the curve using paves ($k=1, 2\ldots, NbPB$), where $NbPB$ is the number of pave blocks</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$BOPAlgo_PaveFiller_::_PutPaveOnCurve()$ and $BOPDS_PaveBlock_::_AppendExtPave()$</td>
</tr>
<tr>
<td></td>
<td>$BOPAlgo_PaveFiller_::_PutBoundPaveOnCurve()$</td>
</tr>
<tr>
<td></td>
<td>$BOPAlgo_PaveFiller_::_MakeBlocks()$</td>
</tr>
</tbody>
</table>

**Build draft section edges $ESk$ using the pave blocks ($k=1, 2\ldots, NbES$), where $NbES$ is the number of draft section edges.** The draft section edge is created from a pave block $PBk$ if $PBk$ has state In or On
<table>
<thead>
<tr>
<th>2.4</th>
<th>for both faces $nFi$ and $nF,j$ and $PBk \neq PBm$ ($m=0, 1, 2... NbPBm$), where $PBm$ is an existing pave block for faces $nFi$ and $nF,j$ (On or $\text{in}$ in terms of $\text{FaceInfo}$), $NbVm$ is the number of existing pave blocks for faces $nFi$ and $nF,j$ and $\neq$ – means non-coincidence (in terms of $\text{Vertex/Face interference}$).</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Intersect the draft vertices $VPk$ ($k=1, 2..., NbVP$) and the draft section edges $ESk$ ($k=1, 2..., NbES$). For this: a) create new object $PFn$ of type $\text{BOPAlgo_PaveFiller}$ with its own DS; b) use vertices $VPk$ and edges $ESk$ as arguments (in terms of $\text{Arguments}$) of $PFn$; c) invoke method $\text{Perform()}$ for $PFn$. Resulting vertices $VPXk$ ($k=1, 2... NbVPX$) and edges $ESXk$ ($k=1, 2... NbESX$) are obtained via mapping between</td>
</tr>
</tbody>
</table>

| BOPTools_Tools::MakeEdge() | |

<p>| BOPAlgo_PaveFiller::PostTreatFF() | |</p>
<table>
<thead>
<tr>
<th>4</th>
<th>$VP_k, ES_k$ and the results of $PV_n$.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Update face info (sections about pave blocks and vertices)</td>
</tr>
<tr>
<td></td>
<td>$BOPAlgo_PaveFiller::PerformFF()$</td>
</tr>
</tbody>
</table>
Build P-Curves

The input data for this step is the DS after building section edges.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For each Face/Face interference $nFi$ and $nFj$ build p-Curves on $nFi$ and $nFj$ for each section edge $ESXk$.</td>
<td><code>BOPAlgo_PaveFiller::MakePCurves()</code></td>
</tr>
<tr>
<td>2</td>
<td>For each pave block that is common for faces $nFi$ and $nFj$ build p-Curves on $nFi$ and $nFj$.</td>
<td><code>BOPAlgo_PaveFiller::MakePCurves()</code></td>
</tr>
</tbody>
</table>
## Process Degenerated Edges

The input data for this step is the DS after building P-curves.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For each degenerated edge $ED$ having vertex $VD$</td>
<td>$\text{BOPAlgo_PaveFiller::ProcessDE()}$</td>
</tr>
<tr>
<td>1</td>
<td>Find pave blocks $PB_i$ ($i=1,2 \ldots NbPB$), where $NbPB$ is the number of pave blocks, that go through vertex $VD$.</td>
<td>$\text{BOPAlgo_PaveFiller::FindPaveBlocks()}$</td>
</tr>
<tr>
<td>2</td>
<td>Compute paves for the degenerated edge $ED$ using a 2D curve of $ED$ and a 2D curve of $PB_i$. Form pave blocks $PBD_i$ ($i=1,2 \ldots NbPBD$), where $NbPBD$ is the number of the pave blocks for the degenerated edge $ED$</td>
<td>$\text{BOPAlgo_PaveFiller::FillPaves()}$</td>
</tr>
<tr>
<td>3</td>
<td>Build split edges $ESD_i$ ($i=1,2 \ldots NbESD$), where $ESD$ is the number of split edges, using the pave blocks $PBD_i$</td>
<td>$\text{BOPAlgo_PaveFiller:: MakeSplitEdge()}$</td>
</tr>
</tbody>
</table>
General description of the Building Part

Building Part (BP) is used to

- Build the result of the operation
- Provide history information (in terms of ::Generated(), ::Modified() and ::IsDeleted()) BP uses the DS prepared by BOPAlgo_PaveFiller described at chapter 5 as input data. BP is implemented in the following classes:
  - BOPAlgo_Builder – for the General Fuse operator (GFA).
  - BOPAlgo_BOP – for the Boolean Operation operator (BOA).
  - BOPAlgo_Section – for the Section operator (SA).
  - BOPAlgo_MakerVolume – for the Volume Maker operator.
  - BOPAlgo_Splitter – for the Splitter operator.
  - BOPAlgo_CellsBuilder – for the Cells Builder operator.

![Diagram for BP classes]

The class BOPAlgo_BuilderShape provides the interface for algorithms
that have:

- A Shape as the result;
- History information (in terms of ::Generated(), ::Modified() and ::IsDeleted()).
General Fuse Algorithm

Arguments

The arguments of the algorithm are shapes (in terms of TopoDS_Shape). The main requirements for the arguments are described in Data Structure chapter.
Results

During the operation argument \( Si \) can be split into several parts \( Si_1, Si_2\ldots Si_{1NbSp}, \) where \( NbSp \) is the number of parts. The set \( (Si_1, Si_2\ldots Si_{1NbSp}) \) is an image of argument \( Si \).

- The result of the General Fuse operation is a compound. Each sub-shape of the compound corresponds to the certain argument shape \( S_1, S_2\ldots S_n \) and has shared sub-shapes in accordance with interferences between the arguments.
- For the arguments of the type EDGE, FACE, SOLID the result contains split parts of the argument.
- For the arguments of the type WIRE, SHELL, COMPSOLID, COMPOUND the result contains the image of the shape of the corresponding type (i.e. WIRE, SHELL, COMPSOLID or COMPOUND). The types of resulting shapes depend on the type of the corresponding argument participating in the operation. See the table below:

<table>
<thead>
<tr>
<th>No</th>
<th>Type of argument</th>
<th>Type of resulting shape</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>COMPOUND</td>
<td>COMPOUND</td>
<td>The resulting COMPOUND is built from images of sub-shapes of type COMPOUND COMPSOLID, SHELL, WIRE and VERTEX. Sets of split sub-shapes of type SOLID, FACE, EDGE.</td>
</tr>
<tr>
<td>2</td>
<td>COMPSOLID</td>
<td>COMPSOLID</td>
<td>The resulting COMPSOLID is built from split SOLIDs.</td>
</tr>
<tr>
<td>3</td>
<td>SOLID</td>
<td>Set of split SOLIDs</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SHELL</td>
<td>SHELL</td>
<td>The resulting SHELL is built from split FACEs</td>
</tr>
<tr>
<td>5</td>
<td>FACE</td>
<td>Set of split FACEs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WIRE</td>
<td>WIRE</td>
<td>The resulting WIRE is built from split EDGEs</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>EDGE</td>
<td></td>
<td>Set of split EDGEs</td>
</tr>
<tr>
<td>8</td>
<td>VERTEX</td>
<td>VERTEX</td>
<td></td>
</tr>
</tbody>
</table>
Examples

Please, have a look at the examples, which can help to better understand the definitions.

Case 1: Three edges intersecting at a point

Let us consider three edges: $E_1$, $E_2$ and $E_3$ that intersect in one 3D point.

Three Intersecting Edges

The result of the GFA operation is a compound containing 6 new edges: $E_{11}$, $E_{12}$, $E_{21}$, $E_{22}$, $E_{31}$, and $E_{32}$. These edges have one shared vertex $V_{n1}$.

In this case:

- The argument edge $E_1$ has resulting split edges $E_{11}$ and $E_{12}$ (image of $E_1$).
- The argument edge $E_2$ has resulting split edges $E_{21}$ and $E_{22}$ (image of $E_2$).
- The argument edge $E_3$ has resulting split edges $E_{31}$ and $E_{32}$ (image of $E_3$).

Case 2: Two wires and an edge

Let us consider two wires $W_1$ ($E_{w11}$, $E_{w12}$, $E_{w13}$) and $W_2$ ($E_{w21}$, $E_{w22}$, $E_{w23}$) and edge $E_1$. 
Two wires and an edge

The result of the GF operation is a compound consisting of 2 wires: \( Wn1 \) (\( Ew11, En1, En2, En3, Ew13 \)) and \( Wn2 \) (\( Ew21, En2, En3, En4, Ew23 \)) and two edges: \( E11 \) and \( E12 \).

In this case:

- The argument \( W1 \) has image \( Wn1 \).
- The argument \( W2 \) has image \( Wn2 \).
- The argument edge \( E1 \) has split edges \( E11 \) and \( E12 \). (image of \( E1 \)).
  The edges \( En1, En2, En3, En4 \) and vertex \( Vn1 \) are new shapes created during the operation. Edge \( Ew12 \) has split edges \( En1, En2 \) and \( En3 \) and edge \( Ew22 \) has split edges \( En2, En3 \) and \( En4 \).

Case 3: An edge intersecting with a face

Let us consider edge \( E1 \) and face \( F2 \):

An edge intersecting with a face
The result of the GF operation is a compound consisting of 3 shapes:

- Split edge parts $E_{11}$ and $E_{12}$ (image of $E_1$).
- New face $F_{21}$ with internal edge $E_{12}$ (image of $F_2$).

**Case 4: An edge lying on a face**

Let us consider edge $E_1$ and face $F_2$:

![Diagram of an edge lying on a face]

The result of the GF operation is a compound consisting of 5 shapes:

- Split edge parts $E_{11}$, $E_{12}$ and $E_{13}$ (image of $E_1$).
- Split face parts $F_{21}$ and $F_{22}$ (image of $F_2$).

**Case 5: An edge and a shell**

Let us consider edge $E_1$ and shell $Sh_2$ that consists of 2 faces: $F_{21}$ and $F_{22}$

![Diagram of an edge and a shell]
The result of the GF operation is a compound consisting of 5 shapes:

- Split edge parts $E_{11}$, $E_{12}$, $E_{13}$ and $E_{14}$ (image of $E_1$).
- Image shell $Sh_{21}$ (that contains split face parts $F_{211}$, $F_{212}$, $F_{221}$ and $F_{222}$).

**Case 6: A wire and a shell**

Let us consider wire $W_1$ ($E_1$, $E_2$, $E_3$, $E_4$) and shell $Sh_2$ ($F_{21}$, $F_{22}$).

A wire and a shell

The result of the GF operation is a compound consisting of 2 shapes:

- Image wire $W_{11}$ that consists of split edge parts from wire $W_1$: $E_{11}$, $E_{12}$, $E_{13}$ and $E_{14}$.
- Image shell $Sh_{21}$ that contains split face parts: $F_{211}$, $F_{212}$, $F_{213}$, $F_{221}$, $F_{222}$ and $F_{223}$.

**Case 7: Three faces**

Let us consider 3 faces: $F_1$, $F_2$ and $F_3$. 
Three faces

The result of the GF operation is a compound consisting of 7 shapes:

- Split face parts: \( Fn1, Fn2, Fn3, Fn4, Fn5, Fn6 \) and \( Fn7 \).

Case 8: A face and a shell

Let us consider shell \( Sh1 (F11, F12, F13) \) and face \( F2 \).

A face and a shell

The result of the GF operation is a compound consisting of 4 shapes:

- Image shell \( Sh11 \) that consists of split face parts from shell \( Sh1: \ Fn1, Fn2, Fn3, Fn4, Fn5 \) and \( Fn6 \).
- Split parts of face \( F2: Fn3, Fn6 \) and \( Fn7 \).

Case 9: A shell and a solid

Let us consider shell \( Sh1 (F11, F12...F16) \) and solid \( So2 \).
The result of the GF operation is a compound consisting of 2 shapes:

- Image shell $Sh11$ consisting of split face parts of $Sh1$: $Fn1$, $Fn2$ ... $Fn8$.
- Solid $So21$ with internal shell. (image of $So2$).

Case 10: A compound and a solid

Let us consider compound $Cm1$ consisting of 2 solids $So11$ and $So12$) and solid $So2$. 
A compound and a solid: arguments

The result of the GF operation is a compound consisting of 4 shapes:

- Image compound \( Cm11 \) consisting of split solid parts from \( So11 \) and \( So12 \) (\( Sn1, Sn2, Sn3, Sn4 \)).
- Split parts of solid \( So2 \) (\( Sn2, Sn3, Sn5 \)).
Class BOPAlgo_Builder

GFA is implemented in the class BOPAlgo_Builder.

Fields

The main fields of the class are described in the Table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>myPaveFiller</td>
<td>Pointer to the BOPAlgo_PaveFiller object</td>
</tr>
<tr>
<td>myDS</td>
<td>Pointer to the BOPDS_DS object</td>
</tr>
<tr>
<td>myContext</td>
<td>Pointer to the intersection Context</td>
</tr>
<tr>
<td>myImages</td>
<td>The Map between the source shape and its images</td>
</tr>
<tr>
<td>myShapesSD</td>
<td>The Map between the source shape (or split part of source shape) and the shape (or part of shape) that will be used in result due to same domain property.</td>
</tr>
</tbody>
</table>

Initialization

The input data for this step is a BOPAlgo_PaveFiller object (in terms of Intersection) at the state after Processing of degenerated edges with the corresponding DS.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Check the readiness of the DS and BOPAlgo_PaveFiller.</td>
<td>BOPAlgo_Builder::CheckData()</td>
</tr>
<tr>
<td>2</td>
<td>Build an empty result of type Compound.</td>
<td>BOPAlgo_Builder::Prepare()</td>
</tr>
</tbody>
</table>

Build Images for Vertices

The input data for this step is BOPAlgo_Builder object after Initialization.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Build Result of Type Vertex**

The input data for this step is *BOPAlgo_Builder* object after building images for vertices and *Type*, which is the shape type *(TopAbs_VERTEX)*.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For the arguments of type <em>Type</em>. If there is an image for the argument: add the image to the result. If there is no image for the argument: add the argument to the result.</td>
<td><em>BOPAlgo_Builder::BuildResult()</em></td>
</tr>
</tbody>
</table>

**Build Images for Edges**

The input data for this step is *BOPAlgo_Builder* object after building result of type vertex.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For all pave blocks in the DS. Fill <em>myImages</em> for the original edge <em>E</em> by split edges <em>ESPi</em> from pave blocks. In case of common blocks on edges, use edge <em>ESPSDj</em> that corresponds to the leading pave block and fill <em>myShapesSD</em> by the pairs <em>ESPi/ESPSDj</em>.</td>
<td><em>BOPAlgo_Builder::FillImagesEdges()</em></td>
</tr>
</tbody>
</table>

**Build Result of Type Edge**

This step is the same as **Building Result of Type Vertex**, but for the
Build Images for Wires

The input data for this step is:

- `BOPAlgo_Builder` object after building result of type `Edge`;
- Original Shape – Wire
- `Type` – the shape type (`TopAbs_WIRE`).

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For all arguments of the type <code>Type</code>. Create a container <code>C</code> of the type <code>Type</code>.</td>
<td><code>BOPAlgo_Builder::FillImagesContainers()</code></td>
</tr>
<tr>
<td>2</td>
<td>Add to <code>C</code> the images or non-split parts of the <code>Original Shape</code>, taking into account its orientation.</td>
<td><code>BOPAlgo_Builder::FillImagesContainers()</code>&lt;br&gt;<code>BOPTools_Tools::IsSplitToReverse()</code></td>
</tr>
<tr>
<td>3</td>
<td>Fill <code>myImages</code> for the <code>Original Shape</code> by the information above.</td>
<td><code>BOPAlgo_Builder::FillImagesContainers()</code></td>
</tr>
</tbody>
</table>

Build Result of Type Wire

This step is the same as `Building Result of Type Vertex` but for the type `Wire`.

Build Images for Faces

The input data for this step is `BOPAlgo_Builder` object after building result of type `Wire`.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Build Split Faces for all interfered DS</td>
<td></td>
</tr>
<tr>
<td>Step</td>
<td>Description</td>
<td>Function</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>1.1</td>
<td>Collect all edges or their images of $Fi(ESPij)$.</td>
<td><code>BOPAlgo_Builder::BuildSplitFaces()</code></td>
</tr>
<tr>
<td>1.2</td>
<td>Impart to ESPij the orientation to be coherent with the original one.</td>
<td><code>BOPAlgo_Builder::BuildSplitFaces()</code></td>
</tr>
<tr>
<td>1.3</td>
<td>Collect all section edges $SEk$ for $Fi$.</td>
<td><code>BOPAlgo_Builder::BuildSplitFaces()</code></td>
</tr>
<tr>
<td>1.4</td>
<td>Build split faces for $Fi$ ($Fi1, Fi2...FiNbSp$), where $NbSp$ is the number of split parts (see Building faces from a set of edges for more details).</td>
<td><code>BOPAlgo_BuilderFace</code></td>
</tr>
<tr>
<td>1.5</td>
<td>Impart to ($Fi1, Fi2...FiNbSp$) the orientation coherent with the original face $Fi$.</td>
<td><code>BOPAlgo_Builder::BuildSplitFaces()</code></td>
</tr>
<tr>
<td>1.6</td>
<td>Fill the map $mySplits$ with $Fi(Fi1, Fi2...FiNbSp)$</td>
<td><code>BOPAlgo_Builder::BuildSplitFaces()</code></td>
</tr>
<tr>
<td>2</td>
<td>Fill Same Domain faces</td>
<td><code>BOPAlgo_Builder::FillSameDomainFaces</code></td>
</tr>
<tr>
<td>2.1</td>
<td>Find and collect in the contents of $mySplits$ the pairs of same domain split faces ($Fij, Fkl)m$, where $m$ is the number of pairs.</td>
<td><code>BOPAlgo_Builder::FillSameDomainFaces</code> <code>BOPTools_Tools::AreFacesSameDomain()</code></td>
</tr>
<tr>
<td></td>
<td>Compute the connexity chains 1) of same domain</td>
<td></td>
</tr>
</tbody>
</table>

shapes $Fi$ of type `FACE`. 

1. Collect all edges or their images of $Fi(ESPij)$.
2. Impart to ESPij the orientation to be coherent with the original one.
3. Collect all section edges $SEk$ for $Fi$.
4. Build split faces for $Fi$ ($Fi1, Fi2...FiNbSp$), where $NbSp$ is the number of split parts (see Building faces from a set of edges for more details).
5. Impart to ($Fi1, Fi2...FiNbSp$) the orientation coherent with the original face $Fi$.
6. Fill the map $mySplits$ with $Fi(Fi1, Fi2...FiNbSp)$

2. Fill Same Domain faces

Find and collect in the contents of $mySplits$ the pairs of same domain split faces ($Fij, Fkl)m$, where $m$ is the number of pairs.

Compute the connexity chains 1) of same domain
### Chains of same domain faces

- The pairs of same domain faces are: \((F11, F21), (F22, F31), (F41, F51), (F41, F6)\) and \((F51, F6)\).
- The pairs produce the three chains: \((F11, F21), (F22, F31)\) and \((F41, F51, F6)\).

### Build Result of Type Face

This step is the same as **Building Result of Type Vertex** but for the type **Face**.

### Build Images for Shells
The input data for this step is:

- *BOPAlgo_Builder* object after building result of type face;
- *Original Shape* – a Shell;
- *Type* – the type of the shape (*TopAbs_SHELL*).

The procedure is the same as for building images for wires.

### Build Result of Type Shell

This step is the same as *Building Result of Type Vertex* but for the type *Shell*.

### Build Images for Solids

The input data for this step is *BOPAlgo_Builder* object after building result of type *Shell*.

The following procedure is executed for all interfered DS shapes *Si* of type *SOLID*.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Collect all images or non-split parts for all faces <em>(FSPij)</em> that have 3D state <em>In Si.</em></td>
<td><em>BOPAlgo_Builder::FillIn3DParts ()</em></td>
</tr>
<tr>
<td>2</td>
<td>Collect all images or non-split parts for all faces of <em>Si</em></td>
<td><em>BOPAlgo_Builder::BuildSplitSolids()</em></td>
</tr>
<tr>
<td>3</td>
<td>Build split solids for <em>Si</em> -&gt; <em>(Si1, Si2…SiNbSp)</em>, where <em>NbSp</em> is the number of split parts (see <em>Building faces from a set of edges</em> for more details)</td>
<td><em>BOPAlgo_BuilderSolid</em></td>
</tr>
<tr>
<td>4</td>
<td>Fill the map Same Domain solids <em>myShapesSD</em></td>
<td><em>BOPAlgo_Builder::BuildSplitSolids()</em></td>
</tr>
</tbody>
</table>
### Build Result of Type Solid

This step is the same as **Building Result of Type Vertex**, but for the type Solid.

### Build Images for Type CompSolid

The input data for this step is:

- `BOPAlgo_Builder` object after building result of type solid;
- *Original Shape* – a Compsolid;
- *Type* – the type of the shape (`TopAbs_COMPSOLID`).

The procedure is the same as for building images for wires.

### Build Result of Type Compsolid

This step is the same as **Building Result of Type Vertex**, but for the type Compsolid.

### Build Images for Compounds

The input data for this step is as follows:

- `BOPAlgo_Builder` object after building results of type `compsolid`;
- *Original Shape* – a Compound;
- *Type* – the type of the shape (`TopAbs_COMPOUND`).

The procedure is the same as for building images for wires.

### Build Result of Type Compound

This step is the same as **Building Result of Type Vertex**, but for the type Compound.
Post-Processing

The purpose of the step is to correct tolerances of the result to provide its validity in terms of *BRepCheck_Analyzer*.

The input data for this step is a *BOPAlgo_Builder* object after building result of type compound.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Correct tolerances of vertices on curves</td>
<td><em>BOPTools_Tools::CorrectPointOnCurve()</em></td>
</tr>
<tr>
<td>2</td>
<td>Correct tolerances of edges on faces</td>
<td><em>BOPTools_Tools::CorrectCurveOnSurface()</em></td>
</tr>
</tbody>
</table>
Splitter Algorithm

The Splitter algorithm allows splitting a group of arbitrary shapes by another group of arbitrary shapes. It is based on the General Fuse algorithm, thus all options of the General Fuse such as Fuzzy mode, safe processing mode, parallel mode, gluing mode and history support are also available in this algorithm.
Arguments

- The arguments of the Splitter algorithm are divided into two groups - *Objects* (shapes that will be split) and *Tools* (shapes, by which the *Objects* will be split);
- The requirements for the arguments (both for *Objects* and *Tools*) are the same as for the General Fuse algorithm - there can be any number of arguments of any type in each group, but each argument should be valid and not self-interfered.
Results

- The result of Splitter algorithm contains only the split parts of the shapes included into the group of *Objects*;
- The split parts of the shapes included only into the group of *Tools* are excluded from the result;
- If there are no shapes in the group of *Tools* the result of the operation will be equivalent to the result of General Fuse operation;
- The shapes can be split by other shapes from the same group (if these shapes are interfering).
Usage

API

On the low level the Splitter algorithm is implemented in class BOPAlgo_Splitter. The usage of this algorithm looks as follows:

```cpp
BOPAlgo_Splitter aSplitter;
BOPCol_ListOfShape aLSObjects = ...; // Objects
BOPCol_ListOfShape aLSTools = ...; // Tools
Standard_Boolean bRunParallel = Standard_False; /* parallel or single mode (the default value is FALSE)*/
Standard_Real aTol = 0.0; /* fuzzy option (default value is 0)*/
Standard_Boolean bSafeMode = Standard_False; /* protect or not the arguments from modification*/
BOPAlgo_Glue aGlue = BOPAlgo_GlueOff; /* Glue option to speed up intersection of the arguments*/
// setting arguments
aSplitter.SetArguments(aLSObjects);
aSplitter.SetTools(aLSTools);
// setting options
aSplitter.SetRunParallel(bRunParallel);
aSplitter.SetFuzzyValue(aTol);
aSplitter.SetNonDestructive(bSafeMode);
aSplitter.SetGlue(aGlue);
//
aSplitter.Perform(); //perform the operation
if (aSplitter.HasErrors()) {
    //check error status
    return;
}
//
const TopoDS_Shape& aResult = aSplitter.Shape(); // result of the operation
```
The command `bsplit` implements the Splitter algorithm in DRAW. Similarly to the `bbuild` command for the General Fuse algorithm, the `bsplit` command should be used after the Pave Filler is filled.

```
# s1 s2 s3 - objects
# t1 t2 t3 - tools
bclearobjects
bcleartools
baddobjects s1 s2 s3
baddtools t1 t2 t3
bfillds
bsplit result
```
Examples

Example 1

Splitting a face by the set of edges:

```plaintext
# draw script for reproducing
bclearobjects
bcleartools

set height 20
cylinder cyl 0 0 0 0 0 1 10
mkface f cyl 0 2*pi -$height $height
baddobjects f

# create tool edges
compound edges

set nb_uedges 10
set pi2 [dval 2*pi]
set ustep [expr $pi2/$nb_uedges]
for {set i 0} {$i <= $pi2} {set i [expr $i + $ustep]} {
    uiso c cyl $i
    mkedge e c -25 25
    add e edges
}

set nb_vedges 10
set vstep [expr 2*$height/$nb_vedges]
for {set i -20} {$i <= 20} {set i [expr $i + $vstep]} {
    viso c cyl $i
    mkedge e c
    add e edges
}
```
Example 2

Splitting a plate by the set of cylinders:

```bash
# draw script for reproducing:
bclearobjects
bcleartools

box plate 100 100 1
baddobjects plate

pcylinder p 1 11
compound cylinders
for {set i 0} {$i < 101} {incr i 5} {
    for {set j 0} {$j < 101} {incr j 5} {
```
Example 3

Splitting shell hull by the planes:
Arguments

Results
Boolean Operations Algorithm

Arguments

- The arguments of BOA are shapes in terms of TopoDS_Shape. The main requirements for the arguments are described in the Data Structure.
- There are two groups of arguments in BOA:
  - Objects ($S1=S11, S12, ...$);
  - Tools ($S2=S21, S22, ...$).
- The following table contains the values of dimension for different types of arguments:

<table>
<thead>
<tr>
<th>No</th>
<th>Type of Argument</th>
<th>Index of Type</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>COMPOUND</td>
<td>0</td>
<td>One of 0, 1, 2, 3</td>
</tr>
<tr>
<td>2</td>
<td>COMPSOLID</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>SOLID</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>SHELL</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>FACE</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>WIRE</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>EDGE</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>VERTEX</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

- For Boolean operation Fuse all arguments should have equal dimensions.
- For Boolean operation Cut the minimal dimension of $S2$ should not be less than the maximal dimension of $S1$.
- For Boolean operation Common the arguments can have any dimension.
Results. General Rules

- The result of the Boolean operation is a compound (if defined). Each sub-shape of the compound has shared sub-shapes in accordance with interferences between the arguments.
- The content of the result depends on the type of the operation (Common, Fuse, Cut12, Cut21) and the dimensions of the arguments.
- The result of the operation Fuse is defined for arguments $S_1$ and $S_2$ that have the same dimension value: $\text{Dim}(S_1)=\text{Dim}(S_2)$. If the arguments have different dimension values the result of the operation Fuse is not defined. The dimension of the result is equal to the dimension of the arguments. For example, it is impossible to fuse an edge and a face.
- The result of the operation Fuse for arguments $S_1$ and $S_2$ contains the parts of arguments that have states OUT relative to the opposite arguments.
- The result of the operation Fuse for arguments $S_1$ and $S_2$ having dimension value 3 (Solids) is refined by removing all possible internal faces to provide minimal number of solids.
- The result of the operation Common for arguments $S_1$ and $S_2$ is defined for all values of the dimensions of the arguments. The result can contain shapes of different dimensions, but the minimal dimension of the result will be equal to the minimal dimension of the arguments. For example, the result of the operation Common between edges cannot be a vertex.
- The result of the operation Common for the arguments $S_1$ and $S_2$ contains the parts of the argument that have states IN and ON relative to the opposite argument.
- The result of the operation Cut is defined for arguments $S_1$ and $S_2$ that have values of dimensions $\text{Dim}(S_2)$ that should not be less than $\text{Dim}(S_1)$. The result can contain shapes of different dimensions, but the minimal dimension of the result will be equal to the minimal dimension of the objects $\text{Dim}(S_1)$. The result of the operation Cut12 is not defined for other cases. For example, it is impossible to cut an edge from a solid, because a solid without an edge is not defined.
- The result of the operation Cut12 for arguments $S_1$ and $S_2$ contains the parts of argument $S_1$ that have state OUT relative to the opposite argument $S_2$. 

• The result of the operation Cut21 for arguments S1 and S2 contains the parts of argument S2 that have state OUT relative to the opposite argument S1.
• For the arguments of collection type (WIRE, SHELL, COMPSOLID) the type will be passed in the result. For example, the result of Common operation between Shell and Wire will be a compound containing Wire.
• For the arguments of collection type (WIRE, SHELL, COMPSOLID) containing overlapping parts the overlapping parts passed into result will be repeated for each container from the input shapes containing such parts. The containers completely included in other containers will be avoided in the result.
• For the arguments of collection type (WIRE, SHELL, COMPSOLID) the containers included into result will have the same orientation as the original containers from arguments. In case of duplication its orientation will be defined by the orientation of the first container in arguments. Each container included into result will have coherent orientation of its sub-shapes.
• The result of the operation Fuse for the arguments of collection type (WIRE, SHELL) will consist of the shapes of the same collection type. The overlapping parts (EDGES/FACES) will be shared among containers, but duplicating containers will be avoided in the result. For example, the result of Fuse operation between two fully coinciding wires will be one wire, but the result of Fuse operation between two partially coinciding wires will be two wires sharing coinciding edges.
• The result of the operation Fuse for the arguments of type COMPSOLID will consist of the compound containing COMPSOLIDs created from connexity blocks of fused solids.
• The result of the operation Common for the arguments of collection type (WIRE, SHELL, COMPSOLID) will consist of the unique containers containing the overlapping parts. For example, the result of Common operation between two fully overlapping wires will be one wire containing all splits of edges. The number of wires in the result of Common operation between two partially overlapping wires will be equal to the number of connexity blocks of overlapping edges.
Examples

Case 1: Two Vertices

Let us consider two interfering vertices $V_1$ and $V_2$:

- The result of *Fuse* operation is the compound that contains new vertex $V$.

- The result of *Common* operation is a compound containing new vertex $V$.
- The result of *Cut12* operation is an empty compound.
- The result of *Cut21* operation is an empty compound.

Case 2: A Vertex and an Edge

Let us consider vertex $V1$ and the edge $E2$, that intersect in a 3D point:
• The result of *Fuse* operation is result is not defined because the dimension of the vertex (0) is not equal to the dimension of the edge (1).
• The result of *Common* operation is a compound containing vertex $V_1$ as the argument $V_1$ has a common part with edge $E2$.

• The result of *Cut12* operation is an empty compound.
• The result of *Cut21* operation is not defined because the dimension of the vertex (0) is less than the dimension of the edge (1).

**Case 3: A Vertex and a Face**

Let us consider vertex $V1$ and face $F2$, that intersect in a 3D point:

• The result of *Fuse* operation is not defined because the dimension of the vertex (0) is not equal to the dimension of the face (2).
• The result of *Common* operation is a compound containing vertex $V_1$ as the argument $V_1$ has a common part with face $F2$. 
The result of Cut12 operation is an empty compound.

The result of Cut21 operation is not defined because the dimension of the vertex (0) is less than the dimension of the face (2).

Case 4: A Vertex and a Solid

Let us consider vertex V1 and solid S2, that intersect in a 3D point:

- The result of Fuse operation is not defined because the dimension of the vertex (0) is not equal to the dimension of the solid (3).
- The result of Common operation is a compound containing vertex V1 as the argument V1 has a common part with solid S2.

- The result of Cut12 operation is an empty compound.
- The result of Cut21 operation is not defined because the dimension of the vertex (0) is less than the dimension of the solid (3).

Case 5: Two edges intersecting at one point
Let us consider edges $E1$ and $E2$ that intersect in a 3D point:

- The result of $Fuse$ operation is a compound containing split parts of arguments i.e. 4 new edges $E11$, $E12$, $E21$, and $E22$. These edges have one shared vertex $Vn1$. In this case:
  - argument edge $E1$ has resulting split edges $E11$ and $E12$ (image of $E1$);
  - argument edge $E2$ has resulting split edges $E21$ and $E22$ (image of $E2$).

- The result of $Common$ operation is an empty compound because the dimension (0) of the common part between the edges (vertex) is less than the dimension of the arguments (1).
- The result of $Cut12$ operation is a compound containing split parts of the argument $E1$, i.e. 2 new edges $E11$ and $E12$. These edges have one shared vertex $Vn1$.

In this case the argument edge $E1$ has resulting split edges $E11$ and $E12$ (image of $E1$).
• The result of Cut21 operation is a compound containing split parts of the argument $E2$, i.e. 2 new edges $E21$ and $E12$. These edges have one shared vertex $Vn1$.

In this case the argument edge $E2$ has resulting split edges $E21$ and $E22$ (image of $E2$).

**Case 6: Two edges having a common block**

Let us consider edges $E1$ and $E2$ that have a common block:

• The result of Fuse operation is a compound containing split parts of arguments i.e. 3 new edges $E11$, $E12$ and $E22$. These edges have two shared vertices. In this case:
  ◦ argument edge $E1$ has resulting split edges $E11$ and $E12$ (image of $E1$);
  ◦ argument edge $E2$ has resulting split edges $E21$ and $E22$ (image of $E2$);
  ◦ edge $E12$ is common for the images of $E1$ and $E2$. 
The result of *Common* operation is a compound containing split parts of arguments i.e. 1 new edge \( E_{12} \). In this case edge \( E_{12} \) is common for the images of \( E_1 \) and \( E_2 \). The common part between the edges (edge) has the same dimension (1) as the dimension of the arguments (1).

The result of *Cut12* operation is a compound containing a split part of argument \( E_1 \), i.e. new edge \( E_{11} \).

The result of *Cut21* operation is a compound containing a split part of argument \( E_2 \), i.e. new edge \( E_{22} \).
Case 7: An Edge and a Face intersecting at a point

Let us consider edge $E_1$ and face $F_2$ that intersect at a 3D point:

- The result of Fuse operation is not defined because the dimension of the edge (1) is not equal to the dimension of the face (2).
- The result of Common operation is an empty compound because the dimension (0) of the common part between the edge and face (vertex) is less than the dimension of the arguments (1).
- The result of Cut12 operation is a compound containing split parts of the argument $E_1$, i.e. 2 new edges $E_{11}$ and $E_{12}$.

In this case the argument edge $E_1$ has no common parts with the face $F_2$ so the whole image of $E_1$ is in the result.

- The result of Cut21 operation is not defined because the dimension of the edge (1) is less than the dimension of the face (2).

Case 8: A Face and an Edge that have a common block

Let us consider edge $E_1$ and face $F_2$ that have a common block:
- The result of *Fuse* operation is not defined because the dimension of the edge (1) is not equal to the dimension of the face (2).
- The result of *Common* operation is a compound containing a split part of the argument *E1*, i.e. new edge *E12*.

In this case the argument edge *E1* has a common part with face *F2* so the corresponding part of the image of *E1* is in the result. The yellow square is not a part of the result. It only shows the place of *F2*.

- The result of *Cut12* operation is a compound containing split part of the argument *E1*, i.e. new edge *E11*.

In this case the argument edge *E1* has a common part with face *F2* so the corresponding part is not included into the result. The yellow square is not a part of the result. It only shows the place of F2.
The result of Cut21 operation is not defined because the dimension of the edge (1) is less than the dimension of the face (2).

Case 9: An Edge and a Solid intersecting at a point

Let us consider edge $E1$ and solid $S2$ that intersect at a point:

The result of Fuse operation is not defined because the dimension of the edge (1) is not equal to the dimension of the solid (3).

The result of Common operation is a compound containing a split part of the argument $E1$, i.e. new edge $E12$.

In this case the argument edge $E1$ has a common part with solid $S2$ so the corresponding part of the image of $E1$ is in the result. The yellow square is not a part of the result. It only shows the place of $S2$. 
The result of Cut12 operation is a compound containing split part of the argument $E1$, i.e. new edge $E11$.

In this case the argument edge $E1$ has a common part with solid $S2$ so the corresponding part is not included into the result. The yellow square is not a part of the result. It only shows the place of $S2$.

The result of Cut21 operation is not defined because the dimension of the edge (1) is less than the dimension of the solid (3).

Case 10: An Edge and a Solid that have a common block

Let us consider edge $E1$ and solid $S2$ that have a common block:
- The result of *Fuse* operation is not defined because the dimension of the edge (1) is not equal to the dimension of the solid (3).
- The result of *Common* operation is a compound containing a split part of the argument *E1*, i.e. new edge *E12*.

In this case the argument edge *E1* has a common part with solid *S2* so the corresponding part of the image of *E1* is in the result. The yellow square is not a part of the result. It only shows the place of *S2*.

- The result of *Cut12* operation is a compound containing split part of the argument *E1*, i.e. new edge *E11*.

In this case the argument edge *E1* has a common part with solid *S2* so the corresponding part is not included into the result. The yellow square is not a part of the result. It only shows the place of *S2*. 
The result of *Cut21* operation is not defined because the dimension of the edge (1) is less than the dimension of the solid (3).

**Case 11: Two intersecting faces**

Let us consider two intersecting faces $F_1$ and $F_2$:

- The result of *Fuse* operation is a compound containing split parts of arguments i.e. 2 new faces $F_{11}$ and $F_{21}$. These faces have one shared edge $E_{n1}$.
• The result of *Common* operation is an empty compound because the dimension (1) of the common part between *F1* and *F2* (edge) is less than the dimension of arguments (2).

• The result of *Cut12* operation is a compound containing split part of the argument *F1*, i.e. new face *F11*.

![Diagram of Cut12](image)

• The result of *Cut21* operation is a compound containing split parts of the argument *F2*, i.e. 1 new face *F21*.

![Diagram of Cut21](image)

**Case 12: Two faces that have a common part**

Let us consider two faces *F1* and *F2* that have a common part:

![Diagram of Case 12](image)

• The result of *Fuse* operation is a compound containing split parts of arguments, i.e. 3 new faces: *F11*, *F12* and *F22*. These faces are shared through edges In this case:
- the argument edge $F1$ has resulting split faces $F11$ and $F12$ (image of $F1$)
- the argument face $F2$ has resulting split faces $F12$ and $F22$ (image of $F2$)
- the face $F12$ is common for the images of $F1$ and $F2$.

The result of *Common* operation is a compound containing split parts of arguments i.e. 1 new face $F12$. In this case: face $F12$ is common for the images of $F1$ and $F2$. The common part between the faces (face) has the same dimension (2) as the dimension of the arguments (2).

The result of *Cut12* operation is a compound containing split part of the argument $F1$, i.e. new face $F11$.

The result of *Cut21* operation is a compound containing split parts of the argument $F2$, i.e. 1 new face $F21$. 
Case 13: Two faces that have a common edge

Let us consider two faces $F_1$ and $F_2$ that have a common edge:

- The result of $Fuse$ operation is a compound containing split parts of arguments, i.e. 2 new faces: $F_{11}$ and $F_{21}$. These faces have one shared edge $En_1$.

- The result of $Common$ operation is an empty compound because the dimension (1) of the common part between $F_1$ and $F_2$ (edge) is less than the dimension of the arguments (2).
- The result of $Cut12$ operation is a compound containing split part of the argument $F_1$, i.e. new face $F_{11}$. The vertices are shown just to clarify the fact that the edges are spitted.
The result of Cut21 operation is a compound containing split parts of the argument $F2$, i.e. 1 new face $F21$. The vertices are shown just to clarify the fact that the edges are spitted.

Case 14: Two faces that have a common vertex

Let us consider two faces $F1$ and $F2$ that have a common vertex:

The result of Fuse operation is a compound containing split parts of arguments, i.e. 2 new faces: $F11$ and $F21$. These faces have one shared vertex $Vn1$. 
The result of \textit{Common} operation is an empty compound because the dimension (0) of the common part between \textit{F1} and \textit{F2} (vertex) is less than the dimension of the arguments (2).

The result of \textit{Cut12} operation is a compound containing split part of the argument \textit{F1}, i.e. new face \textit{F11}.

The result of \textit{Cut21} operation is a compound containing split parts of the argument \textit{F2}, i.e. 1 new face \textit{F21}.

\textbf{Case 15: A Face and a Solid that have an intersection curve.}

Let us consider face \textit{F1} and solid \textit{S2} that have an intersection curve:

The result of \textit{Fuse} operation is not defined because the dimension of the face (2) is not equal to the dimension of the solid (3).
The result of \textit{Common} operation is a compound containing split part of the argument $F1$. In this case the argument face $F1$ has a common part with solid $S2$, so the corresponding part of the image of $F1$ is in the result. The yellow contour is not a part of the result. It only shows the place of $S2$.

The result of \textit{Cut12} operation is a compound containing split part of the argument $F1$. In this case argument face $F1$ has a common part with solid $S2$ so the corresponding part is not included into the result. The yellow contour is not a part of the result. It only shows the place of $S2$.

The result of \textit{Cut21} operation is is not defined because the dimension of the face (2) is less than the dimension of the solid (3).

\textbf{Case 16: A Face and a Solid that have overlapping faces.}

Let us consider face $F1$ and solid $S2$ that have overlapping faces:
• The result of *Fuse* operation is not defined because the dimension of the face (2) is not equal to the dimension of the solid (3).
• The result of *Common* operation is a compound containing split part of the argument *F1*. In this case the argument face *F1* has a common part with solid *S2*, so the corresponding part of the image of *F1* is included in the result. The yellow contour is not a part of the result. It only shows the place of *S2*.

• The result of *Cut12* operation is a compound containing split part of the argument *F1*. In this case argument face *F1* has a common part with solid *S2* so the corresponding part is not included into the result. The yellow contour is not a part of the result. It only shows the place of *S2*. 
The result of Cut21 operation is not defined because the dimension of the face (2) is less than the dimension of the solid (3).

Case 17: A Face and a Solid that have overlapping edges.

Let us consider face $F_1$ and solid $S_2$ that have overlapping edges:

- The result of Fuse operation is not defined because the dimension of the face (2) is not equal to the dimension of the solid (3).
- The result of Common operation is an empty compound because the dimension (1) of the common part between $F_1$ and $S_2$ (edge) is less than the lower dimension of the arguments (2).
- The result of Cut12 operation is a compound containing split part of the argument $F_1$. In this case argument face $F_1$ has a common part with solid $S_2$ so the corresponding part is not included into the result. The yellow contour is not a part of the result. It only shows the place of $S_2$. 
The result of Cut21 operation is not defined because the dimension of the face (2) is less than the dimension of the solid (3).

Case 18: A Face and a Solid that have overlapping vertices.

Let us consider face $F_1$ and solid $S_2$ that have overlapping vertices:

- The result of $Fuse$ operation is not defined because the dimension of the face (2) is not equal to the dimension of the solid (3).
- The result of $Common$ operation is an empty compound because the dimension (1) of the common part between $F_1$ and $S_2$ (vertex) is less than the lower dimension of the arguments (2).
- The result of $Cut12$ operation is a compound containing split part of the argument $F_1$. In this case argument face $F_1$ has a common part with solid $S_2$ so the corresponding part is not included into the result. The yellow contour is not a part of the result. It only shows the place of $S_2$. 
• The result of *Cut21* operation is not defined because the dimension of the face (2) is less than the dimension of the solid (3).

**Case 19: Two intersecting Solids.**

Let us consider two intersecting solids *S1* and *S2*:

• The result of *Fuse* operation is a compound composed from the split parts of arguments *S11*, *S12* and *S22* (*Cut12*, *Common*, *Cut21*). All inner webs are removed, so the result is one new solid *R*.

• The result of *Common* operation is a compound containing split parts of arguments i.e. one new solid *S12*. In this case solid *S12* is
common for the images of $S1$ and $S2$. The common part between the solids (solid) has the same dimension (3) as the dimension of the arguments (3). The yellow contour is not a part of the result. It only shows the place of $S1$.

- The result of $Cut12$ operation is a compound containing split part of the argument $S1$, i.e. 1 new solid $S11$.

- The result of $Cut21$ operation is a compound containing split part of the argument $S2$, i.e. 1 new solid $S21$.

**Case 20: Two Solids that have overlapping faces.**

Let us consider two solids $S1$ and $S2$ that have a common part on face:
- The result of **Fuse** operation is a compound composed from the split parts of arguments $S_{11}$, $S_{12}$ and $S_{22}$ (Cut12, Common, Cut21). All inner webs are removed, so the result is one new solid $R$.

- The result of **Common** operation is an empty compound because the dimension (2) of the common part between $S_1$ and $S_2$ (face) is less than the lower dimension of the arguments (3).
- The result of **Cut12** operation is a compound containing split part of the argument $S_1$, i.e. 1 new solid $S_{11}$.

- The result of **Cut21** operation is a compound containing split part of the argument $S_2$, i.e. 1 new solid $S_{21}$. 
Case 21: Two Solids that have overlapping edges.

Let us consider two solids $S_1$ and $S_2$ that have overlapping edges:

- The result of *Fuse* operation is a compound composed from the split parts of arguments i.e. 2 new solids $S_{11}$ and $S_{21}$. These solids have one shared edge $E_{n1}$. 
The result of *Common* operation is an empty compound because the dimension (1) of the common part between $S1$ and $S2$ (edge) is less than the lower dimension of the arguments (3).

The result of *Cut12* operation is a compound containing split part of the argument $S1$. In this case argument $S1$ has a common part with solid $S2$ so the corresponding part is not included into the result.

The result of *Cut21* operation is a compound containing split part of the argument $S2$. In this case argument $S2$ has a common part with solid $S1$ so the corresponding part is not included into the result.
Case 22: Two Solids that have overlapping vertices.

Let us consider two solids $S_1$ and $S_2$ that have overlapping vertices:

- The result of $Fuse$ operation is a compound composed from the split parts of arguments i.e. 2 new solids $S_{11}$ and $S_{21}$. These solids share $V_{n1}$. 
• The result of *Common* operation is an empty compound because the dimension (0) of the common part between $S1$ and $S2$ (vertex) is less than the lower dimension of the arguments (3).

• The result of *Cut12* operation is a compound containing split part of the argument $S1$.

• The result of *Cut21* operation is a compound containing split part of the argument $S2$. 
Case 23: A Shell and a Wire cut by a Solid.

Let us consider Shell $Sh$ and Wire $W$ as the objects and Solid $S$ as the tool:

- The result of $Fuse$ operation is not defined as the dimension of the arguments is not the same.
- The result of $Common$ operation is a compound containing the parts of the initial Shell and Wire common for the Solid. The new Shell and Wire are created from the objects.
• The result of \textit{Cut12} operation is a compound containing new Shell and Wire split from the arguments \textit{Sh} and \textit{W}. In this case they have a common part with solid \textit{S} so the corresponding part is not included into the result.

• The result of \textit{Cut21} operation is not defined as the objects have a lower dimension than the tool.

\textbf{Case 24: Two Wires that have overlapping edges.}

Let us consider two Wires that have overlapping edges, \textit{W1} is the object and \textit{W2} is the tool:
The result of *Fuse* operation is a compound containing two Wires, which share an overlapping edge. The new Wires are created from the objects:

The result of *Common* operation is a compound containing one Wire consisting of an overlapping edge. The new Wire is created from the objects:
• The result of Cut12 operation is a compound containing a wire split from object W1. Its common part with W2 is not included into the result.

• The result of Cut21 operation is a compound containing a wire split from W2. Its common part with W1 is not included into the result.
**Class BOPAlgo_BOP**

BOA is implemented in the class *BOPAlgo_BOP*. The main fields of this class are described in the Table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>myOperation</td>
<td>The type of the Boolean operation (Common, Fuse, Cut)</td>
</tr>
<tr>
<td>myTools</td>
<td>The tools</td>
</tr>
<tr>
<td>myDims[2]</td>
<td>The values of the dimensions of the arguments</td>
</tr>
<tr>
<td>myRC</td>
<td>The draft result (shape)</td>
</tr>
</tbody>
</table>

The main steps of the *BOPAlgo_BOP* are the same as of *BOPAlgo_Builder* except for some aspects described in the next paragraphs.
Building Draft Result

The input data for this step is as follows:

- \textit{BOPAlgo\_BOP} object after building result of type \textit{Compound};
- \textit{Type} of the Boolean operation.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For the Boolean operation \textit{Fuse} add to \textit{myRC} all images of arguments.</td>
<td>\textit{BOPAlgo_BOP::BuildRC()}</td>
</tr>
<tr>
<td>2</td>
<td>For the Boolean operation \textit{Common} or \textit{Cut} add to \textit{myRC} all images of argument \textit{S1} that are \textit{Common} for the Common operation and are \textit{Not Common} for the Cut operation</td>
<td>\textit{BOPAlgo_BOP::BuildRC()}</td>
</tr>
</tbody>
</table>
Building the Result

The input data for this step is as follows:

- *BOPAlgo_BOP* object the state after building draft result.

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For the Type of the Boolean operation Common, Cut with any dimension and operation Fuse with myDim[0] &lt; 3</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Find containers (WIRE, SHELL, COMPSOLID) in the arguments</td>
<td><em>BOPAlgo_BOP:: BuildShape()</em></td>
</tr>
<tr>
<td>1.2</td>
<td>Make connexity blocks from splits of each container that are in myRC</td>
<td><em>BOPTools Tools::MakeConnexityBlocks()</em></td>
</tr>
<tr>
<td>1.3</td>
<td>Build the result from shapes made from the connexity blocks</td>
<td><em>BOPAlgo_BOP:: BuildShape()</em></td>
</tr>
<tr>
<td>1.4</td>
<td>Add the remaining shapes from myRC to the result</td>
<td><em>BOPAlgo_BOP:: BuildShape()</em></td>
</tr>
<tr>
<td>2</td>
<td>For the Type of the Boolean operation Fuse with myDim[0] = 3</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Find internal faces (FWi) in myRC</td>
<td><em>BOPAlgo_BOP::BuildSolid()</em></td>
</tr>
<tr>
<td>2.2</td>
<td>Collect all faces of myRC except for internal faces (FWi) -&gt;</td>
<td><em>BOPAlgo_BOP::BuildSolid ()</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td><strong>2.3</strong></td>
<td><strong>2.4</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SFS</strong></td>
<td><strong>SFS</strong></td>
<td></td>
</tr>
<tr>
<td>Build solids ((SD_i)) from <strong>SFS</strong>.</td>
<td>Add the solids ((SD_i)) to the result.</td>
<td></td>
</tr>
<tr>
<td><strong>BOPAlgo_BuilderSolid</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Section Algorithm

Arguments

The arguments of BOA are shapes in terms of TopoDS_Shape. The main requirements for the arguments are described in the Algorithms.
Results and general rules

- The result of Section operation is a compound. Each sub-shape of the compound has shared sub-shapes in accordance with interferences between the arguments.
- The result of Section operation contains shapes that have dimension that is less then 2 i.e. vertices and edges.
- The result of Section operation contains standalone vertices if these vertices do not belong to the edges of the result.
- The result of Section operation contains vertices and edges of the arguments (or images of the arguments) that belong to at least two arguments (or two images of the arguments).
- The result of Section operation contains Section vertices and edges obtained from Face/Face interferences.
- The result of Section operation contains vertices that are the result of interferences between vertices and faces.
- The result of Section operation contains edges that are the result of interferences between edges and faces (Common Blocks),
Examples

Case 1: Two Vertices

Let us consider two interfering vertices: $V1$ and $V2$.

The result of Section operation is the compound that contains a new vertex $V$.

Case 1: Case 2: A Vertex and an Edge

Let us consider vertex $V1$ and the edge $E2$, that intersect in a 3D point:

The result of Section operation is the compound that contains vertex $V1$. 
Case 1: Case 2: A Vertex and a Face

Let us consider vertex V1 and face F2, that intersect in a 3D point:

The result of Section operation is the compound that contains vertex V1.

Case 4: A Vertex and a Solid

Let us consider vertex V1 and solid Z2. The vertex V1 is inside the solid Z2.
The result of Section operation is an empty compound.

**Case 5: Two edges intersecting at one point**

Let us consider edges $E1$ and $E2$, that intersect in a 3D point:

The result of Section operation is the compound that contains a new vertex $V_{new}$.

**Case 6: Two edges having a common block**

Let us consider edges $E1$ and $E2$, that have a common block:

The result of Section operation is the compound that contains a new edge $E_{new}$. 
Case 7: An Edge and a Face intersecting at a point

Let us consider edge $E1$ and face $F2$, that intersect at a 3D point:

The result of Section operation is the compound that contains a new vertex $V_{new}$.

Case 8: A Face and an Edge that have a common block

Let us consider edge $E1$ and face $F2$, that have a common block:
The result of *Section* operation is the compound that contains new edge $E_{new}$.

**Case 9: An Edge and a Solid intersecting at a point**

Let us consider edge $E1$ and solid $Z2$, that intersect at a point:

The result of *Section* operation is the compound that contains a new vertex $V_{new}$.

**Case 10: An Edge and a Solid that have a common block**

Let us consider edge $E1$ and solid $Z2$, that have a common block at a face:
The result of *Section* operation is the compound that contains a new edge $E_{new}$.

**Case 11: Two intersecting faces**

Let us consider two intersecting faces $F1$ and $F2$:

The result of *Section* operation is the compound that contains a new edge $E_{new}$. 
Case 12: Two faces that have a common part

Let us consider two faces $F_1$ and $F_2$ that have a common part:

The result of *Section* operation is the compound that contains 4 new edges.

Case 13: Two faces that have overlapping edges

Let us consider two faces $F_1$ and $F_2$ that have a overlapping edges:

The result of *Section* operation is the compound that contains a new edge $E_{\text{new}}$. 
Case 14: Two faces that have overlapping vertices

Let us consider two faces $F_1$ and $F_2$ that have overlapping vertices:

The result of Section operation is the compound that contains a new vertex $V_{new}$.

Case 15: A Face and a Solid that have an intersection curve

Let us consider face $F_1$ and solid $Z_2$ that have an intersection curve:
The result of Section operation is the compound that contains new edges.

Case 16: A Face and a Solid that have overlapping faces.

Let us consider face $F1$ and solid $Z2$ that have overlapping faces:

The result of Section operation is the compound that contains new edges.

Case 17: A Face and a Solid that have overlapping edges.
Let us consider face $F_1$ and solid $Z_2$ that have a common part on edge:

![Diagram showing face $F_1$ and solid $Z_2$ with a common edge $E_{new}$](image1)

The result of *Section* operation is the compound that contains a new edge $E_{new}$.

**Case 18: A Face and a Solid that have overlapping vertices.**

Let us consider face $F_1$ and solid $Z_2$ that have overlapping vertices:

![Diagram showing face $F_1$ and solid $Z_2$ with overlapping vertices and new vertex $V_{new}$](image2)

The result of *Section* operation is the compound that contains a new vertex $V_{new}$.
Case 19: Two intersecting Solids

Let us consider two intersecting solids $Z1$ and $Z2$:

The result of *Section* operation is the compound that contains new edges.

Case 20: Two Solids that have overlapping faces

Let us consider two solids $Z1$ and $Z2$ that have a common part on face:
The result of *Section* operation is the compound that contains new edges.

**Case 21: Two Solids that have overlapping edges**

Let us consider two solids $Z_1$ and $Z_2$ that have overlapping edges:

The result of *Section* operation is the compound that contains a new edge $E_{\text{new}}$.

**Case 22: Two Solids that have overlapping vertices**

Let us consider two solids $Z_1$ and $Z_2$ that have overlapping vertices:
The result of *Section* operation is the compound that contains a new vertex $V_{new}$. 
Class BOPAlgo_Section

SA is implemented in the class BOPAlgo_Section. The class has no specific fields. The main steps of the BOPAlgo_Section are the same as of BOPAlgo_Builder except for the following steps:

- Build Images for Wires;
- Build Result of Type Wire;
- Build Images for Faces;
- Build Result of Type Face;
- Build Images for Shells;
- Build Result of Type Shell;
- Build Images for Solids;
- Build Result of Type Solid;
- Build Images for Type CompSolid;
- Build Result of Type CompSolid;
- Build Images for Compounds; Some aspects of building the result are described in the next paragraph
# Building the Result

<table>
<thead>
<tr>
<th>No</th>
<th>Contents</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Build the result of the operation using all information contained in <em>FaceInfo</em>, Common Block, Shared entities of the arguments, etc.</td>
<td><em>BOPAlgo_Section::BuildSection()</em></td>
</tr>
</tbody>
</table>
Volume Maker Algorithm

The Volume Maker algorithm has been designed for building the elementary volumes (solids) from a set of connected, intersecting, or nested shapes. The algorithm can also be useful for splitting solids into parts, or constructing new solid(s) from set of intersecting or connected faces or shells. The algorithm creates only closed solids. In general case the result solids are non-manifold: fragments of the input shapes (wires, faces) located inside the solids are added as internal sub-shapes to these solids. But the algorithm allows preventing the addition of the internal for solids parts into result. In this case the result solids will be manifold and not contain any internal parts. However, this option does not prevent from the occurrence of the internal edges or vertices in the faces. Non-closed faces, free wires etc. located outside of any solid are always excluded from the result.

The Volume Maker algorithm is implemented in the class BOPAlgo_MakerVolume. It is based on the General Fuse (GF) algorithm. All the options of the GF algorithm such as possibility to run algorithm in parallel mode, fuzzy option, safe mode, glue options and history support are also available in this algorithm.

The requirements for the arguments are the same as for the arguments of GF algorithm - they could be of any type, but each argument should be valid and not self-interfered.

The algorithm allows disabling the calculation of intersections among the arguments. In this case the algorithm will run much faster, but the user should guarantee that the arguments do not interfere with each other, otherwise the result will be invalid (e.g. contain unexpected parts) or empty. This option is useful e.g. for building a solid from the faces of one shell or from the shapes that have already been intersected.
### Usage

#### C++ Level

The usage of the algorithm on the API level:

```cpp
BOPAlgo_MakerVolume aMV;
BOPCol_ListOfShape aLS = ...; // arguments
Standard_Boolean bRunParallel = Standard_False; /*
   parallel or single mode (the default value is FALSE)*/
Standard_Boolean bIntersect = Standard_True; /*
   intersect or not the arguments (the default
   value is TRUE)*/
Standard_Real aTol = 0.0; /* fuzzy option (default
   value is 0)*/
Standard_Boolean bSafeMode = Standard_False; /*
   protect or not the arguments from modification*/
BOPAlgo_Glue aGlue = BOPAlgo_GlueOff; /* Glue option
to speed up intersection of the arguments*/
Standard_Boolean bAvoidInternalShapes =
   Standard_False; /* Avoid or not the internal for
solids shapes in the result*/

//
aMV.SetArguments(aLS);
aMV.SetRunParallel(bRunParallel);
aMV.SetIntersect(bIntersect);
aMV.SetFuzzyValue(aTol);
aMV.SetNonDestructive(bSafeMode);
aMV.SetGlue(aGlue);
aMV.SetAvoidInternalShapes(bAvoidInternalShapes);
//
aMV.Perform(); //perform the operation
if (aMV.HasErrors()) { //check error status
    return;
}
```
const TopoDS_Shape& aResult = aMV.Shape(); // result of the operation

**Tcl Level**

To use the algorithm in Draw the command `mkvolume` has been implemented. The usage of this command is following:

**Usage:** `mkvolume r b1 b2 ... [-c] [-ni] [-ai]`

**Options:**
- `-c` - use this option to have input compounds considered as set of separate arguments (allows passing multiple arguments as one compound);
- `-ni` - use this option to disable the intersection of the arguments;
- `-ai` - use this option to avoid internal for solids shapes in the result.
Examples

Example 1

Creation of 9832 solids from sphere and set of 63 planes:

Example 2

Creating compartments on a ship defined by hull shell and a set of planes. The ship is divided on compartments by five transverse bulkheads and a deck – six compartments are created:
Cells Builder algorithm

The Cells Builder algorithm is an extension of the General Fuse algorithm. The result of General Fuse algorithm contains all split parts of the arguments. The Cells Builder algorithm provides means to specify if any given split part of the arguments (referred to as Cell) can be taken or avoided in the result.

The possibility of selecting any Cell allows combining any possible result and gives the Cells Builder algorithm a very wide sphere of application - from building the result of any Boolean operation to building the result of any application-specific operation.

The algorithm builds Cells only once and then just reuses them for combining the result. This gives this algorithm the performance advantage over Boolean operations, which always rebuild the splits to obtain the desirable result.

Thus, the Cells Builder algorithm can be especially useful for simulating Boolean expressions, i.e. a sequence of Boolean operations on the same arguments. Instead of performing many Boolean operations it allows getting the final result in a single operation. The Cells Builder will also be beneficial to obtain the results of different Boolean operations on the same arguments - Cut and Common, for example.

The Cells Builder algorithm also provides the possibility to remove any internal boundaries between splits of the same type, i.e. to fuse any same-dimensional parts added into the result and to keep any other parts as separate. This possibility is implemented through the Cells material approach: to remove the boundary between two Cells, both Cells should be assigned with the same material ID. However, if the same material ID has been assigned to the Cells of different dimension, the removal of the internal boundaries for that material will not be performed. Currently, such case is considered a limitation for the algorithm.

The algorithm can also create containers from the connected Cells added into result - WIRES from Edges, SHELLS from Faces and COMPSOLID from Solids.
Usage

The algorithm has been implemented in the `BOPAlgo_CellsBuilder` class.

Cells Builder is based on the General Fuse algorithm. Thus all options of the General Fuse algorithm, such as parallel processing mode, fuzzy mode, safe processing mode, gluing mode and history support are also available in this algorithm.

The requirements for the input shapes are the same as for General Fuse - each argument should be valid in terms of `BRepCheck_Analyzer` and `BOPAlgo_ArgumentAnalyzer`.

The result of the algorithm is a compound containing the selected parts of the basic type (VERTEX, EDGE, FACE or SOLID). The default result is an empty compound. It is possible to add any Cell by using the methods `AddToResult()` and `AddAllToResult()`. It is also possible to remove any part from the result by using methods `RemoveFromResult()` and `RemoveAllFromResult()`. The method `RemoveAllFromResult()` is also suitable for clearing the result.

The Cells that should be added/removed to/from the result are defined through the input shapes containing the parts that should be taken *(ShapesToTake)* and the ones containing parts that should be avoided *(ShapesToAvoid)*. To be taken into the result the part must be IN all shapes from *ShapesToTake* and OUT of all shapes from *ShapesToAvoid*.

To remove Internal boundaries, it is necessary to set the same material to the Cells, between which the boundaries should be removed, and call the method `RemoveInternalBoundaries()`. The material should not be equal to 0, as this is the default material ID. The boundaries between Cells with this material ID will not be removed. The same Cell cannot be added with different materials. It is also possible to remove the boundaries when the result is combined. To do this, it is necessary to set the material for parts (not equal to 0) and set the flag `bUpdate` to TRUE. If the same material ID has been set for parts of different dimension, the removal of internal boundaries for this material will not be performed.

It is possible to create typed Containers from the parts added into result
by using method *MakeContainers()*.
The type of the containers will depend on the type of the input shapes: WIRES for EDGE, SHELLS for FACES and COMPSOLIDS for SOLIDS. The result will be a compound containing containers.

**API usage**

Here is the example of the algorithm use on the API level:

```cpp
BOPAlgo_CellsBuilder aCBuilder;
BOPCol_ListOfShape aLS = ...; // arguments
Standard_Boolean bRunParallel = Standard_False; /* parallel or single mode (the default value is FALSE)*/
Standard_Real aTol = 0.0; /* fuzzy option (the default value is 0)*/
Standard_Boolean bSafeMode = Standard_False; /* protect or not the arguments from modification*/
BOPAlgo_Glue aGlue = BOPAlgo_GlueOff; /* Glue option to speed up the intersection of arguments*/

//
aCBuilder.SetArguments(aLS);
aCBuilder.SetRunParallel(bRunParallel);
aCBuilder.SetFuzzyValue(aTol);
aCBuilder.SetNonDestructive(bSafeMode);
aCBuilder.SetGlue(aGlue);

//
aCBuilder.Perform(); // build splits of all arguments (GF)
if (aCBuilder.HasErrors()) { // check error status
    return;
}

// collecting of the cells into result
const TopoDS_Shape& anEmptyRes = aCBuilder.Shape(); // empty result, as nothing has been added yet
const TopoDS_Shape& anAllCells =
```
aCBuilder.GetAllParts(); // all split parts

//
BOPCol_ListOfShape aLSToTake = ...; // parts of these arguments will be taken into result
BOPCol_ListOfShape aLSToAvoid = ...; // parts of these arguments will not be taken into result

//
Standard_Integer iMaterial = 1; // defines the material for the cells
Standard_Boolean bUpdate = Standard_False; // defines whether to update the result right now or not
// adding to result
aCBuilder.AddToResult(aLSToTake, aLSToAvoid, iMaterial, bUpdate);
aCBuilder.RemoveInternalBoundaries(); // removing of the boundaries
TopoDS_Shape aResult = aCBuilder.Shape(); // the result
// removing from result
aCBuilder.AddAllToResult();
aCBuilder.RemoveFromResult(aLSToTake, aLSToAvoid);
aResult = aCBuilder.Shape(); // the result

**DRAW usage**

The following set of new commands has been implemented to run the algorithm in DRAW Test Harness:

```plaintext
bcbuild            : Initialization of the Cells Builder. Use: *bcbuild r*
bcatid            : Add parts to result. Use: *bcadd r s1 (0,1) s2 (0,1) ... [-m material [-u]]*
bcatidall         : Add all parts to result. Use: *bcaddall r [-m material [-u]]*
bcremove          : Remove parts from result. Use: *bcremove r s1 (0,1) s2 (0,1) ...*
bcremoveall       : Remove all parts from result. Use: *
```
Here is the example of the algorithm use on the DRAW level:

```plaintext
psphere s1 15
psphere s2 15
psphere s3 15
ttranslate s1 0 0 10
ttranslate s2 20 0 10
ttranslate s3 10 0 0
bclearobjects; bcleartools
baddobjects s1 s2 s3
bfillds
# rx will contain all split parts
bcbuild rx
# add to result the part that is common for all three spheres
bcadd res s1 1 s2 1 s3 1 -m 1
# add to result the part that is common only for first and third spheres
bcadd res s1 1 s2 0 s3 1 -m 1
# remove internal boundaries
bcremoveint res
```
Examples

The following simple example illustrates the possibilities of the algorithm working on a cylinder and a sphere intersected by a plane:

\begin{verbatim}
pcylinder c 10 30
psphere s 15
ttranslate s 0 0 30
plane p 0 0 20 1 0 0
mkface f p -25 30 -17 17
\end{verbatim}

Arguments

1. Common for all arguments
The result of COMMON operation

2. Common between cylinder and face

The result of COMMON operation between cylinder and face

3. Common between cylinder and sphere
The result of COMMON operation between cylinder and sphere

4. Fuse of cylinder and sphere

```
bcremoveall
bcadd res c 1 -m 1
bcadd res s 1 -m 1
bcremoveint res
```

The result of FUSE operation between cylinder and sphere

5. Parts of the face inside solids - FUSE(COMMON(f, c), COMMON(f, s))

```
bcremoveall
bcadd res f 1 s 1 -m 1
bcadd res f 1 c 1 -m 1
```
Parts of the face inside solids

bcremoveint res

Unified parts of the face inside solids

6. Part of the face outside solids

bcremoveall
bcadd res f 1 c 0 s 0
7. Fuse operation (impossible using standard Boolean Fuse operation)

```
bcremoveall
bcadd res c 1 -m 1
bcadd res s 1 -m 1
bcadd res f 1 c 0 s 0
bcremoveint res
```
Fuse operation

These examples may last forever. To define any new operation, it is just necessary to define, which Cells should be taken and which should be avoided.
Algorithm Limitations

The chapter describes the problems that are considered as Algorithm limitations. In most cases an Algorithm failure is caused by a combination of various factors, such as self-interfered arguments, inappropriate or ungrounded values of the argument tolerances, adverse mutual position of the arguments, tangency, etc.

A lot of failures of GFA algorithm can be caused by bugs in low-level algorithms: Intersection Algorithm, Projection Algorithm, Approximation Algorithm, Classification Algorithm, etc.

- The Intersection, Projection and Approximation Algorithms are mostly used at the Intersection step. Their bugs directly cause wrong section results (i.e. incorrect section edges, section points, missing section edges or micro edges). It is not possible to obtain a correct final result of the GFA if a section result is wrong.
- The Projection Algorithm is used at the Intersection step. The purpose of Projection Algorithm is to compute 2D curves on surfaces. Wrong results here lead to incorrect or missing faces in the final GFA result.
- The Classification Algorithm is used at the Building step. The bugs in the Classification Algorithm lead to errors in selecting shape parts (edges, faces, solids) and ultimately to a wrong final GFA result.

The description below illustrates some known GFA limitations. It does not enumerate exhaustively all problems that can arise in practice. Please, address cases of Algorithm failure to the OCCT Maintenance Service.
Arguments

Common requirements

Each argument should be valid (in terms of BRepCheck_Analyzer), or conversely, if the argument is considered as non-valid (in terms of BRepCheck_Analyzer), it cannot be used as an argument of the algorithm.

The class BRepCheck_Analyzer is used to check the overall validity of a shape. In OCCT a Shape (or its sub-shapes) is considered valid if it meets certain criteria. If the shape is found as invalid, it can be fixed by tools from ShapeAnalysis, ShapeUpgrade and ShapeFix packages.

However, it is important to note that class BRepCheck_Analyzer is just a tool that can have its own problems; this means that due to a specific factor(s) this tool can sometimes provide a wrong result.

Let us consider the following example:

The Analyzer checks distances between couples of 3D check-points \((Pi, PSI)\) of edge \(E\) on face \(F\). Point \(Pi\) is obtained from the 3D curve (at the parameter \(ti\)) of the edge. \(PSi\) is obtained from 2D curve (at the parameter \(ti\)) of the edge on surface \(S\) of face \(F\). To be valid the distance should be less than \(Tol(E)\) for all couples of check-points. The number of these check-points is a predefined value (e.g. 23).

Let us consider the case when edge \(E\) is recognized valid (in terms of BRepCheck_Analyzer).

Further, after some operation, edge \(E\) is split into two edges \(E1\) and \(E2\). Each split edge has the same 3D curve and 2D curve as the original edge \(E\).

Let us check \(E1\) (or \(E2\)). The Analyzer again checks the distances between the couples of check-points points \((Pi, PSI)\). The number of these check-points is the same constant value (23), but there is no guarantee that the distances will be less than \(Tol(E)\), because the points chosen for \(E1\) are not the same as for \(E\).
Thus, if $E1$ is recognized by the Analyzer as non-valid, edge $E$ should also be non-valid. However $E$ has been recognized as valid. Thus the Analyzer gives a wrong result for $E$.

The fact that the argument is a valid shape (in terms of BRepCheck_Analyzer) is a necessary but insufficient requirement to produce a valid result of the Algorithms.

**Pure self-interference**

The argument should not be self-interfered, i.e. all sub-shapes of the argument that have geometrical coincidence through any topological entities (vertices, edges, faces) should share these entities.

**Example 1: Compound of two edges**

The compound of two edges $E1$ and $E2$ is a self-interfered shape and cannot be used as the argument of the Algorithms.

![Compound of two edges](image)

**Example 2: Self-interfered Edge**

The edge $E$ is a self-interfered shape and cannot be used as an argument of the Algorithms.
Example 3: Self-interfered Face

The face $F$ is a self-interfered shape and cannot be used as an argument of the Algorithms.

Example 4: Face of Revolution

The face $F$ has been obtained by revolution of edge $E$ around line $L$. 
In spite of the fact that face $F$ is valid (in terms of $BRepCheck\_Analyzer$) it is a self-interfered shape and cannot be used as the argument of the Algorithms.

**Self-interferences due to tolerances**

**Example 1: Non-closed Edge**

Let us consider edge $E$ based on a non-closed circle.
The distance between the vertices of $E$ is $D=0.69799$. The values of the tolerances $Tol(V1)=Tol(V2)=0.5$.

In spite of the fact that the edge $E$ is valid in terms of $BRepCheck_Analyzer$, it is a self-interfered shape because its vertices are interfered. Thus, edge $E$ cannot be used as an argument of the Algorithms.

**Example 2: Solid containing an interfered vertex**

Let us consider solid $S$ containing vertex $V$. 
The value of tolerance $Tol(V) = 50.000075982061$.

In spite of the fact that solid $S$ is valid in terms of $BRepCheck_Analyzer$ it is a self-interfered shape because vertex $V$ is interfered with a lot of sub-shapes from $S$ without any topological connection with them. Thus solid $S$ cannot be used as an argument of the Algorithms.

**Parametric representation**

The parameterization of some surfaces (cylinder, cone, surface of revolution) can be the cause of limitation.

**Example 1: Cylindrical surface**

The parameterization range for cylindrical surface is:

$$ U: [0, 2\pi] , V: [-\infty, +\infty] $$
The range of $U$ coordinate is always restricted while the range of $V$ coordinate is non-restricted.

Let us consider a cylinder-based Face 1 with radii $R=3$ and $H=6$.

Let us also consider a cylinder-based Face 2 with radii $R=3000$ and $H=6000$ (resulting from scaling Face 1 with scale factor $ScF=1000$).
Please, pay attention to the Zoom value of the Figures.

It is obvious that starting with some value of ScF, e.g. ScF>1000000, all sloped p-Curves on Face 2 will be almost vertical. At least, there will be no difference between the values of angles computed by standard C Run-Time Library functions, such as double acos(double x). The loss of accuracy in computation of angles can cause failure of some BP sub-algorithms, such as building faces from a set of edges or building solids from a set of faces.

**Using tolerances of vertices to fix gaps**

It is possible to create shapes that use sub-shapes of lower order to avoid gaps in the tolerance-based data model.

Let us consider the following example:
Face $F$ has two edges $E1$ and $E2$ and two vertices, the base plane is $\{0,0,0, 0,0,1\}$;

- Edge $E1$ is based on line $\{0,0,0, 1,0,0\}$, $Tol(E1) = 1.e-7$;
- Edge $E2$ is based on line $\{0,1,0, 1,0,0\}$, $Tol(E2) = 1.e-7$;
- Vertex $V1$, point $\{0,0.5,0\}$, $Tol(V1) = 1$;
- Vertex $V2$, point $\{10,0.5,0\}$, $Tol(V2) = 1$;
- Face $F$ is valid (in terms of $BRepCheck\_Analyzer$).

The values of tolerances $Tol(V1)$ and $Tol(V2)$ are big enough to fix the gaps between the ends of the edges, but the vertices $V1$ and $V2$ do not contain any information about the trajectories connecting the corresponding ends of the edges. Thus, the trajectories are undefined. This will cause failure of some sub-algorithms of BP. For example, the sub-algorithms for building faces from a set of edges use the information about all edges connected in a vertex. The situation when a vertex has several pairs of edges such as above will not be solved in a right way.
**Intersection problems**

**Pure intersections and common zones**

**Example: Intersecting Edges**

Let us consider the intersection between two edges:

- $E_1$ is based on a line: $\{0, -10, 0, 1, 0, 0\}$, $Tol(E_1)=2$.
- $E_2$ is based on a circle: $\{0, 0, 0, 0, 0, 1\}$, $R=10$, $Tol(E_2)=2$.

The result of pure intersection between $E_1$ and $E_2$ is vertex $V_x \{0, -10, 0\}$.

The result of intersection taking into account tolerances is the common zone $CZ$ (part of 3D-space where the distance between the curves is less than or equals to the sum of edge tolerances).

The Intersection Part of Algorithms uses the result of pure intersection $V_x$ instead of $CZ$ for the following reasons:

- The Algorithms do not produce Common Blocks between edges
based on underlying curves of explicitly different type (e.g. Line / Circle). If the curves have different types, the rule of thumb is that the produced result is of type vertex. This rule does not work for non-analytic curves (Bezier, B-Spline) and their combinations with analytic curves.

- The algorithm of intersection between two surfaces \textit{IntPatch\_Intersection} does not compute CZ of the intersection between curves and points. So even if CZ were computed by Edge/Edge intersection algorithm, its result could not be treated by Face/Face intersection algorithm.

**Tolerances and inaccuracies**

The following limitations result from modeling errors or inaccuracies.

**Example: Intersection of planar faces**

Let us consider two planar rectangular faces \(F1\) and \(F2\).

The intersection curve between the planes is curve \(C12\). The curve produces a new intersection edge \(EC12\). The edge goes through vertices \(V1\) and \(V2\) thanks to big tolerance values of vertices \(Tol(V1)\) and \(Tol(V2)\). So, two straight edges \(E12\) and \(EC12\) go through two vertices, which is impossible in this case.

The problem cannot be solved in general, because the length of \(E12\) can
be infinite and the values of $Tol(V1)$ and $Tol(V2)$ theoretically can be infinite too.

In a particular case the problem can be solved in several ways:

- Reduce, if possible, the values of $Tol(V1)$ and $Tol(V2)$ (refinement of $F1$).
- Analyze the value of $Tol(EC12)$ and increase $Tol(EC12)$ to get a common part between the edges $EC12$ and $E12$. Then the common part will be rejected as there is an already existing edge $E12$ for face $F1$.

It is easy to see that if $C12$ is slightly above the tolerance spheres of $V1$ and $V2$ the problem does not appear.

**Example: Intersection of two edges**

Let us consider two edges $E1$ and $E2$, which have common vertices $V1$ and $V2$. The edges $E1$ and $E2$ have 3D-curves $C1$ and $C2$. $Tol(E1)=1.e^{-7}$, $Tol(E2)=1.e^{-7}$.

$C1$ practically coincides in 3D with $C2$. The value of deflection is $Dmax$ (e.g. $Dmax=1.e^{-6}$).

The evident and prospective result should be the Common Block between $E1$ and $E2$. However, the result of intersection differs.
The result contains three new vertices Vx1, Vx2 and Vx3, 8 new edges (V1, Vx1, Vx2, Vx3, V2) and no Common Blocks. This is correct due to the source data: \( \text{Tol}(E1)=1.e^{-7}, \text{Tol}(E2)=1.e^{-7} \) and \( D_{\text{max}}=1.e^{-6} \).

In this particular case the problem can be solved by several ways:

- Increase, if possible, the values \( \text{Tol}(E1) \) and \( \text{Tol}(E2) \) to get coincidence in 3D between \( E1 \) and \( E2 \) in terms of tolerance.
- Replace \( E1 \) by a more accurate model.

The example can be extended from 1D (edges) to 2D (faces).

**Intersecting Faces**

The comments and recommendations are the same as for 1D case above.

**Acquired Self-interferences**

**Example 1: Vertex and edge**

Let us consider vertex \( V1 \) and edge \( E2 \).

Vertex \( V1 \) interferes with vertices \( V12 \) and \( V22 \). So vertex \( V21 \) should interfere with vertex \( V22 \), which is impossible because vertices \( V21 \) and
V22 are the vertices of edge $E2$, thus $V21$ is not equal to $V22$.

The problem cannot be solved in general, because the length can be as small as possible to provide validity of $E2$ (in the extreme case: $\text{Length}(E2) = \text{Tol}(V21) + \text{Tol}(V22) + e$, where $e \to 0$).

In a particular case the problem can be solved by refinement of arguments, i.e. by decreasing the values of $\text{Tol}(V21)$, $\text{Tol}(V22)$ and $\text{Tol}(V1)$.

**Example 2: Vertex and wire**

Let us consider vertex $V2$ and wire consisting of edges $E11$ and $E12$.

The arguments themselves are not self-intersected. Vertex $V2$ interferes with edges $E11$ and $E12$. Thus, edge $E11$ should interfere with edge $E22$, but it is impossible because edges $E11$ and $E12$ cannot interfere by the condition.

The cases when a non-self-interfered argument (or its sub-shapes) become interfered due to the intersections with other arguments (or their
sub-shapes) are considered as limitations for the Algorithms.
Advanced Options

The previous chapters describe so called Basic Operations. Most of tasks can be solved using Basic Operations. Nonetheless, there are cases that can not be solved straightforwardly by Basic Operations. The tasks are considered as limitations of Basic Operations.

The chapter is devoted to Advanced Options. In some cases the usage of Advanced Options allows overcoming the limitations, improving the quality of the result of operations, robustness and performance of the operators themselves.
Fuzzy Boolean Operation

Fuzzy Boolean operation is the option of Basic Operations such as General Fuse, Splitting, Boolean, Section, Maker Volume and Cells building operations, in which additional user-specified tolerance is used. This option allows operators to handle robustly cases of touching and near-coincident, misaligned entities of the arguments.

The Fuzzy option is useful on the shapes with gaps or embeddings between the entities of these shapes, which are not covered by the tolerance values of these entities. Such shapes can be the result of modeling mistakes, or translating process, or import from other systems with loss of precision, or errors in some algorithms.

Most likely, the Basic Operations will give unsatisfactory results on such models. The result may contain unexpected and unwanted small entities, faulty entities (in terms of BRepCheck_Analyzer), or there can be no result at all.

With the Fuzzy option it is possible to get the expected result – it is just necessary to define the appropriate value of fuzzy tolerance for the operation. To define that value it is necessary to measure the value of the gap (or the value of embedding depth) between the entities of the models, slightly increase it (to make the shifted entities coincident in terms of their tolerance plus the additional one) and pass it to the algorithm.

Fuzzy option is included in interface of Intersection Part (class BOPAlgo_PaveFiller) and application programming interface (class BRepAlgoAPI_BooleanOperation)

Examples

The following examples demonstrate the advantages of usage Fuzzy option operations over the Basic Operations in typical situations.

Case 1
In this example the cylinder (shown in yellow and transparent) is subtracted from the box (shown in red). The cylinder is shifted by $5e^{-5}$ relatively to the box along its axis (the distance between rear faces of the box and cylinder is $5e^{-5}$).

The following results are obtained using Basic Operations and the Fuzzy ones with the fuzzy value $5e^{-5}$:

Result of CUT operation obtained with Basic Operations
In this example Fuzzy option allows eliminating a very thin part of the result shape produced by Basic algorithm due to misalignment of rear faces of the box and the cylinder.

**Case 2**

In this example two boxes are fused. One of them has dimensions $10 \times 10 \times 10$, and the other is $10 \times 10.000001 \times 10.000001$ and adjacent to the first one. There is no gap in this case as the surfaces of the neighboring faces coincide, but one box is slightly greater than the other.
The following results are obtained using Basic Operations and the Fuzzy ones with the fuzzy value $1e^{-6}$:

Result of CUT operation obtained with Basic Operations
Result of CUT operation obtained with Fuzzy Option

In this example Fuzzy option allows eliminating an extremely narrow face in the result produced by Basic operation.

Case 3

In this example the small planar face (shown in orange) is subtracted from the big one (shown in yellow). There is a gap $1 \times 10^{-5}$ between the edges of these faces.
The following results are obtained using Basic Operations and the Fuzzy ones with the fuzzy value $1e^{-5}$:

In this example Fuzzy options eliminated a pin-like protrusion resulting from the gap between edges of the argument faces.

**Case 4**

In this example the small edge is subtracted from the big one. The edges are overlapping not precisely, with max deviation between them equal to $5.28004e^{-5}$. We will use $6e^{-5}$ value for Fuzzy option.
The following results are obtained using Basic Operations and the Fuzzy ones with the fuzzy value $6\times10^{-5}$:

This example stresses not only the validity, but also the performance issue. The usage of Fuzzy option with the appropriate value allows processing the case much faster than with the pure Basic operation. The
performance gain for the case is 45 (Processor: Intel(R) Core(TM) i5-3450 CPU @ 3.10 GHz).
Gluing Operation

The Gluing operation is the option of the Basic Operations such as General Fuse, Splitting, Boolean, Section, Maker Volume and Cells building operations. It has been designed to speed up the computation of the interferences among arguments of the operations on special cases, in which the arguments may be overlapping but do not have real intersections between their sub-shapes.

This option cannot be used on the shapes having real intersections, like intersection vertex between edges, or intersection vertex between edge and a face or intersection line between faces:

There are two possibilities of overlapping shapes:

- The shapes can be partially coinciding - the faces do not have intersection curves, but overlapping. The faces of such arguments will be split during the operation. The following picture illustrates such shapes:
Partially coinciding faces

- The shapes can be fully coinciding - there should be no partial overlapping of the faces, thus no intersection of type EDGE/FACE at all. In such cases the faces will not be split during the operation.

Full coinciding faces of the boxes

Thus, there are two possible options - for full and partial coincidence of the shapes.

Even though there are no real intersections on such cases without Gluing options the algorithm will still intersect the sub-shapes of the arguments with interfering bounding boxes.
The performance improvement in gluing mode is achieved by excluding the most time consuming computations and in some case can go up to 90%:

- Exclude computation of FACE/FACE intersections for partial coincidence;
- Exclude computation of VERTEX/FACE, EDGE/FACE and FACE/FACE intersections for full coincidence.

By setting the Gluing option for the operation user should guarantee that the arguments are really coinciding. The algorithm does not check this itself. Setting inappropriate option for the operation is likely to lead to incorrect result.

**Usage**

The Gluing option is an enumeration implemented in BOPAlgo_GlueEnum.hxx:

- **BOPAlgo_GlueOff** - default value for the algorithms, Gluing is switched off;
- **BOPAlgo_GlueShift** - Glue option for shapes with partial coincidence;
- **BOPAlgo_GlueFull** - Glue option for shapes with full coincidence.

**API level**

For setting the Gluing options for the algorithm it is just necessary to call the SetGlue(const BOPAlgo_Glue) method with appropriate value:

```cpp
BOPAlgo_Builder aGF;

// ....

// setting the gluing option to speed up intersection of the arguments
aGF.SetGlue(BOPAlgo_GlueShift)

// ....
```
**TCL level**

For setting the Gluing options in DRAW it is necessary to call the *bglue* command with appropriate value:

- 0 - default value, Gluing is off;
- 1 - for partial coincidence;
- 2 - for full coincidence

```
bglue 1
```

**Examples**

**Case1 - Fusing the 64 bspline boxes into one solid**

![BSpline Boxes with partial coincidence](image)

Performance improvement from using the GlueShift option in this case is about 70 percent.

**Case2 - Sewing faces of the shape after reading from IGES**
Faces with coinciding but not shared edges

Performance improvement in this case is also about 70 percent.
Safe processing mode

The safe processing mode is the advanced option in Boolean Operation component. This mode can be applied to all Basic operations such as General Fuse, Splitting, Boolean, Section, Maker Volume, Cells building. This option allows keeping the input arguments untouched. In other words, switching this option on prevents the input arguments from any modification such as tolerance increase, addition of the P-Curves on edges, etc.

The option can be very useful for implementation of the Undo/Redo mechanism in the applications and allows performing the operation many times without changing the inputs.

By default the safe processing option is switched off for the algorithms. Enabling this option might slightly decrease the performance of the operation, because instead of the modification of some entity it will be necessary to create the copy of this entity and modify it. However, this degradation should be very small because the copying is performed only in case of necessity.

The option is also available in the Intersection algorithm - BOPAlgo_PaveFiller. To perform several different operations on the same arguments, the safe processing mode can be enabled in PaveFiller, prepared only once and then used in operations. It is enough to set this option to PaveFiller only and all algorithms taking this PaveFiller will also work in the safe mode.

Usage

API level

To enable/disable the safe processing mode for the algorithm, it is necessary to call SetNonDestructive() method with the appropriate value:

```cpp
BOPAlgo_Builder aGF;
//
....
```
// enabling the safe processing mode to prevent modification of the input shapes
aGF.SetNonDestructive(Standard_True);

TCL level

To enable the safe processing mode for the operation in DRAW, it is necessary to call the `bnondestructive` command with the appropriate value:

- 0 - default value, the safe mode is switched off;
- 1 - the safe mode will be switched on.

`bnondestructive 1`
Errors and warnings reporting system

The chapter describes the Error/Warning reporting system of the algorithms in the Boolean Component.

The errors and warnings are collected in the instance of the class Message_Report maintained as a field by common base class of Boolean operation algorithms BOPAlgo_Options.

The error is reported in for problems which cannot be treated and cause the algorithm to fail. In this case the result of the operation will be incorrect or incomplete or there will be no result at all.

The warnings are reported for the problems which can be potentially handled or ignored and thus do not cause the algorithms to stop their work (but probably affect the result).

All possible errors and warnings that can be set by the algorithm are listed in its header file. The complete list of errors and warnings that can be generated by Boolean operations is defined in BOPAlgo_Alerts.hxx.

Use method HasErrors() to check for presence of error; method HasError() can be used to check for particular error. Methods DumpErrors() outputs textual description of collected errors into the stream. Similar methods HasWarnings(), HasWarning(), and DumpWarnings() are provided for warnings.

Note that messages corresponding to errors and warnings are defined in resource file BOPAlgo.msg. These messages can be localized; for that put translated version to separate file and load it in the application by call to Message_MsgFile::Load().

Here is the example of how to use this system:

```cpp
BOPAlgo_PaveFiller aPF;
aPF.SetArguments(...);
aPF.Perform();
```
if (aPF.HasErrors()) {
    aPF.DumpErrors(std::cerr);
    //
    if (aPF.HasError(STANDARD_TYPE(BOPAlgo_AlertNullInputShapes)) {
        // some actions
    }
    if (aPF.HasWarning(STANDARD_TYPE(BOPAlgo_AlertTooSmallEdge)) {
        // some actions
    }
    ...
}

DRAW commands executing Boolean operations output errors and warnings generated by these operations in textual form. Additional option allows saving shapes for which warnings have been generated, as DRAW variables. To activate this option, run command `bdrawwarnshapes` with argument 1 (or with 0 to deactivate):

```
bdrawwarnshapes 1
```

After setting this option and running an algorithm the result will look as follows:

```
Warning: The interfering vertices of the same argument: ws_1_1 ws_1_2
Warning: The positioning of the shapes leads to creation of small edges without valid range: ws_2_1
```
Usage

The chapter contains some examples of the OCCT Boolean Component usage. The usage is possible on two levels: C++ and Tcl.
Package BRepAlgoAPI

The package BRepAlgoAPI provides the Application Programming Interface of the Boolean Component.

The package consists of the following classes:

- `BRepAlgoAPI_Alg`o – the root class that provides the interface for algorithms.
- `BRepAlgoAPI_BuilderAlg`o – the class API level of General Fuse algorithm.
- `BRepAlgoAPI_Spl`itter – the class API level of the Splitter algorithm.
- `BRepAlgoAPI_BooleanOperation` – the root class for the classes BRepAlgoAPI_Fuse, BRepAlgoAPI_Common, BRepAlgoAPI_Cut and BRepAlgoAPI_Section.
- `BRepAlgoAPI_Fuse` – the class provides Boolean fusion operation.
- `BRepAlgoAPI_Common` – the class provides Boolean common operation.
- `BRepAlgoAPI_Cut` – the class provides Boolean cut operation.
- `BRepAlgoAPI_Section` – the class provides Boolean section operation.

Diagram of BRepAlgoAPI package
The detailed description of the classes can be found in the corresponding .hxx files. The examples are below in this chapter.
Package BOPTest

The package BOPTest provides the usage of the Boolean Component on Tcl level. The method BOPTest::APICommands contains corresponding Tcl commands:

- \texttt{bapibuild} – for General Fuse Operator;
- \texttt{bapisplit} – for Splitter Operator;
- \texttt{bapibop} – for Boolean Operator and Section Operator.

The examples of how to use the commands are below in this chapter.

**Case 1. General Fuse operation**

The following example illustrates how to use General Fuse operator:

**C++ Level**

```cpp
#include <TopoDS_Shape.hxx>
#include <TopTools_ListOfShape.hxx>
#include <BRepAlgoAPI_BuilderAlgo.hxx>
{
  Standard_Boolean bRunParallel;
  Standard_Real aFuzzyValue;
  BRepAlgoAPI_BuilderAlgo aBuilder;
  //
  // prepare the arguments
  TopTools_ListOfShape& aLS=...;
  //
  bRunParallel=Standard_True;
  aFuzzyValue=2.1e-5;
  //
  // set the arguments
  aBuilder.SetArguments(aLS);
  // set parallel processing mode
  // if bRunParallel= Standard_True : the parallel
```
processing is switched on
// if bRunParallel= Standard_False : the parallel processing is switched off
aBuilder.SetRunParallel(bRunParallel);
//
// set Fuzzy value
// if aFuzzyValue=0.: the Fuzzy option is off
// if aFuzzyValue>0.: the Fuzzy option is on
aBuilder.SetFuzzyValue(aFuzzyValue);
//
// safe mode - avoid modification of the arguments
Standard_Boolean bSafeMode = Standard_True;
// if bSafeMode == Standard_True  - the safe mode is switched on
// if bSafeMode == Standard_False - the safe mode is switched off
aBuilder.SetNonDestructive(bSafeMode);
//
// gluing options - for coinciding arguments
BOPAlgo_GlueEnum aGlueOpt = BOPAlgo_GlueFull;
// if aGlueOpt == BOPAlgo_GlueOff  - the gluing mode is switched off
// if aGlueOpt == BOPAlgo_GlueShift - the gluing mode is switched on
// if aGlueOpt == BOPAlgo_GlueFull  - the gluing mode is switched on
aBuilder.SetGlue(aGlueOpt);
//
// run the algorithm
aBuilder.Build();
if (aBuilder.HasErrors()) {
    // an error treatment
    return;
}
//
// result of the operation aR
const TopoDS_Shape& aR=aBuilder.Shape();
Tcl Level

# prepare the arguments
box b1 10 10 10
box b2 3 4 5 10 10 10
box b3 5 6 7 10 10 10
#
# clear inner contents
bclearobjects; bcleartools;
#
# set the arguments
baddobjects b1 b2 b3
# set parallel processing mode
# 1: the parallel processing is switched on
# 0: the parallel processing is switched off
brunparallel 1
#
# set Fuzzy value
# 0. : the Fuzzy option is off
# >0. : the Fuzzy option is on
bfuzzyvalue 0.
#
# set safe processing mode
bnondestructive 1
# set safe mode
# 1 - the safe processing mode is switched on
# 0 - the safe processing mode is switched off
#
# set gluing mode
bglue 1
# set the gluing mode
# 1 or 2 - the gluing mode is switched on
# 0 - the gluing mode is switched off
#
Case 2. Splitting operation

The following example illustrates how to use the Splitter operator:

C++ Level

```cpp
#include <TopoDS_Shape.hxx>
#include <TopTools_ListOfShape.hxx>
#include <BRepAlgoAPI_Splitter.hxx>

BRepAlgoAPI_BuilderAlgo aSplitter;

// prepare the arguments
// objects
TopTools_ListOfShape& aLSObjects = ...;
// tools
TopTools_ListOfShape& aLSTools = ...;

// set the arguments
aSplitter.SetArguments(aLSObjects);
aSplitter.SetTools(aLSTools);

// set options
// parallel processing mode
Standard_Boolean bRunParallel = Standard_True;
// bRunParallel == Standard_True - the parallel processing is switched on
// bRunParallel == Standard_False - the parallel processing is switched off
aSplitter.SetRunParallel();

// fuzzy value - additional tolerance for the operation
```
Standard_Real aFuzzyValue = 1.e-5;
// if aFuzzyValue == 0. - the Fuzzy option is off
// if aFuzzyValue > 0. - the Fuzzy option is on
aSplitter.SetFuzzyValue(aFuzzyValue);

// safe mode - avoid modification of the arguments
Standard_Boolean bSafeMode = Standard_True;
// if bSafeMode == Standard_True - the safe mode is switched on
// if bSafeMode == Standard_False - the safe mode is switched off
aSplitter.SetNonDestructive(bSafeMode);

// gluing options - for coinciding arguments
BOPAlgo_GlueEnum aGlueOpt = BOPAlgo_GlueFull;
// if aGlueOpt == BOPAlgo_GlueOff - the gluing mode is switched off
// if aGlueOpt == BOPAlgo_GlueShift - the gluing mode is switched on
// if aGlueOpt == BOPAlgo_GlueFull - the gluing mode is switched on
aSplitter.SetGlue(aGlueOpt);

// run the algorithm
aSplitter.Build();
// check error status
if (aSplitter.HasErrors()) {
    return;
}

// result of the operation aResult
const TopoDS_Shape& aResult = aSplitter.Shape();

Tcl Level

# prepare the arguments
# objects
box b1 10 10 10
box b2 7 0 0 10 10 10

# tools
plane p 10 5 5 0 1 0
mkface f p -20 20 -20 20
#
# clear inner contents
bclearobjects; bcleartools;
#
# set the objects
baddobjects b1 b2
# set the tools
baddtools f
#
# set parallel processing mode
# 1: the parallel processing is switched on
# 0: the parallel processing is switched off
brunparallel 1
#
# set Fuzzy value
# 0. : the Fuzzy option is off
# >0. : the Fuzzy option is on
bfuzzyvalue 0.
#
# set safe processing mode
bnondestructive 1
# set safe mode
# 1 - the safe processing mode is switched on
# 0 - the safe processing mode is switched off
#
# set gluing mode
bglue 1
# set the gluing mode
# 1 or 2 - the gluing mode is switched on
# 0 - the gluing mode is switched off
#
Case 3. Common operation

The following example illustrates how to use Common operation:

C++ Level

```cpp
#include <TopoDS_Shape.hxx>
#include <TopTools_ListOfShape.hxx>
#include <BRepAlgoAPI_Common.hxx>

{...
    Standard_Boolean bRunParallel;
    Standard_Real aFuzzyValue;
    BRepAlgoAPI_Common aBuilder;

    // prepare the arguments
    TopTools_ListOfShape& aLS=...;
    TopTools_ListOfShape& aLT=...;
    //
    bRunParallel=Standard_True;
    aFuzzyValue=2.1e-5;
    //
    // set the arguments
    aBuilder.SetArguments(aLS);
    aBuilder.SetTools(aLT);
    //
    // set parallel processing mode
    // if  bRunParallel= Standard_True :  the parallel processing is switched on
    // if  bRunParallel= Standard_False :  the parallel processing is switched off
    aBuilder.SetRunParallel(bRunParallel);
    //
    // set Fuzzy value
```
// if aFuzzyValue=0.: the Fuzzy option is off
// if aFuzzyValue>0.: the Fuzzy option is on
aBuilder.SetFuzzyValue(aFuzzyValue);

// safe mode - avoid modification of the arguments
Standard_Boolean bSafeMode = Standard_True;
// if bSafeMode == Standard_True - the safe mode is switched on
// if bSafeMode == Standard_False - the safe mode is switched off
aBuilder.SetNonDestructive(bSafeMode);

// gluing options - for coinciding arguments
BOPAlgo_GlueEnum aGlueOpt = BOPAlgo_GlueFull;
// if aGlueOpt == BOPAlgo_GlueOff - the gluing mode is switched off
// if aGlueOpt == BOPAlgo_GlueShift - the gluing mode is switched on
// if aGlueOpt == BOPAlgo_GlueFull - the gluing mode is switched on
aBuilder.SetGlue(aGlueOpt);

// run the algorithm
aBuilder.Build();
if (aBuilder.HasErrors()) {
    // an error treatment
    return;
}

// result of the operation aR
const TopoDS_Shape& aR=aBuilder.Shape();

Tcl Level

# prepare the arguments
box b1 10 10 10
box b2 7 0 4 10 10 10
box b3 14 0 0 10 10 10
#
# clear inner contents
bclearobjects; bcleartools;
#
# set the arguments
baddobjects b1 b3
baddtools b2
#
# set parallel processing mode
# 1: the parallel processing is switched on
# 0: the parallel processing is switched off
brunparallel 1
#
# set Fuzzy value
# 0: the Fuzzy option is off
# >0: the Fuzzy option is on
bfuzzyvalue 0.
#
# set safe processing mode
bnondestructive 1
# set safe mode
# 1 - the safe processing mode is switched on
# 0 - the safe processing mode is switched off
#
# set gluing mode
bglue 1
# set the gluing mode
# 1 or 2 - the gluing mode is switched on
# 0 - the gluing mode is switched off
#
# run the algorithm
# r is the result of the operation
# 0 means Common operation
bapibop r 0
Case 4. Fuse operation

The following example illustrates how to use Fuse operation:

C++ Level

```cpp
#include <TopoDS_Shape.hxx>
#include <TopTools_ListOfShape.hxx>
#include <BRepAlgoAPI_Fuse.hxx>
{
    Standard_Boolean bRunParallel;
    Standard_Real aFuzzyValue;
    BRepAlgoAPI_Fuse aBuilder;

    // prepare the arguments
    TopTools_ListOfShape& aLS=...;
    TopTools_ListOfShape& aLT=...;
    //
    bRunParallel=Standard_True;
    aFuzzyValue=2.1e-5;
    //
    // set the arguments
    aBuilder.SetArguments(aLS);
    aBuilder.SetTools(aLT);
    //
    // set parallel processing mode
    // if bRunParallel= Standard_True : the parallel processing is switched on
    // if bRunParallel= Standard_False : the parallel processing is switched off
    aBuilder.SetRunParallel(bRunParallel);
    //
    // set Fuzzy value
    // if aFuzzyValue=0. : the Fuzzy option is off
    // if aFuzzyValue>0. : the Fuzzy option is on
    aBuilder.SetFuzzyValue(aFuzzyValue);
    //
```
// safe mode - avoid modification of the arguments
Standard_Boolean bSafeMode = Standard_True;
// if bSafeMode == Standard_True   - the safe mode
  is switched on
// if bSafeMode == Standard_False - the safe mode
  is switched off
aBuilder.SetNonDestructive(bSafeMode);

// gluing options - for coinciding arguments
BOPAlgo_GlueEnum aGlueOpt = BOPAlgo_GlueFull;
// if aGlueOpt == BOPAlgo_GlueOff   - the gluing
  mode is switched off
// if aGlueOpt == BOPAlgo_GlueShift - the gluing
  mode is switched on
// if aGlueOpt == BOPAlgo_GlueFull   - the gluing
  mode is switched on
aBuilder.SetGlue(aGlueOpt);

// run the algorithm
aBuilder.Build();
if (aBuilder.HasErrors()) {
  // an error treatment
  return;
}

// result of the operation aR
const TopoDS_Shape& aR=aBuilder.Shape();
...

Tcl Level

# prepare the arguments
box b1 10 10 10
box b2 7 0 4 10 10 10
box b3 14 0 0 10 10 10
#
# clear inner contents
bclearobjects; bcleartools;
#
# set the arguments
baddobjects b1 b3
baddtools b2
#
# set parallel processing mode
# 1: the parallel processing is switched on
# 0: the parallel processing is switched off
brunparallel 1
#
# set Fuzzy value
# 0.: the Fuzzy option is off
# >0.: the Fuzzy option is on
bfuzzyvalue 0.
#
# set safe processing mode
bnondestructive 1
# set safe mode
# 1 - the safe processing mode is switched on
# 0 - the safe processing mode is switched off
#
# set gluing mode
bglue 1
# set the gluing mode
# 1 or 2 - the gluing mode is switched on
# 0 - the gluing mode is switched off
#
# run the algorithm
# r is the result of the operation
# 1 means Fuse operation
bapibop r 1

**Case 5. Cut operation**

The following example illustrates how to use Cut operation:
```cpp
#include <TopoDS_Shape.hxx>
#include <TopTools_ListOfShape.hxx>
#include <BRepAlgoAPI_Cut.hxx>
{
    Standard_Boolean bRunParallel;
    Standard_Real aFuzzyValue;
    BRepAlgoAPI_Cut aBuilder;

    // prepare the arguments
    TopTools_ListOfShape& aLS=...;
    TopTools_ListOfShape& aLT=...;
    //
    bRunParallel=Standard_True;
    aFuzzyValue=2.1e-5;
    //
    // set the arguments
    aBuilder.SetArguments(aLS);
    aBuilder.SetTools(aLT);
    //
    // set parallel processing mode
    // if bRunParallel= Standard_True : the parallel processing is switched on
    // if bRunParallel= Standard_False : the parallel processing is switched off
    aBuilder.SetRunParallel(bRunParallel);
    //
    // set Fuzzy value
    // if aFuzzyValue=0.: the Fuzzy option is off
    // if aFuzzyValue>0.: the Fuzzy option is on
    aBuilder.SetFuzzyValue(aFuzzyValue);
    //
    // safe mode - avoid modification of the arguments
    Standard_Boolean bSafeMode = Standard_True;
    // if bSafeMode == Standard_True - the safe mode is switched on
```
// if bSafeMode == Standard_False - the safe mode is switched off
aBuilder.SetNonDestructive(bSafeMode);

// gluing options - for coinciding arguments
BOPAlgo_GlueEnum aGlueOpt = BOPAlgo_GlueFull;
// if aGlueOpt == BOPAlgo_GlueOff  - the gluing mode is switched off
// if aGlueOpt == BOPAlgo_GlueShift  - the gluing mode is switched on
// if aGlueOpt == BOPAlgo_GlueFull  - the gluing mode is switched on
aBuilder.SetGlue(aGlueOpt);

// run the algorithm
aBuilder.Build();
if (aBuilder.HasErrors()) {
    // an error treatment
    return;
}

// result of the operation aR
const TopoDS_Shape& aR = aBuilder.Shape();
...

Tcl Level

# prepare the arguments
box b1 10 10 10
box b2 7 0 4 10 10 10
box b3 14 0 0 10 10 10
#
# clear inner contents
bclearobjects; bcleartools;
#
# set the arguments
baddobjects b1 b3
baddtools b2
#
# set parallel processing mode
# 1: the parallel processing is switched on
# 0: the parallel processing is switched off
brunparallel 1
#
# set Fuzzy value
# 0. : the Fuzzy option is off
# >0. : the Fuzzy option is on
bfuzzyvalue 0.
#
# set safe processing mode
bnondestructive 1
# set safe mode
# 1 - the safe processing mode is switched on
# 0 - the safe processing mode is switched off
# set gluing mode
#
bglue 1
# set the gluing mode
# 1 or 2 - the gluing mode is switched on
# 0 - the gluing mode is switched off
#
# run the algorithm
# r is the result of the operation
# 2 means Cut operation
bapibop r 2

Case 6. Section operation

The following example illustrates how to use Section operation:

C++ Level

#include <TopoDS_Shape.hxx>
```cpp
#include <TopTools_ListOfShape.hxx>
#include <BRepAlgoAPI_Section.hxx>
{
  Standard_Boolean bRunParallel;
  Standard_Real aFuzzyValue;
  BRepAlgoAPI_Section aBuilder;

  // prepare the arguments
  TopTools_ListOfShape& aLS = ...;
  TopTools_ListOfShape& aLT = ...;
  //
  bRunParallel = Standard_True;
  aFuzzyValue = 2.1e-5;
  //
  // set the arguments
  aBuilder.SetArguments(aLS);
  aBuilder.SetTools(aLT);
  //
  // set parallel processing mode
  // if bRunParallel = Standard_True : the parallel processing is switched on
  // if bRunParallel = Standard_False : the parallel processing is switched off
  aBuilder.SetRunParallel(bRunParallel);
  //
  // set Fuzzy value
  // if aFuzzyValue = 0. : the Fuzzy option is off
  // if aFuzzyValue > 0. : the Fuzzy option is on
  aBuilder.SetFuzzyValue(aFuzzyValue);
  //
  // safe mode - avoid modification of the arguments
  Standard_Boolean bSafeMode = Standard_True;
  // if bSafeMode == Standard_True - the safe mode is switched on
  // if bSafeMode == Standard_False - the safe mode is switched off
  aBuilder.SetNonDestructive(bSafeMode);
```
/**
// gluing options - for coinciding arguments
BOPAlgo_GlueEnum aGlueOpt = BOPAlgo_GlueFull;
// if aGlueOpt == BOPAlgo_GlueOff  - the gluing
// mode is switched off
// if aGlueOpt == BOPAlgo_GlueShift - the gluing
// mode is switched on
// if aGlueOpt == BOPAlgo_GlueFull  - the gluing
// mode is switched on
aBuilder.SetGlue(aGlueOpt);
//
// run the algorithm
aBuilder.Build();
if (aBuilder.HasErrors()) {
    // an error treatment
    return;
}
//
// result of the operation aR
const TopoDS_Shape& aR=aBuilder.Shape();
...}

Tcl Level

# prepare the arguments
box b1 10 10 10
box b2 3 4 5 10 10 10
box b3 5 6 7 10 10 10
#
# clear inner contents
bclearobjects; bcleartools;
#
# set the arguments
baddobjects b1 b3
baddtools b2
#
# set parallel processing mode
# 1: the parallel processing is switched on
# 0: the parallel processing is switched off
#runparallel 1
#
# set Fuzzy value
# 0. : the Fuzzy option is off
# >0. : the Fuzzy option is on
#fuzzyvalue 0.
#
# set safe processing mode
#nondestructive 1
# set safe mode
# 1 - the safe processing mode is switched on
# 0 - the safe processing mode is switched off
#
# set gluing mode
#glue 1
# set the gluing mode
# 1 or 2 - the gluing mode is switched on
# 0 - the gluing mode is switched off
#
# run the algorithm
# r is the result of the operation
# 4 means Section operation
#apibop r 4
Shape Healing

Table of Contents

- Overview
  - Introduction
  - Examples of use
  - Toolkit Structure
  - Querying the statuses
- Repair
  - Basic Shape Repair
  - Shape Correction.
    - Fixing sub-shapes
  - Repairing tools
    - General Workflow
    - Flags Management
    - Repairing tool for shapes
    - Repairing tool for solids
    - Repairing tool for shells
    - Repairing tool for faces
    - Repairing tool for wires
    - Repairing tool for edges
    - Repairing tool for the
wireframe of a shape

- Tool for removing small faces from a shape
- Tool to modify tolerances of shapes (Class ShapeFix_ShapeTolerance)

Analysis

- Analysis of shape validity
  - Analysis of orientation of wires on a face.
  - Analysis of wire validity
  - Analysis of edge validity
  - Analysis of presence of small faces
  - Analysis of shell validity and closure

- Analysis of shape properties.
  - Analysis of tolerance on shape
  - Analysis of free boundaries.
  - Analysis of shape contents

Upgrading

- Tools for splitting a shape according to a specified criterion

Overview

- Using tools available for shape splitting.
Creation of a new tool for splitting a shape.

General splitting tools.
- General tool for shape splitting
- General tool for face splitting
- General tool for wire splitting
- General tool for edge splitting
- General tools for geometry splitting

Specific splitting tools.
- Conversion of shape geometry to the target continuity
- Splitting by angle
- Conversion of 2D, 3D curves and surfaces to Bezier
- Tool for splitting closed faces
- Tool for splitting a C0 BSpline 2D or 3D curve to a sequence C1 BSpline curves
- Tool for splitting faces

Customization of shapes
- Conversion of indirect surfaces.
- Shape Scaling
- Conversion of curves and surfaces to BSpline
- Conversion of elementary surfaces into surfaces of revolution
- Conversion of elementary surfaces into Bspline surfaces
- Getting the history of modification of sub-shapes.
- Remove internal wires
- Conversion of surfaces
- Unify Same Domain

- Auxiliary tools for repairing, analysis and upgrading
  - Tool for rebuilding shapes
  - Status definition
  - Tool representing a wire
  - Tool for exploring shapes
  - Tool for attaching messages to objects
  - Tools for performance measurement

- Shape Processing
- Usage Workflow
- Operators
- Messaging mechanism
<table>
<thead>
<tr>
<th>Message Gravity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool for loading a message file into memory</td>
</tr>
<tr>
<td>Tool for managing filling messages</td>
</tr>
<tr>
<td>Tool for managing trace files</td>
</tr>
</tbody>
</table>
Overview

Introduction

This manual explains how to use Shape Healing. It provides basic documentation on its operation. For advanced information on Shape Healing and its applications, see our E-learning & Training offerings.

The **Shape Healing** toolkit provides a set of tools to work on the geometry and topology of Open CASCADE Technology (OCCT) shapes. Shape Healing adapts shapes so as to make them as appropriate for use by Open CASCADE Technology as possible.
Examples of use

Here are a few examples of typical problems with illustrations of how Shape Healing deals with them:

**Face with missing seam edge**

The problem: Face on a periodical surface is limited by wires which make a full trip around the surface. These wires are closed in 3d but not closed in parametric space of the surface. This is not valid in Open CASCADE. The solution: Shape Healing fixes this face by inserting seam edge which combines two open wires and thus closes the parametric space. Note that internal wires are processed correctly.

**Wrong orientation of wires**

The problem: Wires on face have incorrect orientation, so that interior and outer parts of the face are mixed. The solution: Shape Healing recovers correct orientation of wires.

**Self-intersecting wire**

The problem: Face is invalid because its boundary wire has self-intersection (on two adjacent edges) The solution: Shape Healing cuts intersecting edges at intersection points thus making boundary valid.

**Lacking edge**

The problem: There is a gap between two edges in the wire, so that wire is not closed The solution: Shape Healing closes a gap by inserting lacking edge.
**Toolkit Structure**

**Shape Healing** currently includes several packages that are designed to help you to:

- analyze shape characteristics and, in particular, identify shapes that do not comply with Open CASCADE Technology validity rules
- fix some of the problems shapes may have
- upgrade shape characteristics for users needs, for example a C0 supporting surface can be upgraded so that it becomes C1 continuous.

The following diagram shows dependencies of API packages:

![Shape Healing packages diagram](image)

Each sub-domain has its own scope of functionality:

- analysis – exploring shape properties, computing shape features, detecting violation of OCCT requirements (shape itself is not modified);
- fixing – fixing shape to meet the OCCT requirements (the shape may change its original form: modifying, removing, constructing sub-shapes, etc.);
- upgrade – shape improvement for better usability in Open
CASCADE Technology or other algorithms (the shape is replaced with a new one, but geometrically they are the same);

- customization – modifying shape representation to fit specific needs (shape is not modified, only the form of its representation is modified);
- processing – mechanism of managing shape modification via a user-editable resource file.

Message management is used for creating messages, filling them with various parameters and storing them in the trace file. This tool provides functionality for attaching messages to the shapes for deferred analysis of various run-time events. In this document only general principles of using Shape Healing will be described. For more detailed information please see the corresponding header files.

Tools responsible for analysis, fixing and upgrading of shapes can give the information about how these operations were performed. This information can be obtained by the user with the help of mechanism of status querying.
Querying the statuses

Each fixing and upgrading tool has its own status, which is reset when their methods are called. The status can contain several flags, which give the information about how the method was performed. For exploring the statuses, a set of methods named Status...() is provided. These methods accept enumeration ShapeExtend_Status and return True if the status has the corresponding flag set. The meaning of flags for each method is described below.

The status may contain a set of Boolean flags (internally represented by bits). Flags are coded by enumeration ShapeExtend_Status. This enumeration provides the following families of statuses:

- **ShapeExtend_OK** – The situation is OK, no operation is necessary and has not been performed.
- **ShapeExtend_DONE** – The operation has been successfully performed.
- **ShapeExtend_FAIL** – An error has occurred during operation.

It is possible to test the status for the presence of some flag(s), using Status...() method(s) provided by the class:

```java
if ( object.Status.. ( ShapeExtend_DONE ) ) { // something was done
}
```

8 'DONE' and 8 'FAIL' flags, named ShapeExtend_DONE1 ... ShapeExtend_FAIL8, are defined for a detailed analysis of the encountered situation. Each method assigns its own meaning to each flag, documented in the header for that method. There are also three enumerative values used for testing several flags at a time:

- **ShapeExtend_OK** – if no flags have been set;
- **ShapeExtend_DONE** – if at least one ShapeExtend_DONEi has been set;
- **ShapeExtend_FAIL** – if at least one ShapeExtend_FAILi has been set.
Repair

Algorithms for fixing problematic (violating the OCCT requirements) shapes are placed in package \textit{ShapeFix}.

Each class of package \textit{ShapeFix} deals with one certain type of shapes or with some family of problems.

There is no necessity for you to detect problems before using \textit{ShapeFix} because all components of package \textit{ShapeFix} make an analysis of existing problems before fixing them by a corresponding tool from package of \textit{ShapeAnalysis} and then fix the discovered problems.

The \textit{ShapeFix} package currently includes functions that:

- add a 2D curve or a 3D curve where one is missing,
- correct a deviation of a 2D curve from a 3D curve when it exceeds a given tolerance value,
- limit the tolerance value of shapes within a given range,
- set a given tolerance value for shapes,
- repair the connections between adjacent edges of a wire,
- correct self-intersecting wires,
- add seam edges,
- correct gaps between 3D and 2D curves,
- merge and remove small edges,
- correct orientation of shells and solids.
Basic Shape Repair

The simplest way for fixing shapes is to use classes \textit{ShapeFix\_Shape} and \textit{ShapeFix\_Wireframe} on a whole shape with default parameters. A combination of these tools can fix most of the problems that shapes may have. The sequence of actions is as follows:

1. Create tool \textit{ShapeFix\_Shape} and initialize it by shape:

   
   ```cpp
   Handle(ShapeFix\_Shape) sfs = new ShapeFix\_Shape;
   sfs->Init( shape );
   ```

2. Set the basic precision, the maximum allowed tolerance, the minimal allowed tolerance:

   ```cpp
   sfs->SetPrecision ( Prec );
   sfs->SetMaxTolerance ( maxTol );
   sfs->SetMinTolerance ( mintol );
   ```

   where:

   - \textit{Prec} – basic precision.
   - \textit{maxTol} – maximum allowed tolerance. All problems will be detected for cases when a dimension of invalidity is larger than the basic precision or a tolerance of sub-shape on that problem is detected. The maximum tolerance value limits the increasing tolerance for fixing a problem such as fix of not connected and self-intersected wires. If a value larger than the maximum allowed tolerance is necessary for correcting a detected problem the problem cannot be fixed. The maximal tolerance is not taking into account during computation of tolerance of edges in \textit{ShapeFix\_SameParameter()} method and \textit{ShapeFix\_Edge::FixVertexTolerance()} method. See \textit{Repairing tool for edges} for details.
   - \textit{minTol} – minimal allowed tolerance. It defines the minimal allowed length of edges. Detected edges having length less than the specified minimal tolerance will be removed if \textit{ModifyTopologyMode} in Repairing tool for wires is set to true. See \textit{Repairing tool for wires} for details.
3. Launch fixing:

```cpp
sfs->Perform();
```

4. Get the result:

```cpp
TopoDS_Shape aResult = sfs->Shape();
```

In some cases using only `ShapeFix_Shape` can be insufficient. It is possible to use tools for merging and removing small edges and fixing gaps between 2D and 3D curves.

5. Create `ShapeFix_Wireframe` tool and initialize it by shape:

```cpp
Handle(ShapeFix_Wireframe) SFWF = new ShapeFix_Wireframe(shape);
Or
Handle(ShapeFix_Wireframe) SFWF = new ShapeFix_Wireframe;
SFWF->Load(shape);
```

6. Set the basic precision and the maximum allowed tolerance:

```cpp
sfs->SetPrecision (Prec);
sfs->SetMaxTolerance (maxTol);
```

See the description for `Prec` and `maxTol` above.

7. Merge and remove small edges:

```cpp
SFWF->DropSmallEdgesMode() = Standard_True;
SFWF->FixSmallEdges();
```

**Note:** Small edges are not removed with the default mode, but in many cases removing small edges is very useful for fixing a shape.

8. Fix gaps for 2D and 3D curves

```cpp
SFWF->FixWireGaps();
```

9. Get the result

```cpp
TopoDS_Shape Result = SFWF->Shape();
```
Shape Correction.

If you do not want to make fixes on the whole shape or make a definite set of fixes you can set flags for separate fix cases (marking them ON or OFF) and you can also use classes for fixing specific types of sub-shapes such as solids, shells, faces, wires, etc.

For each type of sub-shapes there are specific types of fixing tools such as *ShapeFix_Solid, ShapeFix_Shell, ShapeFix_Face, ShapeFix_Wire*, etc.

Fixing sub-shapes

If you want to make a fix on one sub-shape of a certain shape it is possible to take the following steps:

- create a tool for a specified sub-shape type and initialize this tool by the sub-shape;
- create a tool for rebuilding the shape and initialize it by the whole shape (section 5.1);
- set a tool for rebuilding the shape in the tool for fixing the sub-shape;
- fix the sub-shape;
- get the resulting whole shape containing a new corrected sub-shape.

For example, in the following way it is possible to fix face *Face1* of shape *Shape1*:

```cpp
//create tools for fixing a face
Handle(ShapeFix_Face) SFF = new ShapeFix_Face;

// create tool for rebuilding a shape and initialize it by shape
Handle(ShapeBuild_ReShape) Context = new ShapeBuild_ReShape;
Context->Apply(Shape1);

//set a tool for rebuilding a shape in the tool for fixing
```

```cpp```
```cpp
SFF->SetContext(Context);

//initialize the fixing tool by one face
SFF->Init(Face1);

//fix the set face
SFF->Perform();

//get the result
TopoDS_Shape NewShape = Context->Apply(Shape1);
//Resulting shape contains the fixed face.

A set of required fixes and invalid sub-shapes can be obtained with the help of tools responsible for the analysis of shape validity (section 3.2).
Repairing tools

Each class of package ShapeFix deals with one certain type of shapes or with a family of problems. Each repairing tool makes fixes for the specified shape and its sub-shapes with the help of method Perform() containing an optimal set of fixes. The execution of these fixes in the method Perform can be managed with help of a set of control flags (fixes can be either forced or forbidden).

General Workflow

The following sequence of actions should be applied to perform fixes:

1. Create a tool.
2. Set the following values:
   - the working precision by method SetPrecision() (default 1.e-7)
   - set the maximum allowed tolerance by method SetMaxTolerance() (by default it is equal to the working precision).
   - set the minimum tolerance by method SetMinTolerance() (by default it is equal to the working precision).
   - set a tool for rebuilding shapes after the modification (tool ShapeBuild_ReShape) by method SetContext(). For separate faces, wires and edges this tool is set optionally.
   - to force or forbid some of fixes, set the corresponding flag to 0 or 1.
3. Initialize the tool by the shape with the help of methods Init or Load.
4. Use method Perform() or create a custom set of fixes.
5. Check the statuses of fixes by the general method Status or specialized methods Status_(for example StatusSelfIntersection (ShapeExtentd_DONE)). See the description of statuses below.
6. Get the result in two ways:
   - with help of a special method Shape(),Face(),Wire().Edge().
   - from the rebuilding tool by method Apply (for access to rebuilding tool use method Context()):

        TopoDS_Shape resultShape = fixtool->Context() -
        >Apply(initialShape);

Modification history for the shape and its sub-shapes can be
obtained from the tool for shape re-building \((\text{ShapeBuild\_ReShape})\).

\[
\text{TopoDS\_Shape \ modifsubshape = fixtool->Context() -> Apply(initsubshape);}\
\]

**Flags Management**

The flags \textit{Fix...Mode()} are used to control the execution of fixing procedures from the API fixing methods. By default, these flags have values equal to -1, this means that the corresponding procedure will either be called or not called, depending on the situation. If the flag is set to 1, the procedure is executed anyway; if the flag is 0, the procedure is not executed. The name of the flag corresponds to the fixing procedure that is controlled. For each fixing tool there exists its own set of flags. To set a flag to the desired value, get a tool containing this flag and set the flag to the required value.

For example, it is possible to forbid performing fixes to remove small edges - \textit{FixSmall}

\[
\begin{align*}
\text{Handle(ShapeFix\_Shape) Sfs = new ShapeFix\_Shape(shape);} \\
\text{Sfs-> FixWireTool ()-FixSmallMode () =0;}
\end{align*}
\]

\[
\text{if(Sfs->Perform())}
\]

\[
\begin{align*}
\text{TopoDS\_Shape resShape = Sfs->Shape();}
\end{align*}
\]

**Repairing tool for shapes**

Class \textit{ShapeFix\_Shape} allows using repairing tools for all sub-shapes of a shape. It provides access to all repairing tools for fixing sub-shapes of the specified shape and to all control flags from these tools.

For example, it is possible to force the removal of invalid 2D curves from a face.

\[
\text{TopoDS\_Face face ... // face with invalid 2D curves.}
\]

\[
\text{//creation of tool and its initialization by shape.}
\]

\[
\text{Handle(ShapeFix\_Shape) sfs = new}
\]
ShapeFix_Shape(face);
// set work precision and max allowed tolerance.
sfs->SetPrecision(prec);
sfs->SetMaxTolerance(maxTol);
// set the value of flag for forcing the removal of 2D curves
sfs->FixWireTool()->FixRemovePCurveMode() = 1;
// reform fixes
sfs->Perform();
// getting the result
if(sfs->Status(ShapeExtend_DONE)) {
    cout << "Shape was fixed" << endl;
    TopoDS_Shape resFace = sfs->Shape();
}
else if(sfs->Status(ShapeExtend_FAIL)) {
    cout << "Shape could not be fixed" << endl;
}
else if(sfs->Status(ShapeExtent_OK)) {
    cout << "Initial face is valid with specified precision =" << prec << endl;
}

Repairing tool for solids

Class ShapeFix_Solid allows fixing solids and building a solid from a shell to obtain a valid solid with a finite volume. The tool ShapeFix_Shell is used for correction of shells belonging to a solid.

This tool has the following control flags:

- **FixShellMode** – Mode for applying fixes of ShapeFix_Shell, True by default.
- **CreateOpenShellMode** – If it is equal to true solids are created from open shells, else solids are created from closed shells only, False by default.

Repairing tool for shells
Class *ShapeFix_Shell* allows fixing wrong orientation of faces in a shell. It changes the orientation of faces in the shell so that all faces in the shell have coherent orientations. If it is impossible to orient all faces in the shell (like in case of Möbius tape), then a few manifold or non-manifold shells will be created depending on the specified Non-manifold mode. The *ShapeFix_Face* tool is used to correct faces in the shell. This tool has the following control flags:

- *FixFaceMode* – mode for applying the fixes of *ShapeFix_Face*, *True* by default.
- *FixOrientationMode* – mode for applying a fix for the orientation of faces in the shell.

**Repairing tool for faces**

Class *ShapeFix_Face* allows fixing the problems connected with wires of a face. It allows controlling the creation of a face (adding wires), and fixing wires by means of tool *ShapeFix_Wire*. When a wire is added to a face, it can be reordered and degenerated edges can be fixed. This is performed or not depending on the user-defined flags (by default, *False*). The following fixes are available:

- Fixing of wires orientation on the face. If the face has no wire, the natural bounds are computed. If the face is on a spherical surface and has two or more wires on it describing holes, the natural bounds are added. In case of a single wire, it is made to be an outer one. If the face has several wires, they are oriented to lay one outside another (if possible). If the supporting surface is periodic, 2D curves of internal wires can be shifted on integer number of periods to put them inside the outer wire.
- Fixing the case when the face on the closed surface is defined by a set of closed wires, and the seam is missing (this is not valid in OCCT). In that case, these wires are connected by means of seam edges into the same wire.

This tool has the following control flags:

- *FixOrientationMode* – mode for orienting a wire to border a limited square, *True* by default.
- *FixAddNaturalBoundMode* – mode for adding natural bounds to a
face, False by default.

- **FixMissingSeamMode** – mode to fix a missing seam, True by default. If True, tries to insert a seam.
- **FixSmallAreaWireMode** – mode to fix a small-area wire, False by default. If True, drops wires bounding small areas.

```cpp
TopoDS_Face face = ...;
TopoDS_Wire wire = ...;

// Creates a tool and adds a wire to the face
ShapeFix_Face sff (face);
sff.Add (wire);

// Use method Perform to fix the wire and the face
sff.Perform();

// Or make a separate fix for the orientation of wire
// on the face
sff.FixOrientation();

// Get the resulting face
TopoDS_Face newface = sff.Face();
```

**Repairing tool for wires**

Class *ShapeFix_Wire* allows fixing a wire. Its method *Perform()* performs all the available fixes in addition to the geometrical filling of gaps. The geometrical filling of gaps can be made with the help of the tool for fixing the wireframe of shape *ShapeFix_Wireframe*.

The fixing order and the default behavior of *Perform()* is as follows:

- Edges in the wire are reordered by *FixReorder*. Most of fixing methods expect edges in a wire to be ordered, so it is necessary to make call to *FixReorder()* before making any other fixes. Even if it is forbidden, the analysis of whether the wire is ordered or not is performed anyway.
- Small edges are removed by *FixSmall*.
- Edges in the wire are connected (topologically) by *FixConnected* (if
the wire is ordered).
- Edges (3D curves and 2D curves) are fixed by \textit{FixEdgeCurves} (without \textit{FixShifted} if the wire is not ordered).
- Degenerated edges are added by \textit{FixDegenerated} (if the wire is ordered).
- Self-intersection is fixed by \textit{FixSelfIntersection} (if the wire is ordered and \textit{ClosedMode} is True).
- Lacking edges are fixed by \textit{FixLacking} (if the wire is ordered).

The flag \textit{ClosedWireMode} specifies whether the wire is (or should be) closed or not. If that flag is True (by default), fixes that require or force connection between edges are also executed for the last and the first edges.

The fixing methods can be turned on/off by using their corresponding control flags:

- \textit{FixReorderMode},
- \textit{FixSmallMode},
- \textit{FixConnectedMode},
- \textit{FixEdgeCurvesMode},
- \textit{FixDegeneratedMode},
- \textit{FixSelfIntersectionMode}

Some fixes can be made in three ways:

- Increasing the tolerance of an edge or a vertex.
- Changing topology (adding/removing/replacing an edge in the wire and/or replacing the vertex in the edge, copying the edge etc.).
- Changing geometry (shifting a vertex or adjusting ends of an edge curve to vertices, or recomputing a 3D curve or 2D curves of the edge).

When it is possible to make a fix in more than one way (e.g., either by increasing the tolerance or shifting a vertex), it is chosen according to the user-defined flags:

- \textit{ModifyTopologyMode} – allows modifying topology, False by default.
- \textit{ModifyGeometryMode} – allows modifying geometry. Now this flag is used only in fixing self-intersecting edges (allows to modify 2D curves) and is True by default.
Fixing disordered edges

*FixReorder* is necessary for most other fixes (but is not necessary for Open CASCADE Technology). It checks whether edges in the wire go in a sequential order (the end of a preceding edge is the start of a following one). If it is not so, an attempt to reorder the edges is made.

Fixing small edges

*FixSmall* method searches for the edges, which have a length less than the given value (degenerated edges are ignored). If such an edge is found, it is removed provided that one of the following conditions is satisfied:

- both end vertices of that edge are one and the same vertex,
- end vertices of the edge are different, but the flag *ModifyTopologyMode* is True. In the latter case, method *FixConnected* is applied to the preceding and the following edges to ensure their connection.

Fixing disconnected edges

*FixConnected* method forces two adjacent edges to share the same common vertex (if they do not have a common one). It checks whether the end vertex of the preceding edge coincides with the start vertex of the following edge with the given precision, and then creates a new vertex and sets it as a common vertex for the fixed edges. At that point, edges are copied, hence the wire topology is changed (regardless of the *ModifyTopologyMode* flag). If the vertices do not coincide, this method fails.

Fixing the consistency of edge curves

*FixEdgeCurves* method performs a set of fixes dealing with 3D curves and 2D curves of edges in a wire.

These fixes will be activated with the help of a set of fixes from the repairing tool for edges called *ShapeFix_Edge*. Each of these fixes can be forced or forbidden by means of setting the corresponding flag to
either True or False.

The mentioned fixes and the conditions of their execution are:

- fixing a disoriented 2D curve by call to `ShapeFix_Edge::FixReversed2d` – if not forbidden by flag `FixReversed2dMode`;
- removing a wrong 2D curve by call to `ShapeFix_Edge::FixRemovePCurve` – only if forced by flag `FixRemovePCurveMode`;
- fixing a missing 2D curve by call to `ShapeFix_Edge::FixAddPCurve` – if not forbidden by flag `FixAddPCurveMode`;
- removing a wrong 3D curve by call to `ShapeFix_Edge::FixRemoveCurve3d` – only if forced by flag `FixRemoveCurve3dMode`;
- fixing a missing 3D curve by call to `ShapeFix_Edge::FixAddCurve3d` – if not forbidden by flag `FixAddCurve3dMode`;
- fixing 2D curves of seam edges – if not forbidden by flag `FixSeamMode`;
- fixing 2D curves which can be shifted at an integer number of periods on the closed surface by call to `ShapeFix_Edge::FixShifted` – if not forbidden by flag `FixShiftedMode`.

This fix is required if 2D curves of some edges in a wire lying on a closed surface were recomputed from 3D curves. In that case, the 2D curve for the edge, which goes along the seam of the surface, can be incorrectly shifted at an integer number of periods. The method `FixShifted` detects such cases and shifts wrong 2D curves back, ensuring that the 2D curves of the edges in the wire are connected.

- fixing the SameParameter problem by call to `ShapeFix_Edge::FixSameParameter` – if not forbidden by flag `FixSameParameterMode`.

**Fixing degenerated edges**

`FixDegenerated` method checks whether an edge in a wire lies on a degenerated point of the supporting surface, or whether there is a degenerated point between the edges. If one of these cases is detected for any edge, a new degenerated edge is created and it replaces the
current edge in the first case or is added to the wire in the second case. The newly created degenerated edge has a straight 2D curve, which goes from the end of the 2D curve of the preceding edge to the start of the following one.

Fixing intersections of 2D curves of the edges

*FixSelfIntersection* method detects and fixes the following problems:

- self-intersection of 2D curves of individual edges. If the flag `ModifyGeometryMode()` is False this fix will be performed by increasing the tolerance of one of end vertices to a value less then `MaxTolerance()`.
- intersection of 2D curves of each of the two adjacent edges (except the first and the last edges if the flag `ClosedWireMode` is False). If such intersection is found, the common vertex is modified in order to comprise the intersection point. If the flag `ModifyTopologyMode` is False this fix will be performed by increasing the tolerance of the vertex to a value less then `MaxTolerance()`.
- intersection of 2D curves of non-adjacent edges. If such intersection is found the tolerance of the nearest vertex is increased to comprise the intersection point. If such increase cannot be done with a tolerance less than `MaxTolerance` this fix will not be performed.

Fixing a lacking edge

*FixLacking* method checks whether a wire is not closed in the parametric space of the surface (while it can be closed in 3D). This is done by checking whether the gap between 2D curves of each of the two adjacent edges in the wire is smaller than the tolerance of the corresponding vertex. The algorithm computes the gap between the edges, analyses positional relationship of the ends of these edges and (if possible) tries to insert a new edge into the gap or increases the tolerance.

Fixing gaps in 2D and 3D wire by geometrical filling

The following methods check gaps between the ends of 2D or 3D curves of adjacent edges:
- Method *FixGap2d* moves the ends of 2D curves to the middle point.
- Method *FixGaps3d* moves the ends of 3D curves to a common vertex.

Boolean flag *FixGapsByRanges* is used to activate an additional mode applied before converting to B-Splines. When this mode is on, methods try to find the most precise intersection of curves, or the most precise projection of a target point, or an extremity point between two curves (to modify their parametric range accordingly). This mode is off by default. Independently of the additional mode described above, if gaps remain, these methods convert curves to B-Spline form and shift their ends if a gap is detected.

**Example: A custom set of fixes**

Let us create a custom set of fixes as an example.

```cpp
TopoDS_Face face = ...;
TopoDS_Wire wire = ...;
Standard_Real precision = 1e-04;
ShapeFix_Wire sfw (wire, face, precision);
//Creates a tool and loads objects into it
sfw.FixReorder();
//Orders edges in the wire so that each edge starts
     at the end of the one before it.
sfw.FixConnected();
//Forces all adjacent edges to share
  //the same vertex
Standard_Boolean LockVertex = Standard_True;
    if (sfw.FixSmall (LockVertex, precision)) {
       //Removes all edges which are shorter than the
          given precision and have the same vertex at both
          ends.
    }
    if (sfw.FixSelfIntersection()) {
       //Fixes self-intersecting edges and intersecting
          adjacent edges.
       cout <<"Wire was slightly self-intersecting.";
```
Repaired"<<endl;
}
if ( sfw.FixLacking ( Standard_False ) ) {
    // Inserts edges to connect adjacent non-continuous edges.
}
TopoDS_Wire newwire = sfw.Wire();
// Returns the corrected wire

**Example: Correction of a wire**

Let us correct the following wire:

![Initial shape](image)

It is necessary to apply the **tools for the analysis of wire validity** to check that:

- the edges are correctly oriented;
- there are no edges that are too short;
- there are no intersecting adjacent edges; and then immediately apply fixing tools.
TopoDS_Face face = ...;
TopoDS_Wire wire = ...;
Standard_Real precision = 1e-04;
ShapeAnalysis_Wire saw (wire, face, precision);
ShapeFix_Wire sfw (wire, face, precision);
if (saw.CheckOrder()) {
    cout << “Some edges in the wire need to be reordered” << endl;
    // Two edges are incorrectly oriented
    sfw.FixReorder();
    cout << “Reordering is done” << endl;
}
// their orientation is corrected
if (saw.CheckSmall (precision)) {
    cout << “Wire contains edge(s) shorter than “ << precision << endl;
    // An edge that is shorter than the given tolerance is found.
    Standard_Boolean LockVertex = Standard_True;
    if (sfw.FixSmall (LockVertex, precision)) {
        cout << “Edges shorter than “ << precision << “ have been removed” << endl;
        // The edge is removed
    }
}
if (saw.CheckSelfIntersection()) {
    cout << “Wire has self-intersecting or intersecting adjacent edges” << endl;
    // Two intersecting adjacent edges are found.
    if (sfw.FixSelfIntersection()) {
        cout << “Wire was slightly self-intersecting. Repaired” << endl;
        // The edges are cut at the intersection point so that they no longer intersect.
    }
}
As the result all failures have been fixed.

Resulting shape

Repairing tool for edges

Class ShapeFix_Edge provides tools for fixing invalid edges. The following geometrical and/or topological inconsistencies are detected and fixed:

- missing 3D curve or 2D curve,
- mismatching orientation of a 3D curve and a 2D curve,
- incorrect SameParameter flag (curve deviation is greater than the edge tolerance). Each fixing method first checks whether the problem exists using methods of the ShapeAnalysis_Edge class. If the problem is not detected, nothing is done. This tool does not have the method Perform().

To see how this tool works, it is possible to take an edge, where the maximum deviation between the 3D curve and 2D curve P1 is greater than the edge tolerance.
Initial shape

First it is necessary to apply the tool for checking the edge validity to find that the maximum deviation between pcurve and 3D curve is greater than tolerance. Then we can use the repairing tool to increase the tolerance and make the deviation acceptable.

ShapeAnalysis_Edge sae;
TopoDS_Face face = ...;
TopoDS_Wire wire = ...;
Standard_Real precision = 1e-04;
ShapeFix_Edge sfe;
Standard_Real maxdev;
if (sae.CheckSameParameter (edge, maxdev)) {
    cout<<“Incorrect SameParameter flag”<<endl;
    cout<<“Maximum deviation “<<maxdev<< “, tolerance “<<BRep_Tool::Tolerance(edge)<<endl;
    sfe.FixSameParameter();
    cout<<“New tolerance “<<BRep_Tool::Tolerance(edge)<<endl;
}
As the result, the edge tolerance has been increased.

**Repairing tool for the wireframe of a shape**

Class *ShapeFix_Wireframe* provides methods for geometrical fixing of gaps and merging small edges in a shape. This class performs the following operations:

- fills gaps in the 2D and 3D wireframe of a shape.
- merges and removes small edges.

Fixing of small edges can be managed with the help of two flags:

- *ModeDropSmallEdges()* – mode for removing small edges that cannot be merged, by default it is equal to Standard_False.
- *LimitAngle* – maximum possible angle for merging two adjacent edges, by default no limit angle is applied (-1). To perform fixes it is necessary to:
  - create a tool and initialize it by shape,
  - set the working precision problems will be detected with and the maximum allowed tolerance
  - perform fixes
//creation of a tool
Handle(ShapeFix_Wireframe) sfwf = new
    ShapeFix_Wireframe(shape);
//sets the working precision problems will be
    detected with and the maximum allowed tolerance
sfwf->SetPrecision(prec);
sfwf->SetMaxTolerance(maxTol);
//fixing of gaps
sfwf->FixWireGaps();
//fixing of small edges
//setting of the drop mode for the fixing of small
    edges and max possible angle between merged edges.
sfwf->ModeDropSmallEdges = Standard_True;
sfwf->SetLimitAngle(angle);
//performing the fix
sfwf->FixSmallEdges();
//getting the result
TopoDS_Shape resShape = sfwf->Shape();

It is desirable that a shape is topologically correct before applying the
methods of this class.

**Tool for removing small faces from a shape**

Class ShapeFix_FixSmallFaceThis tool is intended for dropping small
faces from the shape. The following cases are processed:

- Spot face: if the size of the face is less than the given precision;
- Strip face: if the size of the face in one dimension is less than the
given precision.

The sequence of actions for performing the fix is the same as for the fixes
described above:

//creation of a tool
Handle(ShapeFix_FixSmallFace) sff = new
    ShapeFix_FixSmallFace(shape);
Tool to modify tolerances of shapes (Class ShapeFix_ShapeTolerance).

This tool provides a functionality to set tolerances of a shape and its sub-shapes. In Open CASCADE Technology only vertices, edges and faces have tolerances.

This tool allows processing each concrete type of sub-shapes or all types at a time. You set the tolerance functionality as follows:

- set a tolerance for sub-shapes, by method SetTolerance,
- limit tolerances with given ranges, by method LimitTolerance.

//creation of a tool
ShapeFix_ShapeTolerance Sft;
//setting a specified tolerance on shape and all of its sub-shapes.
Sft.SetTolerance(shape,toler);
//setting a specified tolerance for vertices only
Sft.SetTolerance(shape,toler,TopAbs_VERTEX);
//limiting the tolerance on the shape and its sub-shapes between minimum and maximum tolerances
Sft.LimitTolerance(shape,tolermiuntolermax);
Analysis

Analysis of shape validity

The *ShapeAnalysis* package provides tools for the analysis of topological shapes. It is not necessary to check a shape by these tools before the execution of repairing tools because these tools are used for the analysis before performing fixes inside the repairing tools. However, if you want, these tools can be used for detecting some of shape problems independently from the repairing tools.

It can be done in the following way:

- create an analysis tool.
- initialize it by shape and set a tolerance problems will be detected with if it is necessary.
- check the problem that interests you.

```cpp
TopoDS_Face face = ...;
ShapeAnalysis_Edge sae;
//Creates a tool for analyzing an edge
for(TopExp_Explorer Exp(face,TopAbs_EDGE);Exp.More();Exp.Next()) {
    TopoDS_Edge edge = TopoDS::Edge (Exp.Current());
    if (!sae.HasCurve3d (edge)) {
        cout <<"Edge has no 3D curve"<< endl;  
    }
}
```

Analysis of orientation of wires on a face.

It is possible to check whether a face has an outer boundary with the help of method *ShapeAnalysis::IsOuterBound*.

```cpp
TopoDS_Face face ... //analyzed face
if(!ShapeAnalysis::IsOuterBound(face)) {
    cout<<"Face has not outer boundary"<<endl;
```
Analysis of wire validity

Class `ShapeAnalysis_Wire` is intended to analyze a wire. It provides functionalities both to explore wire properties and to check its conformance to Open CASCADE Technology requirements. These functionalities include:

- checking the order of edges in the wire,
- checking for the presence of small edges (with a length less than the given value),
- checking for the presence of disconnected edges (adjacent edges having different vertices),
- checking the consistency of edge curves,
- checking for the presence or missing of degenerated edges,
- checking for the presence of self-intersecting edges and intersecting edges (edges intersection is understood as intersection of their 2D curves),
- checking for lacking edges to fill gaps in the surface parametric space,
- analyzing the wire orientation (to define the outer or the inner bound on the face),
- analyzing the orientation of the shape (edge or wire) being added to an already existing wire.

**Note** that all checking operations except for the first one are based on the assumption that edges in the wire are ordered. Thus, if the wire is detected as non-ordered it is necessary to order it before calling other checking operations. This can be done, for example, with the help of the `ShapeFix_Wire::FixOrder()` method.

This tool should be initialized with wire, face (or a surface with a location) or precision. Once the tool has been initialized, it is possible to perform the necessary checking operations. In order to obtain all information on a wire at a time the global method `Perform` is provided. It calls all other API checking operations to check each separate case.

API methods check for corresponding cases only, the value and the status they return can be analyzed to understand whether the case was
detected or not.

Some methods in this class are:

- **CheckOrder** checks whether edges in the wire are in the right order
- **CheckConnected** checks whether edges are disconnected
- **CheckSmall** checks whether there are edges that are shorter than the given value
- **CheckSelfIntersection** checks whether there are self-intersecting or adjacent intersecting edges. If the intersection takes place due to nonadjacent edges, it is not detected.

This class maintains status management. Each API method stores the status of its last execution which can be queried by the corresponding `Status..()` method. In addition, each API method returns a Boolean value, which is True when a case being analyzed is detected (with the set `ShapeExtend_DONE` status), otherwise it is False.

```plaintext
topoDS_Face face = ...;
topoDS_Wire wire = ...;
Standard_Real precision = 1e-04;
ShapeAnalysis_Wire saw (wire, face, precision);
//Creates a tool and loads objects into it
if (saw.CheckOrder()) {
    cout<<"Some edges in the wire need to be reordered"<<endl;
    cout<<"Please ensure that all the edges are correctly ordered before further analysis"<<endl;
    return;
}
if (saw.CheckSmall (precision)) {
    cout<<"Wire contains edge(s) shorter than "<<precision<<endl;
}
if (saw.CheckConnected()) {
    cout<<"Wire is disconnected"<<endl;
}
if (saw.CheckSelfIntersection()) {
```
cout<<"Wire has self-intersecting or intersecting adjacent edges"<< endl;
}

Analysis of edge validity

Class *ShapeAnalysis_Edge* is intended to analyze edges. It provides the following functionalities to work with an edge:

- querying geometrical representations (3D curve and pcurve(s) on a given face or surface),
- querying topological sub-shapes (bounding vertices),
- checking overlapping edges,
- analyzing the curves consistency:
  - mutual orientation of the 3D curve and 2D curve (co-directions or opposite directions),
  - correspondence of 3D and 2D curves to vertices.

This class supports status management described above.

```
TopoDS_Face face = ...;
ShapeAnalysis_Edge sae;
//Creates a tool for analyzing an edge
for(TopExp_Explorer
    Exp(face,TopAbs_EDGE);Exp.More();Exp.Next()) {
    TopoDS_Edge edge = TopoDS::Edge (Exp.Current());
    if (!sae.HasCurve3d (edge)) {
        cout << "Edge has no 3D curve" << endl;
    }
    Handle(Geom2d_Curve) pcurve;
    Standard_Real cf, cl;
    if (sae.PCurve (edge, face, pcurve, cf, cl, Standard_False)) {
        //Returns the pcurve and its range on the given face
        cout<<"Pcurve range ["<<cf<<", "<<cl<<"]"<< endl;
    }
    Standard_Real maxdev;
```
if (sae.CheckSameParameter (edge, maxdev)) {
    //Checks the consistency of all the curves in the edge
    cout<<"Incorrect SameParameter flag"<<endl;
}
cout<<"Maximum deviation "<<maxdev<<", tolerance" <<BRep_Tool::Tolerance(edge)<<endl;

//checks the overlapping of two edges
if(sae.CheckOverlapping(edge1,edge2,prec,dist)) {
    cout<<"Edges are overlapped with tolerance = " <<prec<<endl;
    cout<<"Domain of overlapping ="<<dist<<endl;
}

Analysis of presence of small faces

Class ShapeAnalysis_CheckSmallFace class is intended for analyzing small faces from the shape using the following methods:

- **CheckSpotFace()** checks if the size of the face is less than the given precision;
- **CheckStripFace** checks if the size of the face in one dimension is less than the given precision.

```cpp
TopoDS_Shape shape ... // checked shape
//Creation of a tool
ShapeAnalysis_CheckSmallFace saf;
//exploring the shape on faces and checking each face
Standard_Integer numSmallfaces =0;
for(TopExp_Explorer aExp(shape,TopAbs_FACE);
    aExp.More(); aExp.Next()) {
    TopoDS_Face face = TopoDS::Face(aexp.Current());
    TopoDS_Edge E1,E2;
    if(saf.CheckSpotFace(face,prec) ||
    saf.CheckStripFace(face,E1,E2,prec))
        NumSmallfaces++;
}
if(numSmallfaces)
    cout<<"Number of small faces in the shape ="<<
        numSmallfaces <<endl;

Analysis of shell validity and closure

Class \textit{ShapeAnalysis\_Shell} allows checking the orientation of edges in a manifold shell. With the help of this tool, free edges (edges entered into one face) and bad edges (edges entered into the shell twice with the same orientation) can be found. By occurrence of bad and free edges a conclusion about the shell validity and the closure of the shell can be made.

\begin{verbatim}
TopoDS_Shell shell // checked shape
ShapeAnalysis_Shell sas(shell);
//analysis of the shell , second parameter is set to
    True for //getting free edges,(default False)
sas.CheckOrientedShells(shell,Standard_True);
//getting the result of analysis
if(sas.HasBadEdges()) {
    cout<<"Shell is invalid"<<endl;
    TopoDS_Compound badEdges = sas.BadEdges();
}
if(sas.HasFreeEdges()) {
    cout<<"Shell is open"<<endl;
    TopoDS_Compound freeEdges = sas.FreeEdges();
}
\end{verbatim}
Analysis of shape properties.

Analysis of tolerance on shape

Class *ShapeAnalysis\_ShapeTolerance* allows computing tolerances of the shape and its sub-shapes. In Open CASCADE Technology only vertices, edges and faces have tolerances:

This tool allows analyzing each concrete type of sub-shapes or all types at a time. The analysis of tolerance functionality is the following:

- computing the minimum, maximum and average tolerances of sub-shapes,
- finding sub-shapes with tolerances exceeding the given value,
- finding sub-shapes with tolerances in the given range.

```cpp
TopoDS\_Shape shape = ...;
ShapeAnalysis\_ShapeTolerance sast;
Standard\_Real AverageOnShape = sast.Tolerance (shape, 0);
cout<<"Average tolerance of the shape is " <<AverageOnShape<<endl;
Standard\_Real MinOnEdge = sast.Tolerance (shape,-1,TopAbs\_EDGE);
cout<<"Minimum tolerance of the edges is " <<MinOnEdge<<endl;
Standard\_Real MaxOnVertex = sast.Tolerance (shape,1,TopAbs\_VERTEX);
cout<<"Maximum tolerance of the vertices is " <<MaxOnVertex<<endl;
Standard\_Real MaxAllowed = 0.1;
if (MaxOnVertex > MaxAllowed) {
    cout<<"Maximum tolerance of the vertices exceeds maximum allowed"<<endl;
}
```

Analysis of free boundaries.
Class ShapeAnalysis_FreeBounds is intended to analyze and output the free bounds of a shape. Free bounds are wires consisting of edges referenced only once by only one face in the shape. This class works on two distinct types of shapes when analyzing their free bounds:

- **Analysis of possible free bounds taking the specified tolerance into account.** This analysis can be applied to a compound of faces. The analyzer of the sewing algorithm (*BRepAlgo_Sewing*) is used to forecast what free bounds would be obtained after the sewing of these faces is performed. The following method should be used for this analysis:

```cpp
ShapeAnalysis_FreeBounds safb(shape, toler);
```

- **Analysis of already existing free bounds.** Actual free bounds (edges shared by the only face in the shell) are output in this case. *ShapeAnalysis_Shell* is used for that.

```cpp
ShapeAnalysis_FreeBounds safb(shape);
```

When connecting edges into wires this algorithm tries to build wires of maximum length. Two options are provided for the user to extract closed sub-contours out of closed and/or open contours. Free bounds are returned as two compounds, one for closed and one for open wires. To obtain a result it is necessary to use methods:

```cpp
TopoDS_Compound ClosedWires = safb.GetClosedWires();
TopoDS_Compound OpenWires = safb.GetOpenWires();
```

This class also provides some static methods for advanced use: connecting edges/wires to wires, extracting closed sub-wires from wires, distributing wires into compounds for closed and open wires.

```cpp
TopoDS_Shape shape = ...;
Standard_Real SewTolerance = 1.e-03;
// Tolerance for sewing
Standard_Boolean SplitClosed = Standard_False;
Standard_Boolean SplitOpen = Standard_True;
// in case of analysis of possible free boundaries
ShapeAnalysis_FreeBounds safb (shape, SewTolerance, SplitClosed, SplitOpen);
// in case of analysis of existing free bounds
ShapeAnalysis_FreeBounds safb (shape, SplitClosed, SplitOpen);
// getting the results
TopoDS_Compound ClosedWires = safb.GetClosedWires();
// Returns a compound of closed free bounds
TopoDS_Compound OpenWires = safb.GetClosedWires();
// Returns a compound of open free bounds

**Analysis of shape contents**

Class *ShapeAnalysis_ShapeContents* provides tools counting the number of sub-shapes and selecting a sub-shape by the following criteria:

Methods for getting the number of sub-shapes:

- number of solids,
- number of shells,
- number of faces,
- number of edges,
- number of vertices.

Methods for calculating the number of geometrical objects or sub-shapes with a specified type:

- number of free faces,
- number of free wires,
- number of free edges,
- number of C0 surfaces,
- number of C0 curves,
- number of BSpline surfaces,… etc

and selecting sub-shapes by various criteria.

The corresponding flags should be set to True for storing a shape by a specified criteria:

- faces based on indirect surfaces – safc.ModifyIndirectMode() = *Standard_True*;
- faces based on offset surfaces – safc.ModifyOffsetSurfaceMode() =
**Standard_True;**

- edges if their 3D curves are trimmed – `sa.fc.ModifyTrimmed3dMode()` = **Standard_True**;
- edges if their 3D curves and 2D curves are offset curves – `sa.fc.ModifyOffsetCurveMode()` = **Standard_True**;
- edges if their 2D curves are trimmed – `sa.fc.ModifyTrimmed2dMode()` = **Standard_True**;

Let us, for example, select faces based on offset surfaces.

```cpp
ShapeAnalysis_ShapeContents safc;
//set a corresponding flag for storing faces based on the offset surfaces
sa.fc.ModifyOffsetSurfaceMode() = Standard_True;
safc.Perform(shape);
//getting the number of offset surfaces in the shape
Standard_Integer NbOffsetSurfaces = safc.NbOffsetSurf();
//getting the sequence of faces based on offset surfaces.
Handle(TopTools_HSequenceOfShape) seqFaces = safc.OffsetSurfaceSec();
```
Upgrading

Upgrading tools are intended for adaptation of shapes for better use by Open CASCADE Technology or for customization to particular needs, i.e. for export to another system. This means that not only it corrects and upgrades but also changes the definition of a shape with regard to its geometry, size and other aspects. Convenient API allows you to create your own tools to perform specific upgrading. Additional tools for particular cases provide an ability to divide shapes and surfaces according to certain criteria.
Tools for splitting a shape according to a specified criterion

Overview

These tools provide such modifications when one topological object can be divided or converted to several ones according to specified criteria. Besides, there are high level API tools for particular cases which:

- Convert the geometry of shapes up to a given continuity,
- split revolutions by U to segments less than the given value,
- convert to Bezier surfaces and Bezier curves,
- split closed faces,
- convert C0 BSpline curve to a sequence of C1 BSpline curves.

All tools for particular cases are based on general tools for shape splitting but each of them has its own tools for splitting or converting geometry in accordance with the specified criteria.

General tools for shape splitting are:

- tool for splitting the whole shape,
- tool for splitting a face,
- tool for splitting wires.

Tools for shape splitting use tools for geometry splitting:

- tool for splitting surfaces,
- tool for splitting 3D curves,
- tool for splitting 2D curves.

Using tools available for shape splitting.

If it is necessary to split a shape by a specified continuity, split closed faces in the shape, split surfaces of revolution in the shape by angle or to convert all surfaces, all 3D curves, all 2D curves in the shape to Bezier, it is possible to use the existing/available tools.
The usual way to use these tools exception for the tool of converting a C0 BSpline curve is the following:

- a tool is created and initialized by shape.
- work precision for splitting and the maximum allowed tolerance are set
- the value of splitting criterion is set (if necessary)
- splitting is performed.
- splitting statuses are obtained.
- result is obtained
- the history of modification of the initial shape and its sub-shapes is output (this step is optional).

Let us, for example, split all surfaces and all 3D and 2D curves having a continuity of less the C2.

```cpp
//create a tool and initializes it by shape.
ShapeUpgrade_ShapeDivideContinuity
    ShapeDivideCont(initShape);

//set the working 3D and 2D precision and the maximum allowed tolerance
ShapeDivideCont.SetTolerance(prec);
ShapeDivideCont.SetTolerance2D(prec2d);
ShapeDivideCont.SetMaxTolerance(maxTol);

//set the values of criteria for surfaces, 3D curves and 2D curves.
ShapeDivideCont.SetBoundaryCriterion(GeomAbs_C2);
ShapeDivideCont.SetPCurveCriterion(GeomAbs_C2);
ShapeDivideCont.SetSurfaceCriterion(GeomAbs_C2);

//perform the splitting.
ShapeDivideCont.Perform();

//check the status and gets the result
if(ShapeDivideCont.Status(ShapeExtend_DONE)
    TopoDS_Shape result = ShapeDivideCont.GetResult();

//get the history of modifications made to faces
Creation of a new tool for splitting a shape.

To create a new splitting tool it is necessary to create tools for geometry splitting according to a desirable criterion. The new tools should be inherited from basic tools for geometry splitting. Then the new tools should be set into corresponding tools for shape splitting.

- a new tool for surface splitting should be set into the tool for face splitting
- new tools for splitting of 3D and 2D curves should be set into the splitting tool for wires.

To change the value of criterion of shape splitting it is necessary to create a new tool for shape splitting that should be inherited from the general splitting tool for shapes.

Let us split a shape according to a specified criterion.

```cpp
//creation of new tools for geometry splitting by a specified criterion.
Handle(MyTools_SplitSurfaceTool) MySplitSurfaceTool = new MyTools_SplitSurfaceTool;
Handle(MyTools_SplitCurve3DTool) MySplitCurve3Dtool = new MyTools_SplitCurve3DTool;
Handle(MyTools_SplitCurve2DTool) MySplitCurve2Dtool = new MyTools_SplitCurve2DTool;

//creation of a tool for splitting the shape and initialization of that tool by shape.
TopoDS_Shape initShape
MyTools_ShapeDivideTool ShapeDivide (initShape);
```
//setting of work precision for splitting and maximum allowed tolerance.
ShapeDivide.SetPrecision(prec);
ShapeDivide.SetMaxTolerance(MaxTol);

//setting of new splitting geometry tools in the shape splitting tools
Handle(ShapeUpgrade_FaceDivide) FaceDivide = ShapeDivide->GetSplitFaceTool();
Handle(ShapeUpgrade_WireDivide) WireDivide = FaceDivide->GetWireDivideTool();
FaceDivide->SetSplitSurfaceTool(MySplitSurfaceTool);
WireDivide->SetSplitCurve3dTool(MySplitCurve3DTool);
WireDivide->SetSplitCurve2dTool(MySplitCurve2DTool);

//setting of the value criterion.
ShapeDivide.SetValCriterion(val);

//shape splitting
ShapeDivide.Perform();

//getting the result
TopoDS_Shape splitShape = ShapeDivide.GetResult();

//getting the history of modifications of faces
for(TopExp_Explorer aExp(initShape,TopAbs_FACE);
aExp.More(0; aExp.Next()) { 
TopoDS_Shape modifShape = ShapeDivide.GetContext()->Apply(aExp.Current());
}
General splitting tools.

General tool for shape splitting

Class \textit{ShapeUpgrade\_ShapeDivide} provides shape splitting and converting according to the given criteria. It performs these operations for each face with the given tool for face splitting (\textit{ShapeUpgrade\_FaceDivide} by default).

This tool provides access to the tool for dividing faces with the help of the methods \textit{SetSplitFaceTool} and \textit{GetSplitFaceTool}.

General tool for face splitting

Class \textit{ShapeUpgrade\_FaceDivide} divides a Face (edges in the wires, by splitting 3D and 2D curves, as well as the face itself, by splitting the supporting surface) according to the given criteria.

The area of the face intended for division is defined by 2D curves of the wires on the Face. All 2D curves are supposed to be defined (in the parametric space of the supporting surface). The result is available after the call to the \textit{Perform} method. It is a Shell containing all resulting Faces. All modifications made during the splitting operation are recorded in the external context (\textit{ShapeBuild\_ReShape}).

This tool provides access to the tool for wire division and surface splitting by means of the following methods:

- \textit{SetWireDivideTool},
- \textit{GetWireDivideTool},
- \textit{SetSurfaceSplitTool},
- \textit{GetSurfaceSplitTool}.

General tool for wire splitting

Class \textit{ShapeUpgrade\_WireDivide} divides edges in the wire lying on the face or free wires or free edges with a given criterion. It splits the 3D curve and 2D curve(s) of the edge on the face. Other 2D curves, which
may be associated with the edge, are simply copied. If the 3D curve is split then the 2D curve on the face is split as well, and vice-versa. The original shape is not modified. Modifications made are recorded in the context (ShapeBuild_ReShape).

This tool provides access to the tool for dividing and splitting 3D and 2D curves by means of the following methods:

- `SetEdgeDivideTool`,
- `GetEdgeDivideTool`,
- `SetSplitCurve3dTool`,
- `GetSplitCurve3dTool`,
- `SetSplitCurve2dTool`,
- `GetSplitCurve2dTool`

and it also provides access to the mode for splitting edges by methods `SetEdgeMode` and `GetEdgeMode`.

This mode sets whether only free edges, only shared edges or all edges are split.

**General tool for edge splitting**

Class `ShapeUpgrade_EdgeDivide` divides edges and their geometry according to the specified criteria. It is used in the wire-dividing tool.

This tool provides access to the tool for dividing and splitting 3D and 2D curves by the following methods:

- `SetSplitCurve3dTool`,
- `GetSplitCurve3dTool`,
- `SetSplitCurve2dTool`,
- `GetSplitCurve2dTool`.

**General tools for geometry splitting**

There are three general tools for geometry splitting.

- General tool for surface splitting (`ShapeUpgrade_SplitSurface`)
- General tool for splitting 3D curves (`ShapeUpgrade_SplitCurve3d`)
- General tool for splitting 2D curves (`ShapeUpgrade_SplitCurve2d`)
All these tools are constructed the same way: They have methods:

- for initializing by geometry (method \textit{Init})
- for splitting (method \textit{Perform})
- for getting the status after splitting and the results:
  - \textit{Status} – for getting the result status;
  - \textit{ResSurface} – for splitting surfaces;
  - \textit{GetCurves} – for splitting 3D and 2D curves. During the process of splitting in the method \textit{Perform}:
- splitting values in the parametric space are computed according to a specified criterion (method \textit{Compute})
- splitting is made in accordance with the values computed for splitting (method \textit{Build}).

To create new tools for geometry splitting it is enough to inherit a new tool from the general tool for splitting a corresponding type of geometry and to redefine the method for computation of splitting values according to the specified criterion in them. (method \textit{Compute}).

Header file for the tool for surface splitting by continuity:

```cpp
class ShapeUpgrade_SplitSurfaceContinuity : public ShapeUpgrade_SplitSurface {
  Standard_EXPORT
    ShapeUpgrade_SplitSurfaceContinuity();

  //methods to set the criterion and the tolerance into the splitting tool
  Standard_EXPORT     void SetCriterion(const GeomAbs_Shape Criterion);
  Standard_EXPORT     void SetTolerance(const Standard_Real Tol);

  //redefinition of method Compute
  Standard_EXPORT virtual void Compute(const Standard_Boolean Segment);
  Standard_EXPORT
    ~ShapeUpgrade_SplitSurfaceContinuity();
```

private:
GeomAbs_Shape myCriterion;
Standard_Real myTolerance;
Standard_Integer myCont;
};
Specific splitting tools.

Conversion of shape geometry to the target continuity

Class *ShapeUpgrade.ShapeDivideContinuity* allows converting geometry with continuity less than the specified continuity to geometry with target continuity. If converting is not possible than geometrical object is split into several ones, which satisfy the given criteria. A topological object based on this geometry is replaced by several objects based on the new geometry.

```plaintext
ShapeUpgrade.ShapeDivideContinuity sdc (shape);
sdc.SetTolerance (tol3d);
sdc.SetTolerance3d (tol2d); // if known, else 1.e-09 is taken
sdc.SetBoundaryCriterion (GeomAbs_C2); // for Curves 3D
sdc.SetP CurveCriterion (GeomAbs_C2); // for Curves 2D
sdc.SetSurfaceCriterion (GeomAbs_C2); // for Surfaces
sdc.Perform ();
TopoDS_Shape bshape = sdc.Result ();
//... to also get the correspondances before/after
Handle (ShapeBuild_ReShape) ctx = sdc.Context ();
//.. on a given shape
if (ctx.IsRecorded (sh)) {
    TopoDS_Shape newsh = ctx->Value (sh);
    // if there are several results, they are recorded
    // inside a Compound.
    // .. process as needed
}
```

Splitting by angle

Class *ShapeUpgrade.ShapeDivideAngle* allows splitting all surfaces of revolution, cylindrical, toroidal, conical, spherical surfaces in the given shape so that each resulting segment covers not more than the defined angle (in radians).
Conversion of 2D, 3D curves and surfaces to Bezier

Class `ShapeUpgrade_ShapeConvertToBezier` is an API tool for performing a conversion of 3D, 2D curves to Bezier curves and surfaces to Bezier based surfaces (Bezier surface, surface of revolution based on Bezier curve, offset surface based on any of previous types).

This tool provides access to various flags for conversion of different types of curves and surfaces to Bezier by methods:

- **For 3D curves:**
  - `Set3dConversion`,
  - `Get3dConversion`,
  - `Set3dLineConversion`,
  - `Get3dLineConversion`,
  - `Set3dCircleConversion`,
  - `Get3dCircleConversion`,
  - `Set3dConicConversion`,
  - `Get3dConicConversion`

- **For 2D curves:**
  - `Set2dConversion`,
  - `Get2dConversion`

- **For surfaces:**
  - `GetSurfaceConversion`,
  - `SetPlaneMode`,
  - `GetPlaneMode`,
  - `SetRevolutionMode`,
  - `GetRevolutionMode`,
  - `SetExtrusionMode`,
  - `GetExtrusionMode`,
  - `SetBSplineMode`,
  - `GetBSplineMode`,

Let us attempt to produce a conversion of planes to Bezier surfaces.

```cpp
//Creation and initialization of a tool.
ShapeUpgrade_ShapeConvertToBezierSCB(Shape);
//setting tolerances
...
//setting mode for conversion of planes
```
Tool for splitting closed faces

Class *ShapeUpgrade_ShapeDivideClosed* provides splitting of closed faces in the shape to a defined number of components by the U and V parameters. It topologically and (partially) geometrically processes closed faces and performs splitting with the help of class *ShapeUpgrade_ClosedFaceDivide*.

```cpp
TopoDS_Shape aShape = ...;
ShapeUpgrade_ShapeDivideClosed tool (aShape );
Standard_Real closeTol = ...;
tool.SetPrecision(closeTol);
Standard_Real maxTol = ...;
tool.SetMaxTolerance(maxTol);
Standard_Integer NbSplitPoints = ...;
tool.SetNbSplitPoints(num);
if ( ! tool.Perform() && tool.Status (ShapeExtend_FAIL) ) {
    cout<<"Splitting of closed faces failed"<<endl;
    ...
}
TopoDS_Shape aResult = tool.Result();
```

Tool for splitting a C0 BSpline 2D or 3D curve to a sequence C1 BSpline curves

The API methods for this tool is a package of methods *ShapeUpgrade::C0BSplineToSequenceOfC1BsplineCurve*, which converts a C0 B-Spline curve into a sequence of C1 B-Spline curves. This method splits a B-Spline at the knots with multiplicities equal to degree, it does not use any tolerance and therefore does not change the geometry of the B-Spline. The method returns True if C0 B-Spline was
successfully split, otherwise returns False (if BS is C1 B-Spline).

**Tool for splitting faces**

*ShapeUpgrade_ShapeDivideArea* can work with compounds, solids, shells and faces. During the work this tool examines each face of a specified shape and if the face area exceeds the specified maximal area, this face is divided. Face splitting is performed in the parametric space of this face. The values of splitting in U and V directions are calculated with the account of translation of the bounding box form parametric space to 3D space.

Such calculations are necessary to avoid creation of strip faces. In the process of splitting the holes on the initial face are taken into account. After the splitting all new faces are checked by area again and the splitting procedure is repeated for the faces whose area still exceeds the max allowed area. Sharing between faces in the shape is preserved and the resulting shape is of the same type as the source shape.

An example of using this tool is presented in the figures below:

*Source Face*
ShapeUpgrade_ShapeDivideArea is inherited from the base class ShapeUpgrade_ShapeDivide and should be used in the following way:

- This class should be initialized on a shape with the help of the constructor or method Init() from the base class.
- The maximal allowed area should be specified by the method MaxArea().
- To produce a splitting use method Perform from the base class.
- The result shape can be obtained with the help the method Result().

```cpp
ShapeUpgrade_ShapeDivideArea tool (inputShape);
tool.MaxArea() = aMaxArea;
tool.Perform();
if(tool.Status(ShapeExtend_DONE)) {
    TopoDS_Shape ResultShape = tool.Result();
    ShapeFix::SameParameter ( ResultShape, 
        Standard_False );
}
```

**Note** that the use of method `ShapeFix::SameParameter` is necessary, otherwise the parameter edges obtained as a result of splitting can be different.

**Additional methods**
- Class `ShapeUpgrade_FaceDivideArea` inherited from `ShapeUpgrade_FaceDivide` is intended for splitting a face by the maximal area criterion.
- Class `ShapeUpgrade_SplitSurfaceArea` inherited from `Shape Upgrade_SplitSurface` calculates the parameters of face splitting in the parametric space.
Customization of shapes

Customization tools are intended for adaptation of shape geometry in compliance with the customer needs. They modify a geometrical object to another one in the shape.

To implement the necessary shape modification it is enough to initialize the appropriate tool by the shape and desirable parameters and to get the resulting shape. For example for conversion of indirect surfaces in the shape do the following:

```plaintext
TopoDS_Shape initialShape ..
TopoDS_Shape resultShape =
    ShapeCustom::DirectFaces(initialShape);
```

Conversion of indirect surfaces.

```plaintext
ShapeCustom::DirectFaces
    static TopoDS_Shape DirectFaces(const TopoDS_Shape& S);
```

This method provides conversion of indirect elementary surfaces (elementary surfaces with left-handed coordinate systems) in the shape into direct ones. New 2d curves (recomputed for converted surfaces) are added to the same edges being shared by both the resulting shape and the original shape S.

Shape Scaling

```plaintext
ShapeCustom::ScaleShape
    TopoDS_Shape ShapeCustom::ScaleShape(const TopoDS_Shape& S,
        const Standard_Real scale);
```

This method returns a new shape, which is a scaled original shape with a coefficient equal to the specified value of scale. It uses the tool ShapeCustom::TrsfModification.
Conversion of curves and surfaces to BSpline

*ShapeCustom_BSplineRestriction* allows approximation of surfaces, curves and 2D curves with a specified degree, maximum number of segments, 2d tolerance and 3d tolerance. If the approximation result cannot be achieved with the specified continuity, the latter can be reduced.

The method with all parameters looks as follows:

```cpp
ShapeCustom::BsplineRestriction
    TopoDS_Shape ShapeCustom::BsplineRestriction
    (const TopoDS_Shape& S,
    const Standard_Real Tol3d, const
    Standard_Real Tol2d,
    const Standard_Integer MaxDegree,
    const Standard_Integer MaxNbSegment,
    const GeomAbs_Shape Continuity3d,
    const GeomAbs_Shape Continuity2d,
    const Standard_Boolean Degree,
    const Standard_Boolean Rational,
    const
    Handle(ShapeCustom_RestrictionParameters)&
    aParameters)
```

It returns a new shape with all surfaces, curves and 2D curves of BSpline/Bezier type or based on them, converted with a degree less than *MaxDegree* or with a number of spans less then *NbMaxSegment* depending on the priority parameter *Degree*. If this parameter is equal to True then *Degree* will be increased to the value *GmaxDegree*, otherwise *NbMaxSegments* will be increased to the value *GmaxSegments*. *GmaxDegree* and *GMaxSegments* are the maximum possible degree and the number of spans correspondingly. These values will be used in cases when an approximation with specified parameters is impossible and either *GmaxDegree* or *GMaxSegments* is selected depending on the priority.

Note that if approximation is impossible with *GMaxDegree*, even then the number of spans can exceed the specified *GMaxSegment*. *Rational*
specifies whether Rational BSpline/Beziershould be converted into polynomial B-Spline.

Also note that the continuity of surfaces in the resulting shape can be less than the given value.

Flags

To convert other types of curves and surfaces to BSpline with required parameters it is necessary to use flags from class ShapeCustom::RestrictionParameters, which is just a container of flags. The following flags define whether a specified-type geometry has been converted to BSpline with the required parameters:

- ConvertPlane,
- ConvertBezierSurf,
- ConvertRevolutionSurf,
- ConvertExtrusionSurf,
- ConvertOffsetSurf,
- ConvertCurve3d, – for conversion of all types of 3D curves.
- ConvertOffsetCurv3d, – for conversion of offset 3D curves.
- ConvertCurve2d, – for conversion of all types of 2D curves.
- ConvertOffsetCurv2d, – for conversion of offset 2D curves.
- SegmentSurfaceMode – defines whether the surface would be approximated within the boundaries of the face lying on this surface.

Conversion of elementary surfaces into surfaces of revolution

ShapeCustom::ConvertToRevolution()
TopoDS_Shape
ShapeCustom::ConvertToRevolution(const TopoDS_Shape& S);

This method returns a new shape with all elementary periodic surfaces converted to Geom_SurfaceOfRevolution. It uses the tool ShapeCustom::ConvertToRevolution.

Conversion of elementary surfaces into Bspline surfaces
ShapeCustom::ConvertToBSpline()

```cpp
TopoDS_Shape ShapeCustom::ConvertToBSpline( const TopoDS_Shape& S,
const Standard_Boolean extrMode,
const Standard_Boolean revolMode,
const Standard_Boolean offsetMode);
```

This method returns a new shape with all surfaces of linear extrusion, revolution and offset surfaces converted according to flags to `Geom_BSplineSurface` (with the same parameterization). It uses the tool `ShapeCustom_ConvertToBSpline`.

### Getting the history of modification of sub-shapes.

If, in addition to the resulting shape, you want to get the history of modification of sub-shapes you should not use the package methods described above and should use your own code instead:

1. Create a tool that is responsible for the necessary modification.
2. Create the tool `BRepTools_Modifier` that performs a specified modification in the shape.
3. To get the history and to keep the assembly structure use the method `ShapeCustom::ApplyModifier`.

The general calling syntax for scaling is

```cpp
TopoDS_Shape scaled_shape = 
    ShapeCustom::ScaleShape(shape, scale);
```

Note that `scale` is a real value. You can refine your mapping process by using additional calls to follow shape mapping sub-shape by sub-shape. The following code along with pertinent includes can be used:

```cpp
p_Trsf T;
Standard_Real scale = 100; // for example!
T.SetScale (gp_Pnt (0, 0, 0), scale);
Handle(ShapeCustom_TrsfModification) TM = new ShapeCustom_TrsfModification(T);
TopTools_DataMapOfShapeShape context;
```
BRepTools_Modifier MD;
TopoDS_Shape res = ShapeCustom::ApplyModifier (Shape, TM, context, MD);

The map, called context in our example, contains the history. Substitutions are made one by one and all shapes are transformed. To determine what happens to a particular sub-shape, it is possible to use:

TopoDS_Shape oneres = context.Find (oneshape);
//In case there is a doubt, you can also add:
if (context.IsBound(oneshape)) oneres =  
    context.Find(oneshape);
//You can also sweep the entire data map by using:
TopTools_DataMapIteratorOfDataMapOfShapeShape
//To do this, enter:
for(TopTools_DataMapIteratorOfDataMapOfShapeShape
iter(context);iter(more ();iter.next ()) {
    TopoDs_Shape oneshape = iter.key ();
    TopoDs_Shape oneres = iter.value ();
}

Remove internal wires

ShapeUpgrade_RemoveInternalWires tool removes internal wires with contour area less than the specified minimal area. It can work with compounds, solids, shells and faces.

If the flag RemoveFaceMode is set to TRUE, separate faces or a group of faces with outer wires, which consist only of edges that belong to the removed internal wires, are removed (seam edges are not taken into account). Such faces can be removed only for a sewed shape.

Internal wires can be removed by the methods Perform. Both methods Perform can not be carried out if the class has not been initialized by the shape. In such case the status of Perform is set to FAIL.

The method Perform without arguments removes from all faces in the specified shape internal wires whose area is less than the minimal area.
The other method *Perform* has a sequence of shapes as an argument. This sequence can contain faces or wires. If the sequence of shapes contains wires, only the internal wires are removed.

If the sequence of shapes contains faces, only the internal wires from these faces are removed.

- The status of the performed operation can be obtained using method *Status()*;
- The resulting shape can be obtained using method *GetResult()*.

An example of using this tool is presented in the figures below:
After the processing three internal wires with contour area less than the specified minimal area have been removed. One internal face has been removed. The outer wire of this face consists of the edges belonging to the removed internal wires and a seam edge. Two other internal faces have not been removed because their outer wires consist not only of edges belonging to the removed wires.
After the processing six internal wires with contour area less than the specified minimal area have been removed. Six internal faces have been removed. These faces can be united into groups of faces. Each group of faces has an outer wire consisting only of edges belonging to the removed internal wires. Such groups of faces are also removed.

The example of method application is also given below:

```cpp
//Initialization of the class by shape.
Handle(ShapeUpgrade_RemoveInternalWires) aTool = new ShapeUpgrade_RemoveInternalWires(inputShape);
//setting parameters
aTool->MinArea() = aMinArea;
aTool->RemoveFaceMode() = aModeRemoveFaces;

//when method Perform is carried out on separate shapes.
aTool->Perform(aSeqShapes);

//when method Perform is carried out on whole shape.
aTool->Perform();
//check status set after method Perform
if(aTool->Status(ShapeExtend_FAIL) {
```
cout << "Operation failed" << endl;
    return;
}

if(aTool->Status(ShapeExtend_DONE1)) {
    const TopTools_SequenceOfShape& aRemovedWires = aTool->RemovedWires();
    cout << aRemovedWires.Length() << " internal wires were removed" << endl;
}

if(aTool->Status(ShapeExtend_DONE2)) {
    const TopTools_SequenceOfShape& aRemovedFaces = aTool->RemovedFaces();
    cout << aRemovedFaces.Length() << " small faces were removed" << endl;
}

    // getting result shape
    TopoDS_Shape res = aTool->GetResult();

**Conversion of surfaces**

Class ShapeCustom_Surface allows:

- converting BSpline and Bezier surfaces to the analytical form (using method `ConvertToAnalytical()`)
- converting closed B-Spline surfaces to periodic ones (using method `ConvertToPeriodic`)

To convert surfaces to analytical form this class analyzes the form and the closure of the source surface and defines whether it can be approximated by analytical surface of one of the following types:

- `Geom_Plane`,
- `Geom_SphericalSurface`,
- `Geom_CylindricalSurface`,
The conversion is done only if the new (analytical) surface does not deviate from the source one more than by the given precision.

```cpp
Handle(Geom_Surface) initSurf;
ShapeCustom_Surface ConvSurf(initSurf);
//conversion to analytical form
Handle(Geom_Surface) newSurf =
    ConvSurf.ConvertToAnalytical(allowedtol,Standard_False);
//or conversion to a periodic surface
Handle(Geom_Surface) newSurf =
    ConvSurf.ConvertToPeriodic(Standard_False);
//getting the maximum deviation of the new surface from the initial surface
Standard_Real maxdist = ConvSurf.Gap();
```

**Unify Same Domain**

`ShapeUpgrade_UnifySameDomain` tool allows unifying all possible faces and edges of a shape, which lies on the same geometry. Faces/edges are considered as 'same-domain' if the neighboring faces/edges lie on coincident surfaces/curves. Such faces/edges can be unified into one face/edge. This tool takes an input shape and returns a new one. All modifications of the initial shape are recorded during the operation.

The following options are available:

- If the flag `UnifyFaces` is set to TRUE, `UnifySameDomain` tries to unify all possible faces;
- If the flag `UnifyEdges` is set to TRUE, `UnifySameDomain` tries to unify all possible edges;
- if the flag `ConcatBSplines` is set to TRUE, all neighboring edges, which lie on the BSpline or Bezier curves with C1 continuity on their common vertices will be merged into one common edge.

By default, `UnifyFaces` and `UnifyEdges` are set to TRUE; `ConcatBSplines`
is set to FALSE.

The common methods of this tool are as follows:

- Method `Build()` is used to unify.
- Method `Shape()` is used to get the resulting shape.
- Method `Generated()` is used to get a new common shape from the old shape. If a group of edges has been unified into one common edge then method `Generated()` called on any edge from this group will return the common edge. The same goes for the faces.

The example of the usage is given below:

```cpp
// 'Sh' is the initial shape
ShapeUpgrade_UnifySameDomain USD(Sh, true, true,
   true); // UnifyFaces mode on, UnifyEdges mode
   on, ConcatBSplines mode on.
USD.Build();
//get the result
TopoDS_Shape Result = USD.Shape();
//Let Sh1 as a part of Sh
//get the new (probably unified) shape form the Sh1
TopoDS_Shape ResSh1 = USD.Generated(Sh1);
```
Auxiliary tools for repairing, analysis and upgrading

Tool for rebuilding shapes

Class *ShapeBuild_ReShape* rebuilds a shape by making predefined substitutions on some of its components. During the first phase, it records requests to replace or remove some individual shapes. For each shape, the last given request is recorded. Requests may be applied as *Oriented* (i.e. only to an item with the same orientation) or not (the orientation of the replacing shape corresponds to that of the original one). Then these requests may be applied to any shape, which may contain one or more of these individual shapes.

This tool has a flag for taking the location of shapes into account (for keeping the structure of assemblies) (*ModeConsiderLocation*). If this mode is equal to Standard_True, the shared shapes with locations will be kept. If this mode is equal to Standard_False, some different shapes will be produced from one shape with different locations after rebuilding. By default, this mode is equal to Standard_False.

To use this tool for the reconstruction of shapes it is necessary to take the following steps:

1. Create this tool and use method *Apply()* for its initialization by the initial shape. Parameter *until* sets the level of shape type and requests are taken into account up to this level only. Sub-shapes of the type standing beyond the line set by parameter until will not be rebuilt and no further exploration will be done.
2. Replace or remove sub-shapes of the initial shape. Each sub-shape can be replaced by a shape of the same type or by shape containing shapes of that type only (for example, *TopoDS_Edge* can be replaced by *TopoDS_Edge*, *TopoDS_Wire* or *TopoDS_Compound* containing *TopoDS_Edges*). If an incompatible shape type is encountered, it is ignored and flag FAIL1 is set in Status. For a sub-shape it is recommended to use method *Apply* before methods *Replace* and *Remove*, because the sub-shape has already been
changed for the moment by its previous modifications or modification of its sub-shape (for example TopoDS_Edge can be changed by a modification of its TopoDS_Vertex, etc.).

3. Use method Apply for the initial shape again to get the resulting shape after all modifications have been made.

4. Use method Apply to obtain the history of sub-shape modification.

Additional method IsNewShape can be used to check if the shape has been recorded by BRepTools_ReShape tool as a value.

**Note** that in fact class ShapeBuild_ReShape is an alias for class BRepTools_ReShape. They differ only in queries of statuses in the ShapeBuild_ReShape class.

Let us use the tool to get the result shape after modification of sub-shapes of the initial shape:

```cpp
TopoDS_Shape initialShape...
//creation of a rebuilding tool
Handle(ShapeBuild_ReShape) Context = new ShapeBuild_ReShape.

//next step is optional. It can be used for keeping the assembly structure.
Context->ModeConsiderLocation = Standard_True;

//initialization of this tool by the initial shape
Context->Apply(initialShape);
...
//getting the intermediate result for replacing subshape1 with the modified subshape1.
TopoDS_Shape tempshape1 = Context->Apply(subshape1);

//replacing the intermediate shape obtained from subshape1 with the newsubshape1.
Context->Replace(tempsubshape1,newsubshape1);
...
//for removing the sub-shape
TopoDS_Shape tempshape2 = Context->Apply(subshape2);
```
Context->Remove(tempsubshape2);

//getting the result and the history of modification
TopoDS_Shape resultShape = Context->Apply(initialShape);

//getting the resulting sub-shape from the subshape1 of the initial shape.
TopoDS_Shape result_subshape1 = Context->Apply(subshape1);
**Status definition**

*ShapExtend_Status* is used to report the status after executing some methods that can either fail, do something, or do nothing. The status is a set of flags *DONEi* and *FAILi*. Any combination of them can be set at the same time. For exploring the status, enumeration is used.

The values have the following meaning:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>OK</em>,</td>
<td>Nothing is done, everything OK</td>
</tr>
<tr>
<td><em>DONE1</em>,</td>
<td>Something was done, case 1</td>
</tr>
<tr>
<td><em>DONE8</em>,</td>
<td>Something was done, case 8</td>
</tr>
<tr>
<td><em>DONE</em>,</td>
<td>Something was done (any of <em>DONE</em>#)</td>
</tr>
<tr>
<td><em>FAIL1</em>,</td>
<td>The method failed, case 1</td>
</tr>
<tr>
<td><em>FAIL8</em>,</td>
<td>The method failed, case 8</td>
</tr>
<tr>
<td><em>FAIL</em></td>
<td>The method failed (any of <em>FAIL</em># occurred)</td>
</tr>
</tbody>
</table>
Tool representing a wire

Class `ShapeExtend_WireData` provides a data structure necessary to work with the wire as with an ordered list of edges, and that is required for many algorithms. The advantage of this class is that it allows to work with incorrect wires.

The object of the class `ShapeExtend_WireData` can be initialized by `TopoDS_Wire` and converted back to `TopoDS_Wire`.

An edge in the wire is defined by its rank number. Operations of accessing, adding and removing an edge at/to the given rank number are provided. Operations of circular permutation and reversing (both orientations of all edges and the order of edges) are provided on the whole wire as well.

This class also provides a method to check if the edge in the wire is a seam (if the wire lies on a face).

Let us remove edges from the wire and define whether it is seam edge

```cpp
TopoDS_Wire ini = ..
Handle(ShapeExtend_Wire) asewd = new
   ShapeExtend_Wire(initwire);
//Removing edge Edge1 from the wire.
Standard_Integer index_edge1 = asewd->Index(Edge1);
asewd.Remove(index_edge1);
//Definition of whether Edge2 is a seam edge
Standard_Integer index_edge2 = asewd->Index(Edge2);
asewd->IsSeam(index_edge2);
```
Tool for exploring shapes

Class ShapeExtend_Explorer is intended to explore shapes and convert different representations (list, sequence, compound) of complex shapes. It provides tools for:

- obtaining the type of the shapes in the context of TopoDS_Compound,
- exploring shapes in the context of TopoDS_Compound,
- converting different representations of shapes (list, sequence, compound).
Tool for attaching messages to objects

Class ShapeExtend_MsgRegistrator attaches messages to objects (generic Transient or shape). The objects of this class are transmitted to the Shape Healing algorithms so that they could collect messages occurred during shape processing. Messages are added to the Maps (stored as a field) that can be used, for instance, by Data Exchange processors to attach those messages to initial file entities.

Let us send and get a message attached to object:

```cpp
Handle(ShapeExtend_MsgRegistrator) MessageReg = new ShapeExtend_MsgRegistrator;
//attaches messages to an object (shape or entity)
Message_Msg msg..
TopoDS_Shape Shape1...
MessageReg->Send(Shape1,msg,Message_WARNING);
Handle(Standard_Transient) ent ..
MessageReg->Send(ent,msg,Message_WARNING);
//gets messages attached to shape
const ShapeExtend_DataMapOfShapeListOfMsg& msgmap = MessageReg->MapShape();
if (msgmap.IsBound (Shape1)) {
  const Message_ListOfMsg &msglist = msgmap.Find (Shape1);
  for (Message_ListIteratorOfListOfMsg iter (msglist); iter.More(); iter.Next()) {
    Message_Msg msg = iter.Value();
  }
} else {
  // No messages found.
}
```
Tools for performance measurement

Classes \textit{MoniTool.Timer} and \textit{MoniTool.TimerSentry} are used for measuring the performance of a current operation or any part of code, and provide the necessary API. Timers are used for debugging and performance optimizing purposes.

Let us try to use timers in \textit{XSDRAWIGES.cxx} and \textit{IGESBRep_Reader.cxx} to analyse the performance of command \textit{igesbrep}:

\textbf{XSDRAWIGES.cxx}

...  
#include \textless MoniTool_Timer.hxx\textgreater  
#include \textless MoniTool_TimerSentry.hxx\textgreater  
...  
MoniTool_Timer::ClearTimers();  
...  
MoniTool_TimerSentry MTS("IGES_LoadFile");  
Standard_Integer status =  
\hspace{1em} \text{Reader.LoadFile(fnom.ToCString());}  
MTS.Stop();  
...  
MoniTool_Timer::DumpTimers(cout);  
return;

\textbf{IGESBRep_Reader.cxx}

...  
#include \textless MoniTool_TimerSentry.hxx\textgreater  
...  
Standard_Integer nb = theModel->NbEntities();  
...  
for (Standard_Integer i=1; i<=nb; i++) {  
\hspace{1em} MoniTool_TimerSentry MTS("IGESToBRep_Transfer");  
\hspace{1em} try {  

TP.Transfer(ent);
    shape = TransferBRep::ShapeResult
       (theProc,ent);
}
...
}

The result of *DumpTimer()* after file translation is as follows:

<table>
<thead>
<tr>
<th>TIMER</th>
<th>Elapsed</th>
<th>CPU User</th>
<th>CPU Sys</th>
<th>Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGES_LoadFile</td>
<td>1.0 sec</td>
<td>0.9 sec</td>
<td>0.0 sec</td>
<td>1</td>
</tr>
<tr>
<td>IGESToBRep_Transfer</td>
<td>14.5 sec</td>
<td>4.4 sec</td>
<td>0.1 sec</td>
<td>1311</td>
</tr>
</tbody>
</table>
Shape Processing

Usage Workflow

The Shape Processing module allows defining and applying the general Shape Processing as a customizable sequence of Shape Healing operators. The customization is implemented via the user-editable resource file, which defines the sequence of operators to be executed and their parameters.

The Shape Processing functionality is implemented with the help of the XSAlgo interface. The main function

\[ \text{XSAlgo::AlgoContainer::ProcessShape()} \]

does shape processing with specified tolerances and returns the resulting shape and associated information in the form of *Transient*.

This function is used in the following way:

```cpp
TopoDS_Shape aShape = ...;
Standard_Real Prec = ..., Standard_Real MaxTol = ...;
TopoDS_Shape aResult;
Handle(Standard_Transient) info;
TopoDS_Shape aResult = XSAlgo::AlgoContainer()->
    ProcessShape(aShape, Prec, MaxTol, "Name of ResourceFile", "NameSequence", info);
```

Let us create a custom sequence of operations:

1. Create a resource file with the name *ResourceFile*, which includes the following string:

\[
\text{NameSequence.exec.op: MyOper}
\]

where *MyOper* is the name of operation.

2. Input a custom parameter for this operation in the resource file, for example:

\[
\text{NameSequence.MyOper.Tolerance: 0.01}
\]
where \textit{Tolerance} is the name of the parameter and 0.01 is its value.

3. Add the following string into \textit{void ShapeProcess\_OperLibrary::Init()}:

\begin{verbatim}
ShapeProcess::RegisterOperator(;MyOper;,
new ShapeProcess\_UOperator(myfunction));
\end{verbatim}

where \textit{myfunction} is a function which implements the operation.

4. Create this function in \textit{ShapeProcess\_OperLibrary} as follows:

\begin{verbatim}
static Standard\_Boolean myfunction (const
Handle(ShapeProcess\_Context)&
context)
{
    Handle(ShapeProcess\_ShapeContext) ctx =
    Handle(ShapeProcess\_ShapeContext)::DownCast(c
    ontext);
    if(ctx.IsNull()) return Standard\_False;
    TopoDS\_Shape aShape = ctx->Result();
    //receive our parameter:
    Standard\_Real toler;
    ctx->GetReal(;Tolerance;,
toler);
    return Standard\_True;
}
\end{verbatim}

5. Make the necessary operations with \textit{aShape} using the received value of parameter \textit{Tolerance} from the resource file.

6. Define some operations (with their parameters) \textit{MyOper1, MyOper2, MyOper3}, etc. and describe the corresponding functions in \textit{ShapeProcess\_OperLibrary}.

7. Perform the required sequence using the specified name of operations and values of parameters in the resource file.

For example: input of the following string:

\begin{verbatim}
NameSequence.exec.op: MyOper1,MyOper3
\end{verbatim}

means that the corresponding functions from \textit{ShapeProcess\_OperLibrary} will be performed with the original shape \textit{aShape} using parameters defined for \textit{MyOper1} and \textit{MyOper3} in the resource file.

It is necessary to note that these operations will be performed step by step and the result obtained after performing the first operation will be
used as the initial shape for the second operation.
Operators

DirectFaces

This operator sets all faces based on indirect surfaces, defined with left-handed coordinate systems as direct faces. This concerns surfaces defined by Axis Placement (Cylinders, etc). Such Axis Placement may be indirect, which is allowed in Cascade, but not allowed in some other systems. This operator reverses indirect placements and recomputes PCurves accordingly.

SameParameter

This operator is required after calling some other operators, according to the computations they do. Its call is explicit, so each call can be removed according to the operators, which are either called or not afterwards. This mainly concerns splitting operators that can split edges.

The operator applies the computation SameParameter which ensures that various representations of each edge (its 3d curve, the pcurve on each of the faces on which it lies) give the same 3D point for the same parameter, within a given tolerance.

- For each edge coded as same parameter, deviation of curve representation is computed and if the edge tolerance is less than that deviation, the tolerance is increased so that it satisfies the deviation. No geometry modification, only an increase of tolerance is possible.
- For each edge coded as not same parameter the deviation is computed as in the first case. Then an attempt is made to achieve the edge equality to same parameter by means of modification of 2d curves. If the deviation of this modified edge is less than the original deviation then this edge is returned, otherwise the original edge (with non-modified 2d curves) is returned with an increased (if necessary) tolerance. Computation is done by call to the standard algorithm BRepLib::SameParameter.

This operator can be called with the following parameters:
- **Boolean : Force** (optional) – if True, encodes all edges as *not same parameter* then runs the computation. Else, the computation is done only for those edges already coded as *not same parameter*.
- **Real : Tolerance3d** (optional) – if not defined, the local tolerance of each edge is taken for its own computation. Else, this parameter gives the global tolerance for the whole shape.

**BSplineRestriction**

This operator is used for conversion of surfaces, curves 2d curves to BSpline surfaces with a specified degree and a specified number of spans. It performs approximations on surfaces, curves and 2d curves with a specified degree, maximum number of segments, 2d tolerance, 3d tolerance. The specified continuity can be reduced if the approximation with a specified continuity was not done successfully.

This operator can be called with the following parameters:

- **Boolean : SurfaceMode** allows considering the surfaces;
- **Boolean : Curve3dMode** allows considering the 3d curves;
- **Boolean : Curve2dMode** allows considering the 2d curves;
- **Real : Tolerance3d** defines 3d tolerance to be used in computation;
- **Real : Tolerance2d** defines 2d tolerance to be used when computing 2d curves;
- **GeomAbs_Shape (C0 G1 C1 G2 C2 CN) : Continuity3d** is the continuity required in 2d;
- **GeomAbs_Shape (C0 G1 C1 G2 C2 CN) : Continuity2d** is the continuity required in 3d;
- **Integer : RequiredDegree** gives the required degree;
- **Integer : Required NbSegments** gives the required number of segments;
- **Boolean : PreferDegree** if true, **RequiredDegree** has a priority, else **Required NbSegments** has a priority;
- **Boolean : RationalToPolynomial** serves for conversion of BSplines to polynomial form;
- **Integer : MaxDegree** gives the maximum allowed Degree, if **RequiredDegree** cannot be reached;
- **Integer : Max NbSegments** gives the maximum allowed NbSegments, if **Required NbSegments** cannot be reached.
The following flags allow managing the conversion of special types of curves or surfaces, in addition to BSpline. They are controlled by `SurfaceMode`, `Curve3dMode` or `Curve2dMode` respectively; by default, only BSplines and Bezier Geometries are considered:

- Boolean : `OffsetSurfaceMode`
- Boolean : `LinearExtrusionMode`
- Boolean : `RevolutionMode`
- Boolean : `OffsetCurve3dMode`
- Boolean : `OffsetCurve2dMode`
- Boolean : `PlaneMode`
- Boolean : `BezierMode`
- Boolean : `ConvCurve3dMode`
- Boolean : `ConvCurve2dMode`

For each of the Mode parameters listed above, if it is True, the specified geometry is converted to BSpline, otherwise only its basic geometry is checked and converted (if necessary) keeping the original type of geometry (revolution, offset, etc).

- Boolean : `SegmentSurfaceMode` has effect only for Bsplines and Bezier surfaces. When False a surface will be replaced by a Trimmed Surface, else new geometry will be created by splitting the original Bspline or Bezier surface.

**ElementaryToRevolution**

This operator converts elementary periodic surfaces to SurfaceOfRevolution.

**SplitAngle**

This operator splits surfaces of revolution, cylindrical, toroidal, conical, spherical surfaces in the given shape so that each resulting segment covers not more than the defined number of degrees.

It can be called with the following parameters:

- Real : `Angle` – the maximum allowed angle for resulting faces;
- Real : `MaxTolerance` – the maximum tolerance used in
computations.

**SurfaceToBSpline**

This operator converts some specific types of Surfaces, to BSpline (according to parameters). It can be called with the following parameters:

- *Boolean : LinearExtrusionMode* allows converting surfaces of Linear Extrusion;
- *Boolean : RevolutionMode* allows converting surfaces of Revolution;
- *Boolean : OffsetMode* allows converting Offset Surfaces

**ToBezier**

This operator is used for data supported as Bezier only and converts various types of geometries to Bezier. It can be called with the following parameters used in computation of conversion:

- *Boolean : SurfaceMode*
- *Boolean : Curve3dMode*
- *Boolean : Curve2dMode*
- *Real : MaxTolerance*
- *Boolean : SegmentSurfaceMode* (default is True) has effect only for Bsplines and Bezier surfaces. When False a surface will be replaced by a Trimmed Surface, else new geometry will be created by splitting the original Bspline or Bezier surface.

The following parameters are controlled by *SurfaceMode, Curve3dMode* or *Curve2dMode* (according to the case):

- *Boolean : Line3dMode*
- *Boolean : Circle3dMode*
- *Boolean : Conic3dMode*
- *Boolean : PlaneMode*
- *Boolean : RevolutionMode*
- *Boolean : ExtrusionMode*
- *Boolean : BSplineMode*

**SplitContinuity**
This operator splits a shape in order to have each geometry (surface, curve 3d, curve 2d) correspond the given criterion of continuity. It can be called with the following parameters:

- *Real* : `Tolerance3d`
- *Integer (GeomAbs_Shape)* : `CurveContinuity`
- *Integer (GeomAbs_Shape)* : `SurfaceContinuity`
- *Real* : `MaxTolerance`

Because of algorithmic limitations in the operator `BSplineRestriction` (in some particular cases, this operator can produce unexpected C0 geometry), if `SplitContinuity` is called, it is recommended to call it after `BSplineRestriction`. Continuity Values will be set as `GeomAbs_Shape` (i.e. C0 G1 C1 G2 C2 CN) besides direct integer values (resp. 0 1 2 3 4 5).

**SplitClosedFaces**

This operator splits faces, which are closed even if they are not revolutionary or cylindrical, conical, spherical, toroidal. This corresponds to BSpline or Bezier surfaces which can be closed (whether periodic or not), hence they have a seam edge. As a result, no more seam edges remain. The number of points allows to control the minimum count of faces to be produced per input closed face.

This operator can be called with the following parameters:

- *Integer* : `NbSplitPoints` gives the number of points to use for splitting (the number of intervals produced is `NbSplitPoints+1`);
- *Real* : `CloseTolerance` tolerance used to determine if a face is closed;
- *Real* : `MaxTolerance` is used in the computation of splitting.

**FixGaps**

This operator must be called when `FixFaceSize` and/or `DropSmallEdges` are called. Using Surface Healing may require an additional call to `BSplineRestriction` to ensure that modified geometries meet the requirements for BSpline. This operators repairs geometries which contain gaps between edges in wires (always performed) or gaps on
faces, controlled by parameter *SurfaceMode*, Gaps on Faces are fixed by using algorithms of Surface Healing This operator can be called with the following parameters:

- **Real : Tolerance3d** sets the tolerance to reach in 3d. If a gap is less than this value, it is not fixed.
- **Boolean : SurfaceMode** sets the mode of fixing gaps between edges and faces (yes/no)
- **Integer : SurfaceAddSpans** sets the number of spans to add to the surface in order to fix gaps;
- **GeomAbs_Shape (C0 G1 C1 G2 C2 CN) : SurfaceContinuity** sets the minimal continuity of a resulting surface;
- **Integer : NbIterations** sets the number of iterations
- **Real : Beta** sets the elasticity coefficient for modifying a surface [1-1000] ;
- **Reals : Coeff1 to Coeff6** sets energy coefficients for modifying a surface [0-10000] ;
- **Real : MaxDeflection** sets maximal deflection of surface from an old position.

This operator may change the original geometry. In addition, it is CPU consuming, and it may fail in some cases. Also **FixGaps** can help only when there are gaps obtained as a result of removal of small edges that can be removed by **DropSmallEdges** or **FixFaceSize**.

**FixFaceSize**

This operator removes faces, which are small in all directions (spot face) or small in one direction (strip face). It can be called with the parameter **Real : Tolerance**, which sets the minimal dimension, which is used to consider a face, is small enough to be removed.

**DropSmallEdges**

This operator drops edges in a wire, and merges them with adjacent edges, when they are smaller than the given value (Tolerance3d) and when the topology allows such merging (i.e. same adjacent faces for each of the merged edges). Free (non-shared by adjacent faces) small edges can be also removed in case if they share the same vertex Parameters.
It can be called with the parameter $Real : Tolerance3d$, which sets the dimension used to determine if an edge is small.

**FixShape**

This operator may be added for fixing invalid shapes. It performs various checks and fixes, according to the modes listed hereafter. Management of a set of fixes can be performed by flags as follows:

- if the flag for a fixing tool is set to 0, it is not performed;
- if set to 1, it is performed in any case;
- if not set, or set to -1, for each shape to be applied on, a check is done to evaluate whether a fix is needed. The fix is performed if the check is positive.

By default, the flags are not set, the checks are carried out each individual shape.

This operator can be called with the following parameters:

- $Real : Tolerance3d$ sets basic tolerance used for fixing;
- $Real : MaxTolerance3d$ sets maximum allowed value for the resulting tolerance;
- $Real : MinTolerance3d$ sets minimum allowed value for the resulting tolerance.
- $Boolean : FixFreeShellMode$
- $Boolean : FixFreeFaceMode$
- $Boolean : FixFreeWireMode$
- $Boolean : FixSameParameterMode$
- $Boolean : FixSolidMode$
- $Boolean : FixShellMode$
- $Boolean : FixFaceMode$
- $Boolean : FixWireMode$
- $Boolean : FixOrientationMode$
- $Boolean : FixMissing Seam Mode$
- $Boolean : FixSmall Area Wire Mode$
- $Boolean (not checked) : ModifyTopologyMode$ specifies the mode for modifying topology. Should be False (default) for shapes with shells and can be True for free faces.
- $Boolean (not checked) : ModifyGeometryMode$ specifies the mode for modifying geometry. Should be False if geometry is to be kept
and True if it can be modified.

- **Boolean (not checked)**: *ClosedWireMode* specifies the mode for wires. Should be True for wires on faces and False for free wires.
- **Boolean (not checked)**: *PreferencePCurveMode (not used)* specifies the preference of 3d or 2d representations for an edge

*Boolean*: *FixReorderMode*
*Boolean*: *FixSmallMode*
*Boolean*: *FixConnectedMode*
*Boolean*: *FixEdgeCurvesMode*
*Boolean*: *FixDegeneratedMode*
*Boolean*: *FixLackingMode*
*Boolean*: *FixSelfIntersectionMode*
*Boolean*: *FixGaps3dMode*
*Boolean*: *FixGaps2dMode*
*Boolean*: *FixReversed2dMode*
*Boolean*: *FixRemovePCurveMode*
*Boolean*: *FixRemoveCurve3dMode*
*Boolean*: *FixAddPCurveMode*
*Boolean*: *FixAddCurve3dMode*
*Boolean*: *FixSeamMode*
*Boolean*: *FixShiftedMode*
*Boolean*: *FixEdgeSameParameterMode*
*Boolean*: *FixSelfIntersectingEdgeMode*
*Boolean*: *FixIntersectingEdgesMode*
*Boolean*: *FixNonAdjacentIntersectingEdgesMode*

**SplitClosedEdges**

This operator handles closed edges i.e. edges with one vertex. Such edges are not supported in some receiving systems. This operator splits topologically closed edges (i.e. edges having one vertex) into two edges. Degenerated edges and edges with a size of less than Tolerance are not processed.
Messaging mechanism

Various messages about modification, warnings and fails can be generated in the process of shape fixing or upgrade. The messaging mechanism allows generating messages, which will be sent to the chosen target medium a file or the screen. The messages may report failures and/or warnings or provide information on events such as analysis, fixing or upgrade of shapes.
Message Gravity

Enumeration `Message.Gravity` is used for defining message gravity. It provides the following message statuses:

- `Message.FAIL` – the message reports a fail;
- `Message.WARNING` – the message reports a warning;
- `Message.INFO` – the message supplies information.
Tool for loading a message file into memory

Class Message_MsgFile allows defining messages by loading a custom message file into memory. It is necessary to create a custom message file before loading it into memory, as its path will be used as the argument to load it. Each message in the message file is identified by a key. The user can get the text content of the message by specifying the message key.

Format of the message file

The message file is an ASCII file, which defines a set of messages. Each line of the file must have a length of less than 255 characters. All lines in the file starting with the exclamation sign (perhaps preceded by spaces and/or tabs) are considered as comments and are ignored. A message file may contain several messages. Each message is identified by its key (string). Each line in the file starting with the dot character (perhaps preceded by spaces and/or tabs) defines the key. The key is a string starting with a symbol placed after the dot and ending with the symbol preceding the ending of the newline character \n. All lines in the file after the key and before the next keyword (and which are not comments) define the message for that key. If the message consists of several lines, the message string will contain newline symbols \n between each line (but not at the end).

The following example illustrates the structure of a message file:

```plaintext
!This is a sample message file
!-----------------------------
!Messages for ShapeAnalysis package
!
.SampleKeyword
Your message string goes here
!
!...
!
!End of the message file
```
Loading the message file

A custom file can be loaded into memory using the method `Message_MsgFile::LoadFile`, taking as an argument the path to your file as in the example below:

```
Standard_CString MsgFilePath = ;(path)/sample.file;;
Message_MsgFile::LoadFile (MsgFilePath);
```
Tool for managing filling messages

The class `Message_Msg` allows using the message file loaded as a template. This class provides a tool for preparing the message, filling it with parameters, storing and outputting to the default trace file. A message is created from a key: this key identifies the message to be created in the message file. The text of the message is taken from the loaded message file (class `Message_MsgFile` is used). The text of the message can contain places for parameters, which are to be filled by the proper values when the message is prepared. These parameters can be of the following types:

- string – coded in the text as %s,
- integer – coded in the text as %d,
- real – coded in the text as %f.

The parameter fields are filled by the message text by calling the corresponding methods `AddInteger`, `AddReal` and `AddString`. Both the original text of the message and the input text with substituted parameters are stored in the object. The prepared and filled message can be output to the default trace file. The text of the message (either original or filled) can be also obtained.

```plaintext
Message_Msg msg01 (;SampleKeyword;);
//Creates the message msg01, identified in the
//file by the keyword SampleKeyword
msg1.AddInteger (73);
msg1.AddString (;SampleFile;);
//fills out the code areas
```
Tool for managing trace files

Class `Message_TraceFile` is intended to manage the trace file (or stream) for outputting messages and the current trace level. Trace level is an integer number, which is used when messages are sent. Generally, 0 means minimum, > 0 various levels. If the current trace level is lower than the level of the message it is not output to the trace file. The trace level is to be managed and used by the users. There are two ways of using trace files:

- define an object of `Message_TraceFile`, with its own definition (file name or cout, trace level), and use it where it is defined,
- use the default trace file (file name or cout, trace level), usable from anywhere. Use the constructor method to define the target file and the level of the messages as in the example below:

```cpp
Message_TraceFile myTF
  (tracelevel, "tracefile.log",
   Standard_False);
```

The parameters are as follows:
- `tracelevel` is a Standard_Integer and modifies the level of messages. It has the following values and semantics:
  - 0: gives general information such as the start and end of process;
  - 1: gives exceptions raised and fail messages;
  - 2: gives the same information as 1 plus warning messages.
- `filename` is the string containing the path to the log file. The Boolean set to False will rewrite the existing file. When set to True, new messages will be appended to the existing file.

A new default log file can be added using method `SetDefault` with the same arguments as in the constructor. The default trace level can be changed by using method `SetDefLevel`. In this way, the information received in the log file is modified. It is possible to close the log file and set the default trace output to the screen display instead of the log file using the method `SetDefault` without any arguments.
1.8.13
# Visualization

## Table of Contents

- **Introduction**
- **Fundamental Concepts**
  - **Presentation**
    - Structure of the Presentation
    - Presentation packages
    - A Basic Example: How to display a 3D object
  - **Selection**
    - Terms and notions
    - Algorithm
    - Packages and classes
    - Examples of usage
- **Application Interactive Services**
  - **Introduction**
  - **Interactive objects**
    - Presentations
    - Hidden Line Removal
    - Presentation modes
    - Selection
    - Graphic
Overview

3D View Facilities

- Overview
- A programming example
- Define viewing parameters
- Orthographic Projection
- Perspective Projection
- Stereographic Projection
- View frustum culling
- View background styles
- Dumping a 3D scene into an image file
- Ray tracing support
- Display priorities
- Z-layer support
- Clipping planes
- Automatic back face culling

Examples: creating a 3D scene

- Create attributes
- Create a 3D Viewer (a Windows example)
- Create a 3D view (a Windows example)
- Create an
Create your own interactive object
Create primitives in the interactive object
Mesh Visualization Services
Introduction

Visualization in Open CASCADE Technology is based on the separation of:

- on the one hand – the data which stores the geometry and topology of the entities you want to display and select, and
- on the other hand – its **presentation** (what you see when an object is displayed in a scene) and **selection** (possibility to choose the whole object or its sub-parts interactively to apply application-defined operations to the selected entities).

Presentations are managed through the **Presentation** component, and selection through the **Selection** component.

**Application Interactive Services** (AIS) provides the means to create links between an application GUI viewer and the packages, which are used to manage selection and presentation, which makes management of these functionalities in 3D more intuitive and consequently, more transparent.

**AIS** uses the notion of the **Interactive Object**, a displayable and selectable entity, which represents an element from the application data. As a result, in 3D, you, the user, have no need to be familiar with any functions underlying AIS unless you want to create your own interactive objects or selection filters.

If, however, you require types of interactive objects and filters other than those provided, you will need to know the mechanics of presentable and selectable objects, specifically how to implement their virtual functions. To do this requires familiarity with such fundamental concepts as the Sensitive Primitive and the Presentable Object.

The following packages are used to display 3D objects:

- **AIS**;
- **StdPrs**;
- **Prs3d**;
- **PrsMgr**;
The packages used to display 3D objects are also applicable for visualization of 2D objects.

The figure below presents a schematic overview of the relations between the key concepts and packages in visualization. Naturally, "Geometry & Topology" is just an example of application data that can be handled by AIS, and application-specific interactive objects can deal with any kind of data.

Key concepts and packages in visualization

To answer different needs of CASCADE users, this User's Guide offers the following three paths in reading it.

- If the 3D services proposed in AIS meet your requirements, you need only read chapter 3 AIS: Application Interactive Services.
- If you need more detail, for example, a selection filter on another type of entity – you should read chapter 2 Fundamental Concepts, chapter 3 AIS: Application Interactive Services, and 4 3D Presentations. You may want to begin with the chapter presenting
AIS.

For advanced information on visualization algorithms, see our E-learning & Training offerings.
Fundamental Concepts

Presentation

In Open CASCADE Technology, presentation services are separated from the data, which they represent, which is generated by applicative algorithms. This division allows you to modify a geometric or topological algorithm and its resulting objects without modifying the visualization services.

Structure of the Presentation

Displaying an object on the screen involves three kinds of entities:

- a presentable object, the AIS_InteractiveObject
- a viewer
- an interactive context, the AIS_InteractiveContext.

The presentable object

The purpose of a presentable object is to provide the graphical representation of an object in the form of Graphic3d structure. On the first display request, it creates this structure by calling the appropriate algorithm and retaining this framework for further display.

Standard presentation algorithms are provided in the StdPrs and Prs3d packages. You can, however, write specific presentation algorithms of your own, provided that they create presentations made of structures from the Graphic3d packages. You can also create several presentations of a single presentable object: one for each visualization mode supported by your application.

Each object to be presented individually must be presentable or associated with a presentable object.

The viewer
The viewer allows interactively manipulating views of the object. When you zoom, translate or rotate a view, the viewer operates on the graphic structure created by the presentable object and not on the data model of the application. Creating Graphic3d structures in your presentation algorithms allows you to use the 3D viewers provided in Open CASCADE Technology for 3D visualisation.

**The Interactive Context**

The interactive context controls the entire presentation process from a common high-level API. When the application requests the display of an object, the interactive context requests the graphic structure from the presentable object and sends it to the viewer for displaying.

**Presentation packages**

Presentation involves at least the *AIS, PrsMgr, StdPrs* and *V3d* packages. Additional packages, such as *Prs3d* and *Graphic3d* may be used if you need to implement your own presentation algorithms.

- **AIS** package provides all classes to implement interactive objects (presentable and selectable entities).
- **PrsMgr** package provides low level services and is only to be used when you do not want to use the services provided by AIS. It contains all classes needed to implement the presentation process: abstract classes *Presentation* and *PresentableObject* and concrete class *PresentationManager3d*.
- **StdPrs** package provides ready-to-use standard presentation algorithms for specific geometries: points, curves and shapes of the geometry and topology toolkits.
- **Prs3d** package provides generic presentation algorithms such as wireframe, shading and hidden line removal associated with a *Drawer* class, which controls the attributes of the presentation to be created in terms of color, line type, thickness, etc.
- **V3d** package provides the services supported by the 3D viewer.
- **Graphic3d** package provides resources to create 3D graphic structures.
- **Visual3d** package contains classes implementing commands for 3D viewer.
- **DsgPrs** package provides tools for display of dimensions, relations
A Basic Example: How to display a 3D object

```cpp
Handle(V3d_Viewer) theViewer;
Handle(AIS_InteractiveContext) aContext = new
    AIS_InteractiveContext (theViewer);

BRepPrimAPI_MakeWedge aWedgeMaker (theWedgeDX,
    theWedgeDY, theWedgeDZ, theWedgeLtx);
TopoDS_Solid aShape = aWedgeMaker.Solid();
Handle(AIS_Shape) aShapePrs = new AIS_Shape (aShape);
    // creation of the presentable object
aContext->Display (aShapePrs); // display the
    presentable object in the 3d viewer
```

The shape is created using the `BRepPrimAPI_MakeWedge` command. An `AIS_Shape` is then created from the shape. When calling the `Display` command, the interactive context calls the Compute method of the presentable object to calculate the presentation data and transfer it to the viewer. See figure below.
Processes involved in displaying a presentable shape

Class PresentationManager3d
calls Compute() method of AIS_InteractiveObject
to obtain Graphics3d structure

calls Compute()
deals with

Deferred class
AIS_InteractiveObject
Declares Compute() method

inherits

Class AIS_Shape
Compute()
uses algorithms of Prs3d
and StdPrs packages to
create Graphics3d structure

uses

Topological Object

StdPrs and Prs3d packages
Contain various algorithms
to transform object structures
into Graphics3d structures

Creation of AIS_Shape

Object presentation

Deferred class
AIS_InteractiveObject
Declares Compute() method

Creation of topological object
Selection

Standard OCCT selection algorithm is represented by 2 parts: dynamic and static. Dynamic selection causes objects to be automatically highlighted as the mouse cursor moves over them. Static selection allows to pick particular object (or objects) for further processing.

There are 3 different selection types:

- **Point selection** – allows picking and highlighting a single object (or its part) located under the mouse cursor;
- **Rectangle selection** – allows picking objects or parts located under the rectangle defined by the start and end mouse cursor positions;
- **Polyline selection** – allows picking objects or parts located under a user-defined non-self-intersecting polyline.

For OCCT selection algorithm, all selectable objects are represented as a set of sensitive zones, called **sensitive entities**. When the mouse cursor moves in the view, the sensitive entities of each object are analyzed for collision.

Terms and notions

This section introduces basic terms and notions used throughout the algorithm description.

Sensitive entity

Sensitive entities in the same way as entity owners are links between objects and the selection mechanism.

The purpose of entities is to define what parts of the object will be selectable in particular. Thus, any object that is meant to be selectable must be split into sensitive entities (one or several). For instance, to apply face selection to an object it is necessary to explode it into faces and use them for creation of a sensitive entity set.
Example of a shape divided into sensitive entities

Depending on the user's needs, sensitive entities may be atomic (point or edge) or complex. Complex entities contain many sub-elements that can be handled by detection mechanism in a similar way (for example, a polyline stored as a set of line segments or a triangulation).

Entities are used as internal units of the selection algorithm and do not contain any topological data, hence they have a link to an upper-level interface that maintains topology-specific methods.

**Entity owner**

Each sensitive entity stores a reference to its owner, which is a class connecting the entity and the corresponding selectable object. Besides, owners can store any additional information, for example, the topological shape of the sensitive entity, highlight colors and methods, or if the entity is selected or not.

**Selection**

To simplify the handling of different selection modes of an object, sensitive entities linked to their owners are organized into sets, called *selections*. Each selection contains entities created for a certain mode along with the sensitivity and update states.

**Selectable object**

Selectable object stores information about all created selection modes
and sensitive entities.

All successors of a selectable object must implement the method that splits its presentation into sensitive entities according to the given mode. The computed entities are arranged in one selection and added to the list of all selections of this object. No selection will be removed from the list until the object is deleted permanently.

For all standard OCCT shapes, zero mode is supposed to select the whole object (but it may be redefined easily in the custom object). For example, the standard OCCT selection mechanism and \textit{AIS\_Shape} determine the following modes:

- 0 – selection of entire object (AIS\_Shape);
- 1 – selection of the vertices;
- 2 – selection of the edges;
- 3 – selection of the wires;
- 4 – selection of the faces;
- 5 – selection of the shells;
- 6 – selection of the constituent solids.

\textbf{Hierarchy of references from sensitive entity to selectable object}

\textbf{The principle of entities organization within the selectable object}
Viewer selector

For each OCCT viewer there is a **Viewer selector** class instance. It provides a high-level API for the whole selection algorithm and encapsulates the processing of objects and sensitive entities for each mouse pick. The viewer selector maintains activation and deactivation of selection modes, launches the algorithm, which detects candidate entities to be picked, and stores its results, as well as implements an interface for keeping selection structures up-to-date.

Selection manager

Selection manager is a high-level API to manipulate selection of all displayed objects. It handles all viewer selectors, activates and deactivates selection modes for the objects in all or particular selectors, manages computation and update of selections for each object. Moreover, it keeps selection structures updated taking into account applied changes.

![Diagram of the relations chain between viewer selector and selection manager]

**The relations chain between viewer selector and selection manager**

Algorithm

All three types of OCCT selection are implemented as a single concept, based on the search for overlap between frustum and sensitive entity through 3-level BVH tree traversal.

Selection Frustum
The first step of each run of selection algorithm is to build the selection frustum according to the currently activated selection type.

For the point or the rectangular selection the base of the frustum is a rectangle built in conformity with the pixel tolerance or the dimensions of a user-defined area, respectively. For the polyline selection, the polygon defined by the constructed line is triangulated and each triangle is used as the base for its own frustum. Thus, this type of selection uses a set of triangular frustums for overlap detection.

The frustum length is limited by near and far view volume planes and each plane is built parallel to the corresponding view volume plane.

The image above shows the rectangular frustum: a) after mouse move or click, b) after applying the rectangular selection.
In the image above triangular frustum is set: a) by a user-defined polyline, b) by triangulation of the polygon based on the given polyline, c) by a triangular frustum based on one of the triangles.

**BVH trees**

To maintain selection mechanism at the viewer level, a speedup structure composed of 3 BVH trees is used.

The first level tree is constructed of axis-aligned bounding boxes of each selectable object. Hence, the root of this tree contains the combination of all selectable boundaries even if they have no currently activated selections. Objects are added during the display of `AIS_InteractiveObject` and will be removed from this tree only when the object is destroyed. The 1st level BVH tree is build on demand simultaneously with the first run of the selection algorithm.

The second level BVH tree consists of all sensitive entities of one selectable object. The 2nd level trees are built automatically when the default mode is activated and rebuilt whenever a new selection mode is calculated for the first time.

The third level BVH tree is used for complex sensitive entities that contain many elements: for example, triangulations, wires with many segments, point sets, etc. It is built on demand for sensitive entities with under 800K sub-elements.
Selection BVH tree hierarchy: from the biggest object-level (first) to the smallest complex entity level (third)

Stages of the algorithm

The algorithm includes pre-processing and three main stages.

Pre-processing

Implies calculation of the selection frustum and its main characteristics.

First stage – traverse of the first level BVH tree

After successful building of the selection frustum, the algorithm starts traversal of the object-level BVH tree. The nodes containing axis-aligned bounding boxes are tested for overlap with the selection frustum following the terms of separating axis theorem (SAT). When the traversal goes down to the leaf node, it means that a candidate object with possibly overlapping sensitive entities has been found. If no such objects have been detected, the algorithm stops and it is assumed that no object
needs to be selected. Otherwise it passes to the next stage to process the entities of the found selectable object.

**Second stage – traversal of the second level BVH tree**

At this stage it is necessary to determine if there are candidates among all sensitive entities of one object.

First of all, at this stage the algorithm checks if there is any transformation applied for the current object. If it has its own location, then the correspondingly transformed frustum will be used for further calculations. At the next step the nodes of the second level BVH tree of the given object are visited to search for overlapping leaves. If no such leafs have been found, the algorithm returns to the second stage. Otherwise it starts processing the found entities by performing the following checks:

- activation check - the entity may be inactive at the moment as it belongs to deactivated selection;
- tolerance check - current selection frustum may be too large for further checks as it is always built with the maximum tolerance among all activated entities. Thus, at this step the frustum may be scaled.

After these checks the algorithm passes to the last stage.

**Third stage – overlap or inclusion test of a particular sensitive entity**

If the entity is atomic, a simple SAT test is performed. In case of a complex entity, the third level BVH tree is traversed. The quantitative characteristics (like depth, distance to the center of geometry) of matched sensitive entities is analyzed and clipping planes are applied (if they have been set). The result of detection is stored and the algorithm returns to the second stage.

**Packages and classes**

Selection is implemented as a combination of various algorithms divided among several packages – *SelectBasics, Select3D, SelectMgr* and *StdSelect*. 
SelectBasics

*SelectBasics* package contains basic classes and interfaces for selection. The most notable are:

- *SelectBasics_SensitiveEntity* – the base definition of a sensitive entity;
- *SelectBasics_EntityOwner* – the base definition of the entity owner – the link between the sensitive entity and the object to be selected;
- *SelectBasics_PickResult* – the structure for storing quantitative results of detection procedure, for example, depth and distance to the center of geometry;
- *SelectBasics_SelectingVolumeManager* – the interface for interaction with the current selection frustum.

Each custom sensitive entity must inherit at least *SelectBasics_SensitiveEntity*.

Select3D

*Select3D* package provides a definition of standard sensitive entities, such as:

- box;
- circle;
- curve;
- face;
- group;
- point;
- segment;
- triangle;
- triangulation;
- wire.

Each basic sensitive entity inherits *Select3D_SensitiveEntity*, which is a child class of *SelectBasics_SensitiveEntity*. The package also contains two auxiliary classes, *Select3D_SensitivePoly* and *Select3D_SensitiveSet*. 
Select3D_SensitiveSet – a base class for all complex sensitive entities that require the third level BVH usage. It implements traverse of the tree and defines an interface for the methods that check sub-entities.

Select3D_SensitivePoly – describes an arbitrary point set and implements basic functions for selection. It is important to know that this class does not perform any internal data checks. Hence, custom implementations of sensitive entity inherited from Select3D_SensitivePoly must satisfy the terms of Separating Axis Theorem to use standard OCCT overlap detection methods.

SelectMgr

SelectMgr package is used to maintain the whole selection process. For this purpose, the package provides the following services:

- activation and deactivation of selection modes for all selectable objects;
- interfaces to compute selection mode of the object;
- definition of selection filter classes;
- keeping selection BVH data up-to-date.

A brief description of the main classes:

- SelectMgr_FrustumBase, SelectMgr_Frustum, SelectMgr_RectangularFrustum, SelectMgr_TriangularFrustum and SelectMgr_TriangularFrustumSet – interfaces and implementations of selecting frustums, these classes implement different SAT tests for overlap and inclusion detection. They also contain methods to measure characteristics of detected entities (depth, distance to center of geometry);
- SelectMgr_SensitiveEntity, SelectMgr_Selection and SelectMgr_SensitiveEntitySet – store and handle sensitive entities; SelectMgr_SensitiveEntitySet implements a primitive set for the second level BVH tree;
- SelectMgr_SelectableObject and SelectMgr_SelectableObjectSet – describe selectable objects. They also manage storage, calculation and removal of selections. SelectMgr_SelectableObjectSet implements a primitive set for the first level BVH tree;
- SelectMgr_ViewerSelector – encapsulates all logics of the selection
algorithm and implements the third level BVH tree traverse;

- \textit{SelectMgr\_SelectionManager} – manages activation/deactivation, calculation and update of selections of every selectable object, and keeps BVH data up-to-date.

\textbf{StdSelect}

\textit{StdSelect} package contains the implementation of some \textit{SelectMgr} classes and tools for creation of selection structures. For example,

- \textit{StdSelect\_BRepOwner} – defines an entity owner with a link to its topological shape and methods for highlighting;
- \textit{StdSelect\_BRepSelectionTool} – contains algorithms for splitting standard AIS shapes into sensitive primitives;
- \textit{StdSelect\_ViewerSelector3d} – an example of \textit{SelectMgr\_ViewerSelector} implementation, which is used in a default OCCT selection mechanism;
- \textit{StdSelect\_FaceFilter}, \textit{StdSelect\_EdgeFilter} – implementation of selection filters.

\textbf{Examples of usage}

The first code snippet illustrates the implementation of \textit{SelectMgr\_SelectableObject::ComputeSelection()} method in a custom interactive object. The method is used for computation of user-defined selection modes. Let us assume it is required to make a box selectable in two modes – the whole shape (mode 0) and each of its edges (mode 1).

To select the whole box, the application can create a sensitive primitive for each face of the interactive object. In this case, all primitives share the same owner – the box itself. To select box's edge, the application must create one sensitive primitive per edge. Here all sensitive entities cannot share the owner since different geometric primitives must be highlighted as the result of selection procedure.

```cpp
void InteractiveBox::ComputeSelection (const Handle(SelectMgr_Selection)& theSel, const Standard_Integer theMode)
{
```
switch (theMode)
{
    case 0:  // creation of face sensitives for
             // selection of the whole box
    {
        Handle(SelectMgr_EntityOwner) anOwner = new
            SelectMgr_EntityOwner (this, 5);
        for (Standard_Integer aFaceIter = 1; aFaceIter <= myNbFaces; ++aFaceIter)
        {
            Select3D_TypeOfSensitivity aSensType =
                myIsInterior;
            theSel->Add (new Select3D_SensitiveFace
                             (anOwner, myFaces[aFaceIter]->PointArray(),
                              aSensType));
        }
        break;
    }
    case 1: // creation of edge sensitives for
             // selection of box edges only
    {
        for (Standard_Integer anEdgeIter = 1;
             anEdgeIter <= 12; ++anEdgeIter)
        {
            // 1 owner per edge, where 6 is a priority of
            // the sensitive
            Handle(MySelection_EdgeOwner) anOwner = new
                MySelection_EdgeOwner (this, anEdgeIter, 6);
            theSel->Add (new Select3D_SensitiveSegment
                           (anOwner, myFirstPnt[anEdgeIter]),
                           myLastPnt[anEdgeIter]));
        }
        break;
    }
}
The algorithms for creating selection structures store sensitive primitives in \textit{SelectMgr_Selection} instance. Each \textit{SelectMgr_Selection} sequence in the list of selections of the object must correspond to a particular selection mode. To describe the decomposition of the object into selectable primitives, a set of ready-made sensitive entities is supplied in \textit{Select3D} package. Custom sensitive primitives can be defined through inheritance from \textit{SelectBasics_SensitiveEntity}. To make custom interactive objects selectable or customize selection modes of existing objects, the entity owners must be defined. They must inherit \textit{SelectMgr_EntityOwner} interface.

Selection structures for any interactive object are created in \textit{SelectMgrSelectableObject::ComputeSelection()} method. The example below shows how computation of different selection modes of the topological shape can be done using standard OCCT mechanisms, implemented in \textit{StdSelect_BRepSelectionTool}.

```cpp
void MyInteractiveObject::ComputeSelection (const Handle(SelectMgr_Selection)& theSelection, const Standard_Integer theMode)
{
    switch (theMode)
    {
    case 0: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_SHAPE); break;
    case 1: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_VERTEX); break;
    case 2: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_EDGE); break;
    case 3: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_WIRE); break;
    case 4: StdSelect_BRepSelectionTool::Load (theSelection, this, myShape, TopAbs_FACE); break;
    ```
The `StdSelect_BRepSelectionTool` class provides a high level API for computing sensitive entities of the given type (for example, face, vertex, edge, wire and others) using topological data from the given `TopoDS_Shape`.

The traditional way of highlighting selected entity owners adopted by Open CASCADE Technology assumes that each entity owner highlights itself on its own. This approach has two drawbacks:

- each entity owner has to maintain its own `Prs3d_Presentation` object, that results in a considerable memory overhead;
- drawing selected owners one by one is not efficient from the visualization point of view.

Therefore, to overcome these limitations, OCCT has an alternative way to implement the highlighting of a selected presentation. Using this approach, the interactive object itself will be responsible for the highlighting, not the entity owner.

On the basis of `SelectMgr_EntityOwner::IsAutoHilight()` return value, `AIS_InteractiveContext` object either uses the traditional way of highlighting (in case if `IsAutoHilight()` returns TRUE) or groups such owners according to their selectable objects and finally calls `SelectMgr_SelectableObject::HilightSelected()` or `SelectMgr_SelectableObject::ClearSelected()`, passing a group of owners as an argument.

Hence, an application can derive its own interactive object and redefine virtual methods `HilightSelected()`, `ClearSelected()` and `HilightOwnerWithColor()` from `SelectMgr_SelectableObject`. `SelectMgr_SelectableObject::GetHilightPresentation` and `SelectMgr_SelectableObject::GetSelectPresentation` methods can be used to optimize filling of selection and highlight presentations according to the user's needs. The `AIS_InteractiveContext::HighlightSelected()` method can be used for efficient redrawing of the selection presentation for a given interactive object from an application code.

After all the necessary sensitive entities are computed and packed in
SelectMgr_Selection instance with the corresponding owners in a redefinition of SelectMgr_SelectableObject::ComputeSelection() method, it is necessary to register the prepared selection in SelectMgr_SelectionManager through the following steps:

- if there was no AIS_InteractiveContext opened, create an interactive context and display the selectable object in it;
- load the selectable object to the selection manager of the interactive context using AIS_InteractiveContext::Load() method. If the selection mode passed as a parameter to this method is not equal to -1, ComputeSelection() for this selection mode will be called;
- activate or deactivate the defined selection mode using AIS_InteractiveContext::Activate() or AIS_InteractiveContext::Deactivate() methods.

After these steps, the selection manager of the created interactive context will contain the given object and its selection entities, and they will be involved in the detection procedure.

The code snippet below illustrates the above steps. It also contains the code to start the detection procedure and parse the results of selection.

```cpp
// Suppose there is an instance of class InteractiveBox from the previous sample.
// It contains an implementation of method InteractiveBox::ComputeSelection() for selection
// modes 0 (whole box must be selected) and 1 (edge of the box must be selectable)
Handle(InteractiveBox) theBox;
Handle(AIS_InteractiveContext) theContext;
// To prevent automatic activation of the default selection mode
theContext->SetAutoActivateSelection (false);
theContext->Display (theBox, false);

// Load a box to the selection manager without computation of any selection mode
theContext->Load (theBox, -1, true);
// Activate edge selection
```
theContext->Activate (theBox, 1);

// Run the detection mechanism for activated entities in the current mouse coordinates and in the current view.
// Detected owners will be highlighted with context highlight color
theContext->MoveTo (aXMousePos, aYMousePos, myView);
// Select the detected owners
theContext->Select();
// Iterate through the selected owners
for (theContext->InitSelected(); theContext->MoreSelected() && !aHasSelected; theContext->NextSelected())
{
    Handle(AIS_InteractiveObject) anIO = theContext->SelectedInteractive();
}

// deactivate all selection modes for aBox1
theContext->Deactivate (aBox1);

It is also important to know, that there are 2 types of detection implemented for rectangular selection in OCCT:

- **inclusive** detection. In this case the sensitive primitive is considered detected only when all its points are included in the area defined by the selection rectangle;
- **overlap** detection. In this case the sensitive primitive is considered detected when it is partially overlapped by the selection rectangle.

The standard OCCT selection mechanism uses inclusion detection by default. To change this, use the following code:

// Assume there is a created interactive context
const Handle(AIS_InteractiveContext) theContext;
// Retrieve the current viewer selector
const Handle(StdSelect_ViewerSelector3d)&
aMainSelector = theContext->MainSelector();
// Set the flag to allow overlap detection
aMainSelector->AllowOverlapDetection (true);
Application Interactive Services

Introduction

Application Interactive Services allow managing presentations and dynamic selection in a viewer in a simple and transparent manner. The central entity for management of visualization and selections is the **Interactive Context**. It is connected to the main viewer.

Interactive context by default starts at **Neutral Point** with each selectable object picked as a whole, but the user might activate **Local Selection** for specific objects to make selectable parts of the objects. Local/global selection is managed by a list of selection modes activated for each displayed object with 0 (default selection mode) usually meaning Global (entire object) selection.

**Interactive Objects** are the entities, which are visualized and selected. You can use classes of standard interactive objects for which all necessary functions have already been programmed, or you can implement your own classes of interactive objects, by respecting a certain number of rules and conventions described below.

An Interactive Object is a "virtual" entity, which can be presented and selected. An Interactive Object can have a certain number of specific graphic attributes, such as visualization mode, color and material. When an Interactive Object is visualized, the required graphic attributes are taken from its own **Drawer** (*Prs3d_Drawer*) if it has the required custom attributes or otherwise from the context drawer.
It can be necessary to filter the entities to be selected. Consequently there are Filter entities, which allow refining the dynamic detection context. Some of these filters can be used only within at the Neutral Point, others only within Local Selection. It is possible to program custom filters and load them into the interactive context.
Interactive objects

Entities which are visualized and selected in the AIS viewer are objects. They connect the underlying reference geometry of a model to its graphic representation in AIS. You can use the predefined OCCT classes of standard interactive objects, for which all necessary functions have already been programmed, or, if you are an advanced user, you can implement your own classes of interactive objects.

Presentations

An interactive object can have as many presentations as its creator wants to give it. 3D presentations are managed by Presentation Manager (`PrsMgr_PresentationManager`). As this is transparent in AIS, the user does not have to worry about it.

A presentation is identified by an index (Display Mode) and by the reference to the Presentation Manager, which it depends on. By convention, the default mode of representation for the Interactive Object has index 0.

Calculation of different presentations of an interactive object is done by the `Compute` functions inheriting from `PrsMgr_PresentableObject::Compute` functions. They are automatically called by `PresentationManager` at a visualization or an update request.

If you are creating your own type of interactive object, you must implement the Compute function in one of the following ways:

For 3D:
void PackageName_ClassName::Compute (const Handle(PrsMgr_PresentationManager3d)& thePresentationManager,
const Handle(Prs3d_Presentation)& thePresentation,
const Standard_Integer theMode);

#### For hidden line removal (HLR) mode in 3D:

void PackageName_ClassName::Compute (const Handle(Prs3d_Projector)& theProjector,
const Handle(Prs3d_Presentation)& thePresentation);

Hidden Line Removal

The view can have two states: the normal mode or the computed mode (Hidden Line Removal mode). When the latter is active, the view looks for all presentations displayed in the normal mode, which have been signalled as accepting HLR mode. An internal mechanism allows calling the interactive object’s own `Compute`, that is projector function.

By convention, the Interactive Object accepts or rejects the representation of HLR mode. It is possible to make this declaration in one of two ways:

- Initially by using one of the values of the enumeration `PrsMgr_TypeOfPresentation`:
  - `PrsMgr_TOP_AllView`,
  - `PrsMgr_TOP_ProjectorDependant`
- Later by using the function `PrsMgr_PresentableObject::SetTypeOfPresentation`

`AIS_Shape` class is an example of an interactive object that supports HLR representation. The type of the HLR algorithm is stored in `Prs3d_Drawer` of the shape. It is a value of the `Prs3d_TypeOfHLR` enumeration and can be set to:

- `Prs3d_TOH_PolyAlgo` for a polygonal algorithm based on the
shape's triangulation;
- \textit{Prs3d\_TOH\_Algo} for an exact algorithm that works with the shape's real geometry;
- \textit{Prs3d\_TOH\_NotSet} if the type of algorithm is not set for the given interactive object instance.

The type of the HLR algorithm used for \textit{AIS\_Shape} can be changed by calling the \textit{AIS\_Shape::SetTypeOfHLR()} method. The current HLR algorithm type can be obtained using \textit{AIS\_Shape::TypeOfHLR()} method is to be used.

These methods get the value from the drawer of \textit{AIS\_Shape}. If the HLR algorithm type in the \textit{AIS\_Drawer} is set to \textit{Prs3d\_TOH\_NotSet}, the \textit{AIS\_Drawer} gets the value from the default drawer of \textit{AIS\_InteractiveContext}. So it is possible to change the default HLR algorithm used by all newly displayed interactive objects. The value of the HLR algorithm type stored in the context drawer can be \textit{Prs3d\_TOH\_Algo} or \textit{Prs3d\_TOH\_PolyAlgo}. The polygonal algorithm is the default one.

\textbf{Presentation modes}

There are four types of interactive objects in AIS:

- the "construction element" or Datum,
- the Relation (dimensions and constraints)
- the Object
- the None type (when the object is of an unknown type).

Inside these categories, additional characterization is available by means of a signature (an index.) By default, the interactive object has a NONE type and a signature of 0 (equivalent to NONE). If you want to give a particular type and signature to your interactive object, you must redefine two virtual functions:

- \textit{AIS\_InteractiveObject::Type}
- \textit{AIS\_InteractiveObject::Signature}.

\textbf{Note} that some signatures are already used by "standard" objects provided in AIS (see the \textbf{List of Standard Interactive Object Classes}).
The interactive context can have a default mode of representation for the set of interactive objects. This mode may not be accepted by a given class of objects. Consequently, to get information about this class it is necessary to use virtual function

\textit{AIS\_InteractiveObject::AcceptDisplayMode}.

\textbf{Display Mode}

The functions \textit{AIS\_InteractiveContext::SetDisplayMode} and \textit{AIS\_InteractiveContext::UnsetDisplayMode} allow setting a custom display mode for an objects, which can be different from that proposed by the interactive context.

\textbf{Highlight Mode}

At dynamic detection, the presentation echoed by the Interactive Context, is by default the presentation already on the screen.

The functions \textit{AIS\_InteractiveObject::SetHilightMode} and \textit{AIS\_InteractiveObject::UnSetHilightMode} allow specifying the display mode used for highlighting (so called highlight mode), which is valid independently from the active representation of the object. It makes no difference whether this choice is temporary or definitive.

Note that the same presentation (and consequently the same highlight mode) is used for highlighting \textit{detected} objects and for highlighting \textit{selected} objects, the latter being drawn with a special selection color (refer to the section related to Interactive Context services).

For example, you want to systematically highlight the wireframe presentation of a shape - non regarding if it is visualized in wireframe presentation or with shading. Thus, you set the highlight mode to \textit{0} in the constructor of the interactive object. Do not forget to implement this representation mode in the \textit{Compute} functions.

\textbf{Infinite Status}

If you do not want an object to be affected by a \textit{FitAll} view, you must declare it infinite; you can cancel its "infinite" status using
AIS InteractiveObject::SetInfiniteState and AIS InteractiveObject::IsInfinite functions.

Let us take for example the class called IShape representing an interactive object:

```cpp
myPk_IShape::myPK_IShape (const TopoDS_Shape& theShape, PrsMgr_TypeOfPresentation theType) :
AIS_InteractiveObject (theType), myShape (theShape)
{
SetHilightMode (0);
}

void myPk_IShape::Compute (const
  Handle(PrsMgr_PresentationManager3d)& thePrsMgr,
  const
  Handle(Prs3d_Presentation)& thePrs,
  const Standard_Integer theMode)
{
  switch (theMode)
  {
    // algo for calculation of wireframe presentation
    case 0: StdPrs_WFDeflectionShape::Add (thePrs, myShape, myDrawer); return;
    // algo for calculation of shading presentation
    case 1: StdPrs_ShadedShape::Add (thePrs, myShape, myDrawer); return;
  }
}

void myPk_IShape::Compute (const
  Handle(Prs3d_Projector)& theProjector,
  const
  Handle(Prs3d_Presentation)& thePrs)
{
  // Hidden line mode calculation algorithm
  StdPrs_HLRPolyShape::Add (thePrs, myShape, myDrawer, theProjector);
}
Selection

An interactive object can have an indefinite number of selection modes, each representing a "decomposition" into sensitive primitives. Each primitive has an **Owner** (*SelectMgr_EntityOwner*) which allows identifying the exact interactive object or shape which has been detected (see Selection chapter).

The set of sensitive primitives, which correspond to a given mode, is stocked in a **Selection** (*SelectMgr_Selection*).

Each selection mode is identified by an index. By convention, the default selection mode that allows us to grasp the interactive object in its entirety is mode 0. However, it can be modified in the custom interactive objects using method *SelectMgr_SelectableObject::setGlobalSelMode()*.

The calculation of selection primitives (or sensitive entities) is done in a virtual function *ComputeSelection*. It should be implemented for each type of interactive object that is assumed to have different selection modes using the function *AIS_InteractiveObject::ComputeSelection*. A detailed explanation of the mechanism and the manner of implementing this function has been given in Selection chapter.

There are some examples of selection mode calculation for the most widely used interactive object in OCCT – *AIS_Shape* (selection by vertex, by edges, etc). To create new classes of interactive objects with the same selection behavior as *AIS_Shape* – such as vertices and edges – you must redefine the virtual function *AIS_InteractiveObject::AcceptShapeDecomposition*.

Graphic attributes

Graphic attributes manager, or *Prs3d_Drawer*, stores graphic attributes for specific interactive objects and for interactive objects controlled by interactive context.

Initially, all drawer attributes are filled out with the predefined values which will define the default 3D object appearance. When an interactive object is visualized, the required graphic attributes are first taken from its own drawer if one exists, or from the context drawer if no specific drawer
for that type of object exists.

Keep in mind the following points concerning graphic attributes:

- Each interactive object can have its own visualization attributes.
- By default, the interactive object takes the graphic attributes of the context in which it is visualized (visualization mode, deflection values for the calculation of presentations, number of isoparameters, color, type of line, material, etc.)
- In the `AIS_InteractiveObject` abstract class, standard attributes including color, line thickness, material, and transparency have been privileged. Consequently, there is a certain number of virtual functions, which allow acting on these attributes. Each new class of interactive object can redefine these functions and change the behavior of the class.

![Diagram showing the redefinition of virtual functions for changes in AIS_Shape and AIS_TextLabel.]

**Redefinition of virtual functions for changes in AIS_Shape and AIS_TextLabel.**

The following virtual functions provide settings for color, width, material and transparency:

- `AIS_InteractiveObject::UnsetColor`
- `AIS_InteractiveObject::SetWidth`
- `AIS_InteractiveObject::UnsetWidth`
- `AIS_InteractiveObject::SetMaterial`
- `AIS_InteractiveObject::UnsetMaterial`
- *AIS_InteractiveObject::SetTransparency*
- *AIS_InteractiveObject::UnsetTransparency*

These methods can be used as a shortcut assigning properties in common way, but result might be not available. Some interactive objects might not implement these methods at all or implement only a sub-set of them. Direct modification of *Prs3d_Drawer* properties returned by *AIS_InteractiveObject::Attributes* can be used for more precise and predictable configuration.

It is important to know which functions may imply the recalculation of presentations of the object. If the presentation mode of an interactive object is to be updated, a flag from *PrsMgr_PresentableObject* indicates this. The mode can be updated using the functions *Display* and *Redisplay* in *AIS_InteractiveContext*.

**Complementary Services**

When you use complementary services for interactive objects, pay special attention to the cases mentioned below.

**Change the location of an interactive object**

The following functions allow "moving" the representation and selection of Interactive Objects in a view without recalculation (and modification of the original shape).

- *AIS_InteractiveContext::SetLocation*
- *AIS_InteractiveContext::ResetLocation*
- *AIS_InteractiveContext::HasLocation*
- *AIS_InteractiveContext::Location*

**Connect an interactive object to an applicative entity**

Each Interactive Object has functions that allow attributing it an *Owner* in form of a *Transient*.

- *AIS_InteractiveObject::SetOwner*
- *AIS_InteractiveObject::HasOwner*
- *AIS_InteractiveObject::Owner*
An interactive object can therefore be associated or not with an applicative entity, without affecting its behavior.

**NOTE:** Don't be confused by owners of another kind - *SelectBasics_EntityOwner* used for identifying selectable parts of the object or object itself.

**Resolving coincident topology**

Due to the fact that the accuracy of three-dimensional graphics coordinates has a finite resolution the elements of topological objects can coincide producing the effect of "popping" some elements one over another.

To the problem when the elements of two or more Interactive Objects are coincident you can apply the polygon offset. It is a sort of graphics computational offset, or depth buffer offset, that allows you to arrange elements (by modifying their depth value) without changing their coordinates. The graphical elements that accept this kind of offsets are solid polygons or displayed as boundary lines and points. The polygons could be displayed as lines or points by setting the appropriate interior style.

The methods *AIS_InteractiveObject::SetPolygonOffsets* and *AIS_InteractiveContext::SetPolygonOffsets* allow setting up the polygon offsets.

**Object hierarchy**

Each *PrsMgr_PresentableObject* has a list of objects called *myChildren*. Any transformation of *PrsMgr_PresentableObject* is also applied to its children. This hierarchy does not propagate to *Graphic3d* level and below.

*PrsMgr_PresentableObject* sends its combined (according to the hierarchy) transformation down to *Graphic3d_Structure*. The materials of structures are not affected by the hierarchy.

Object hierarchy can be controlled by the following API calls:
• PrsMgr_PresentableObject::AddChild;
• PrsMgr_PresentableObject::RemoveChild.

Instancing

The conception of instancing operates the object hierarchy as follows:

• Instances are represented by separated AIS objects.
• Instances do not compute any presentations.

Classes AIS_ConnectedInteractive and AIS_MultipleConnectedInteractive are used to implement this conception.

AIS_ConnectedInteractive is an object instance, which reuses the geometry of the connected object but has its own transformation, material, visibility flag, etc. This connection is propagated down to OpenGl level, namely to OpenGl_Structure. OpenGl_Structure can be connected only to a single other structure.

AIS_ConnectedInteractive can be referenced to any AIS_InteractiveObject in general. When it is referenced to another AIS_ConnectedInteractive, it just copies the reference.

AIS_MultipleConnectedInteractive represents an assembly, which does not have its own presentation. The assemblies are able to participate in the object hierarchy and are intended to handle a grouped set of instanced objects. It behaves as a single object in terms of selection. It applies high level transformation to all sub-elements since it is located above in the hierarchy.

All AIS_MultipleConnectedInteractive are able to have child assemblies. Deep copy of object instances tree is performed if one assembly is attached to another.

Note that AIS_ConnectedInteractive cannot reference AIS_MultipleConnectedInteractive. AIS_ConnectedInteractive copies sensitive entities of the origin object for selection, unlike AIS_MultipleConnectedInteractive that re-uses the entities of the origin object.

Instances can be controlled by the following DRAW commands:
• **vconnect**: Creates and displays *AIS_MultipleConnectedInteractive* object from input objects and location.
• **vconnectto**: Makes an instance of object with the given position.
• **vdisconnect**: Disconnects all objects from an assembly or disconnects an object by name or number.
• **vaddconnected**: Adds an object to the assembly.
• **vlistconnected**: Lists objects in the assembly.

Have a look at the examples below:

```plaintext
pload ALL
vinit
psphere s 1
vdisplay s
vconnectto s2 3 0 0 s # make instance
vfit
```

See how proxy *OpenGL_Structure* is used to represent instance:
The original object does not have to be displayed in order to make instance. Also selection handles transformations of instances correctly:

```plaintext
pload ALL
vinit
psphere s 1
psphere p 0.5
vdisplay s  # p is not displayed
vsetloc s -2 0 0
vconnect x 3 0 0 s p  # make assembly
vfit
```

Here is the example of a more complex hierarchy involving sub-assemblies:

```plaintext
pload ALL
vinit
box b 1 1 1
psphere s 0.5
vdisplay b s
vsetlocation s 0 2.5 0
box d 0.5 0.5 3
```
box d2 0.5 3 0.5
vdisplay d d2
vconnectto b1 -2 0 0 b
vconnect z 2 0 0 b s
vconnect z2 4 0 0 d d2
vconnect z3 6 0 0 z z2
vfit
Interactive Context

Rules

The Interactive Context allows managing in a transparent way the graphic and selectable behavior of interactive objects in one or more viewers. Most functions which allow modifying the attributes of interactive objects, and which were presented in the preceding chapter, will be looked at again here.

There is one essential rule to follow: the modification of an interactive object, which is already known by the Context, must be done using Context functions. You can only directly call the functions available for an interactive object if it has not been directly loaded into an Interactive Context.

Handle(AIS_Shape) aShapePrs = new AIS_Shape
  (theShape);
myIntContext->Display (aShapePrs, AIS_Shaded, 0,
  false, aShapePrs->AcceptShapeDecomposition());
myIntContext->SetColor(aShapePrs, Quantity_NOC_RED);

You can also write

Handle(AIS_Shape) aShapePrs = new AIS_Shape
  (theShape);
aShapePrs->SetColor (Quantity_NOC_RED);
aShapePrs->SetDisplayMode (AIS_Shaded);
myIntContext->Display (aShapePrs);

Groups of functions

Neutral Point and Local Selection constitute the two operating modes or states of the Interactive Context, which is the central entity which pilots visualizations and selections. The Neutral Point, which is the default mode, allows easily visualizing and selecting interactive objects, which have been loaded into the context. Activating Local Selection for specific Objects allows selecting of their sub-parts.
Management of the Interactive Context

An interactive object can have a certain number of specific graphic attributes, such as visualization mode, color, and material. Correspondingly, the interactive context has a set of graphic attributes, the Drawer, which is valid by default for the objects it controls. When an interactive object is visualized, the required graphic attributes are first taken from the object’s own Drawer if it exists, or from the context drawer if otherwise.

The following adjustable settings allow personalizing the behavior of presentations and selections:

- Default Drawer, containing all the color and line attributes which can be used by interactive objects, which do not have their own attributes.
- Default Visualization Mode for interactive objects. By default: mode 0;
- Highlight color of entities detected by mouse movement. By default: Quantity_NOC_CYAN1;
- Pre-selection color. By default: Quantity_NOC_GREEN;
- Selection color (when you click on a detected object). By default: Quantity_NOC_GRAY80;

All of these settings can be modified by functions proper to the Context. When you change a graphic attribute pertaining to the Context (visualization mode, for example), all interactive objects, which do not have the corresponding appropriate attribute, are updated.

Let us examine the case of two interactive objects: theObj1 and theObj2:

```c
theCtx->Display (theObj1, false);
theCtx->Display (theObj2, true);  // TRUE for viewer update
theCtx->SetDisplayMode (theObj1, 3, false);
theCtx->SetDisplayMode (2, true);
// theObj2 is visualised in mode 2 (if it accepts this mode)
// theObj1 stays visualised in its mode 3
```
*PresentationManager* and *Selector3D*, which manage the presentation and selection of present interactive objects, are associated to the main Viewer.
Local Selection

Selection Modes

The Local Selection is defined by index (Selection Mode). The Selection Modes implemented by a specific interactive object and their meaning should be checked within the documentation of this class. See, for example, MeshVS_SelectionModeFlags for MeshVS_Mesh object.

The interactive object, which is used the most by applications, is AIS_Shape. Consequently, there are standard functions, which allow you to easily prepare selection operations on the constituent elements of shapes (selection of vertices, edges, faces, etc.). The Selection Mode for a specific shape type (TopAbs_ShapeEnum) is returned by method AIS_Shape::SelectionMode().

The function AIS_InteractiveObject::Display (without argument taking Selection Mode) activates the object’s default Selection Mode. The functions AIS_InteractiveContext::Activate and AIS_InteractiveContext::Deactivate activate and deactivate specific Selection Mode.

More than one Selection Mode can be activated at the same time (but default 0 mode for selecting entire object is exclusive - it cannot be combined with others). The list of active modes can be retrieved using function AIS_InteractiveContext::ActivatedModes.

Filters

To define an environment of dynamic detection, you can use standard filter classes or create your own. A filter questions the owner of the sensitive primitive to determine if it has the desired qualities. If it answers positively, it is kept. If not, it is rejected.

The root class of objects is SelectMgr_Filter. The principle behind it is straightforward: a filter tests to see whether the owners (SelectMgr_EntityOwner) detected in mouse position by selector answer OK. If so, it is kept, otherwise it is rejected. You can create a custom class of filter objects by implementing the deferred function
SelectMgr_Filter::IsOk().

In SelectMgr, there are also Composition filters (AND Filters, OR Filters), which allow combining several filters. In Interactive Context, all filters that you add are stored in an OR filter (which answers OK if at least one filter answers OK).

There are Standard filters, which have already been implemented in several packages:

- **StdSelect_EdgeFilter** – for edges, such as lines and circles;
- **StdSelect_FaceFilter** – for faces, such as planes, cylinders and spheres;
- **StdSelect_ShapeTypeFilter** – for shape types, such as compounds, solids, shells and wires;
- **AIS_TypeFilter** – for types of interactive objects;
- **AIS_SignatureFilter** – for types and signatures of interactive objects;
- **AIS_AttributeFilter** – for attributes of Interactive Objects, such as color and width.

There are several functions to manipulate filters:

- **AIS_InteractiveContext::AddFilter** adds a filter passed as an argument.
- **AIS_InteractiveContext::RemoveFilter** removes a filter passed as an argument.
- **AIS_InteractiveContext::RemoveFilters** removes all present filters.
- **AIS_InteractiveContext::Filters** gets the list of filters active in a context.

**Example**

```plaintext
// shading visualization mode, no specific mode, authorization for decomposition into sub-shapes
const TopoDS_Shape theShape;
Handle(AIS_Shape) aShapePrs = new AIS_Shape (theShape);
myContext->Display (aShapePrs, AIS_Shaded, -1, true, true);
```
// activates decomposition of shapes into faces
const int aSubShapeSelMode = AIS_Shape::SelectionMode(TopAbs_Face);
myContext->Activate(aShapePrs, aSubShapeSelMode);

Handle(StdSelect_FaceFilter) aFil1 = new StdSelect_FaceFilter(StdSelect_Revol);
Handle(StdSelect_FaceFilter) aFil2 = new StdSelect_FaceFilter(StdSelect_Plane);
myContext->AddFilter(aFil1);
myContext->AddFilter(aFil2);

// only faces of revolution or planar faces will be selected
myContext->MoveTo(thePixelX, thePixelY, myView);

Selection

Dynamic detection and selection are put into effect in a straightforward way. There are only a few conventions and functions to be familiar with:

- **AIS_InteractiveContext::MoveTo** – passes mouse position to Interactive Context selectors.
- **AIS_InteractiveContext::Select** – stores what has been detected at the last MoveTo. Replaces the previously selected object. Empties the stack if nothing has been detected at the last move.
- **AIS_InteractiveContext::ShiftSelect** – if the object detected at the last move was not already selected, it is added to the list of the selected objects. If not, it is withdrawn. Nothing happens if you click on an empty area.
- **AIS_InteractiveContext::Select** – selects everything found in the surrounding area.
- **AIS_InteractiveContext::ShiftSelect** – selects what was not previously in the list of selected, deselects those already present.

Highlighting of detected and selected entities is automatically managed by the Interactive Context. The Highlight colors are those dealt with above. You can nonetheless disconnect this automatic mode if you want to manage this part yourself:
You can question the Interactive context by moving the mouse. The following functions can be used:

- `AIS_InteractiveContext::HasDetected` – checks if there is a detected entity;
- `AIS_InteractiveContext::DetectedOwner` – returns the (currently highlighted) detected entity.

After using the `Select` and `ShiftSelect` functions, you can explore the list of selections. The following functions can be used:

- `AIS_InteractiveContext::InitSelected` – initializes an iterator;
- `AIS_InteractiveContext::MoreSelected` – checks if the iterator is valid;
- `AIS_InteractiveContext::NextSelected` – moves the iterator to the next position;
- `AIS_InteractiveContext::SelectedOwner` – returns an entity at the current iterator position.

The owner object `SelectMgr_EntityOwner` is a key object identifying selectable entity in the viewer (returned by methods `AIS_InteractiveContext::DetectedOwner` and `AIS_InteractiveContext::SelectedOwner`). The Interactive Object itself can be retrieved by method `SelectMgr_EntityOwner::Selectable`, while identifying sub-part depends on type of Interactive Object. In case of `AIS_Shape`, the (sub)shape is returned by method `StdSelect_BRepOwner::Shape`.

#### Example

```cpp
for (myAISCtx->InitSelected(); myAISCtx->MoreSelected(); myAISCtx->NextSelected())
{
    Handle(SelectMgr_EntityOwner) anOwner = myAISCtx->SelectedOwner();
    Handle(AIS_InteractiveObject) anObj = Handle(AIS_InteractiveObject)::DownCast
```
(anOwner->Selectable());
if (Handle(StdSelect_BRepOwner) aBRepOwner =
    Handle(StdSelect_BRepOwner)::DownCast (anOwner))
{
    // to be able to use the picked shape
    TopoDS_Shape aShape = aBRepOwner->Shape();
}
Standard Interactive Object Classes

Interactive Objects are selectable and viewable objects connecting graphic representation and the underlying reference geometry.

They are divided into four types:

- the **Datum** – a construction geometric element;
- the **Relation** – a constraint on the interactive shape and the corresponding reference geometry;
- the **Object** – a topological shape or connection between shapes;
- **None** – a token, that instead of eliminating the object, tells the application to look further until it finds an acceptable object definition in its generation.

Inside these categories, there is a possibility of additional characterization by means of a signature. The signature provides an index to the further characterization. By default, the **Interactive Object** has a *None* type and a signature of 0 (equivalent to *None*). If you want to give a particular type and signature to your interactive object, you must redefine the two virtual methods: *Type* and *Signature*.

**Datum**

The **Datum** groups together the construction elements such as lines, circles, points, trihedrons, plane trihedrons, planes and axes.

*AIS_Point, AIS_Axis, AIS_Line, AIS_Circle, AIS_Plane* and *AIS_Trihedron* have four selection modes:

- mode 0 : selection of a trihedron;
- mode 1 : selection of the origin of the trihedron;
- mode 2 : selection of the axes;
- mode 3 : selection of the planes XOY, YOZ, XOZ.

when you activate one of modes: 1 2 3 4, you pick AIS objects of type:

- *AIS_Point*;
- *AIS_Axis* (and information on the type of axis);
• *AIS_Plane* (and information on the type of plane).

*AIS_PlaneTrihedron* offers three selection modes:

- mode 0: selection of the whole trihedron;
- mode 1: selection of the origin of the trihedron;
- mode 2: selection of the axes – same remarks as for the Trihedron.

For the presentation of planes and trihedra, the default length unit is millimeter and the default value for the representation of axes is 10. To modify these dimensions, you must temporarily recover the object **Drawer**. From it, take the *DatumAspect()* and change the value *FirstAxisLength*. Finally, recalculate the presentation.

**Object**

The **Object** type includes topological shapes, and connections between shapes.

*AIS_Shape* has two visualization modes:

- mode 0: Line (default mode)
- mode 1: Shading (depending on the type of shape)

*AIS_ConnectedInteractive* is an Interactive Object connecting to another interactive object reference, and located elsewhere in the viewer makes it possible not to calculate presentation and selection, but to deduce them from your object reference. *AIS_MultipleConnectedInteractive* is an object connected to a list of interactive objects (which can also be Connected objects. It does not require memory-hungry presentation calculations).

*MeshVS_Mesh* is an Interactive Object that represents meshes, it has a data source that provides geometrical information (nodes, elements) and can be built up from the source data with a custom presentation builder.

The class *AIS_ColoredShape* allows using custom colors and line widths for *TopoDS_Shape* objects and their sub-shapes.

```plaintext
AIS_ColoredShape aColoredShape = new AIS_ColoredShape (theShape);
```
// setup color of entire shape
aColoredShape->SetColor (Quantity_NOC_RED);

// setup line width of entire shape
aColoredShape->SetWidth (1.0);

// set transparency value
aColoredShape->SetTransparency (0.5);

// customize color of specified sub-shape
aColoredShape->SetCustomColor (theSubShape, Quantity_NOC_BLUE1);

// customize line width of specified sub-shape
aColoredShape->SetCustomWidth (theSubShape, 0.25);

The presentation class \textit{AIS\_PointCloud} can be used for efficient drawing of large arbitrary sets of colored points. It uses \textit{Graphic3d\_ArrayOfPoints} to pass point data into OpenGL graphic driver to draw a set points as an array of "point sprites". The point data is packed into vertex buffer object for performance.

- The type of point marker used to draw points can be specified as a presentation aspect.
- The presentation provides selection by a bounding box of the visualized set of points. It supports two display / highlighting modes: points or bounding box.
Example:

```cpp
Handle(Graphic3d_ArrayOfPoints) aPoints = new Graphic3d_ArrayOfPoints(2000, Standard_True);
aPoints->AddVertex(gp_Pnt(-40.0, -40.0, -40.0), Quantity_Color(Quantity_NOC_BLUE1));
aPoints->AddVertex(gp_Pnt(40.0, 40.0, 40.0), Quantity_Color(Quantity_NOC_BLUE2));

Handle(AIS_PointCloud) aPntCloud = new AIS_PointCloud();
aPntCloud->SetPoints(aPoints);
```

The draw command `vpointcloud` builds a cloud of points from shape triangulation. This command can also draw a sphere surface or a volume with a large amount of points (more than one million).

**Relations**
The Relation is made up of constraints on one or more interactive shapes and the corresponding reference geometry. For example, you might want to constrain two edges in a parallel relation. This constraint is considered as an object in its own right, and is shown as a sensitive primitive. This takes the graphic form of a perpendicular arrow marked with the $\|\|$ symbol and lying between the two edges.

The following relations are provided by AIS:

- AIS_ConcentricRelation
- AIS_FixRelation
- AIS_IdenticRelation
- AIS_ParallelRelation
- AIS_PerpendicularRelation
- AIS_Relation
- AIS_SymmetricRelation
- AIS_TangentRelation

The list of relations is not exhaustive.

Dimensions

- AIS_AngleDimension
- AIS_Chamf3dDimension
- AIS_DiameterDimension
- AIS_DimensionOwner
- AIS_LengthDimension
- AIS_OffsetDimension
- AIS_RadiusDimension

MeshVS_Mesh

MeshVS_Mesh is an Interactive Object that represents meshes. This object differs from the AIS_Shape as its geometrical data is supported by the data source MeshVS_DataSource that describes nodes and elements of the object. As a result, you can provide your own data source.

However, the DataSource does not provide any information on attributes, for example nodal colors, but you can apply them in a special way – by
choosing the appropriate presentation builder.

The presentations of `MeshVS_Mesh` are built with the presentation builders `MeshVS_PrsBuilder`. You can choose between the builders to represent the object in a different way. Moreover, you can redefine the base builder class and provide your own presentation builder.

You can add/remove builders using the following methods:

```cpp
MeshVS_Mesh::AddBuilder (const Handle(MeshVS_PrsBuilder)& theBuilder,
                          Standard_Boolean theToTreatAsHilighter);
MeshVS_Mesh::RemoveBuilder (const Standard_Integer theIndex);
MeshVS_Mesh::RemoveBuilderById (const
                                Standard_Integer theId);
```

There is a set of reserved display and highlighting mode flags for `MeshVS_Mesh`. Mode value is a number of bits that allows selecting additional display parameters and combining the following mode flags, which allow displaying mesh in wireframe, shading and shrink modes:

- `MeshVS_DMF_WireFrame`
- `MeshVS_DMF_Shading`
- `MeshVS_DMF_Shrink`

It is also possible to display deformed mesh in wireframe, shading or shrink modes using:

- `MeshVS_DMF_DeformedPrsWireFrame`
- `MeshVS_DMF_DeformedPrsShading`
- `MeshVS_DMF_DeformedPrsShrink`

The following methods represent different kinds of data:

- `MeshVS_DMF_VectorDataPrs`
- `MeshVS_DMF_NodalColorDataPrs`
- `MeshVS_DMF_ElementalColorDataPrs`
- `MeshVS_DMF_TextDataPrs`
MeshVS_DMF_EntitiesWithData

The following methods provide selection and highlighting:

MeshVS_DMF_SelectionPrs
MeshVS_DMF_HilightPrs

MeshVS_DMF_User is a user-defined mode.

These values will be used by the presentation builder. There is also a set of selection modes flags that can be grouped in a combination of bits:

- MeshVS_SMF_0D
- MeshVS_SMF_Link
- MeshVS_SMF_Face
- MeshVS_SMF_Volume
- MeshVS_SMF_Element – groups 0D, Link, Face and Volume as a bit mask;
- MeshVS_SMF_Node
- MeshVS_SMF_All – groups Element and Node as a bit mask;
- MeshVS_SMF_Mesh
- MeshVS_SMF_Group

Such an object, for example, can be used for displaying the object and stored in the STL file format:

```cpp
// read the data and create a data source
Handle(Poly_Triangulation) aSTLMesh = RWStl::ReadFile(aFileName);
Handle(XSDRAWSTLVRML_DataSource) aDataSource = new XSDRAWSTLVRML_DataSource (aSTLMesh);

// create mesh
Handle(MeshVS_Mesh) aMeshPrs = new MeshVS();
aMeshPrs->SetDataSource (aDataSource);

// use default presentation builder
Handle(MeshVS_MeshPrsBuilder) aBuilder = new MeshVS_MeshPrsBuilder (aMeshPrs);
```
MeshVS_NodalColorPrsBuilder allows representing a mesh with a color scaled texture mapped on it. To do this you should define a color map for the color scale, pass this map to the presentation builder, and define an appropriate value in the range of 0.0 - 1.0 for every node. The following example demonstrates how you can do this (check if the view has been set up to display textures):

```cpp
// assign nodal builder to the mesh
Handle(MeshVS_NodalColorPrsBuilder) aBuilder = new
    MeshVS_NodalColorPrsBuilder (theMeshPrs,
    MeshVS_DMF_NodalColorDataPrs |
    MeshVS_DMF_OCCMask);
aBuilder->UseTexture (true);

// prepare color map
Aspect_SequenceOfColor aColorMap;
aColorMap.Append (Quantity_NOC_RED);
aColorMap.Append (Quantity_NOC_BLUE1);

// assign color scale map values (0..1) to nodes
TColStd_DataMapOfIntegerReal aScaleMap;
...
// iterate through the nodes and add an node id and an appropriate value to the map
aScaleMap.Bind (anId, aValue);

// pass color map and color scale values to the builder
aBuilder->SetColorMap (aColorMap);
aBuilder->SetInvalidColor (Quantity_NOC_BLACK);
aBuilder->SetTextureCoords (aScaleMap);
aMesh->AddBuilder (aBuilder, true);
```
Dynamic Selection

The dynamic selection represents the topological shape, which you want to select, by decomposition of sensitive primitives – the sub-parts of the shape that will be detected and highlighted. The sets of these primitives are handled by the powerful three-level BVH tree selection algorithm.

For more details on the algorithm and examples of usage, please, refer to Selection chapter.
3D Presentations

Glossary of 3D terms

- **Group** – a set of primitives and attributes on those primitives. Primitives and attributes may be added to a group but cannot be removed from it, unless erased globally. A group can have a pick identity.
- **Light** There are five kinds of light source – ambient, headlight, directional, positional and spot. The light is only activated in a shading context in a view.
- **Primitive** – a drawable element. It has a definition in 3D space. Primitives can either be lines, faces, text, or markers. Once displayed markers and text remain the same size. Lines and faces can be modified e.g. zoomed. Primitives must be stored in a group.
- **Structure** – manages a set of groups. The groups are mutually exclusive. A structure can be edited, adding or removing groups. A structure can reference other structures to form a hierarchy. It has a default (identity) transformation and other transformations may be applied to it (rotation, translation, scale, etc). It has no default attributes for the primitive lines, faces, markers, and text. Attributes may be set in a structure but they are overridden by the attributes in each group. Each structure has a display priority associated with it, which rules the order in which it is redrawn in a 3D viewer. If the visualization mode is incompatible with the view it is not displayed in that view, e.g. a shading-only object is not visualized in a wireframe view.
- **View** – is defined by a view orientation, a view mapping, and a context view.
- **Viewer** – manages a set of views.
- **View orientation** – defines the manner in which the observer looks at the scene in terms of View Reference Coordinates.
- **View mapping** – defines the transformation from View Reference Coordinates to the Normalized Projection Coordinates. This follows the Phigs scheme.
- **Z-Buffering** – a form of hidden surface removal in shading mode only. This is always active for a view in the shading mode. It cannot be suppressed.
Graphic primitives

The Graphic3d package is used to create 3D graphic objects in a 3D viewer. These objects called structures are made up of groups of primitives and attributes, such as polylines, planar polygons with or without holes, text and markers, and attributes, such as color, transparency, reflection, line type, line width, and text font. A group is the smallest editable element of a structure. A transformation can be applied to a structure. Structures can be connected to form a tree of structures, composed by transformations. Structures are globally manipulated by the viewer.

Graphic structures can be:

- Displayed,
- Highlighted,
- Erased,
- Transformed,
- Connected to form a tree hierarchy of structures, created by transformations.

There are classes for:

- Visual attributes for lines, faces, markers, text, materials,
- Vectors and vertices,
- Graphic objects, groups, and structures.

Structure hierarchies

The root is the top of a structure hierarchy or structure network. The attributes of a parent structure are passed to its descendants. The attributes of the descendant structures do not affect the parent. Recursive structure networks are not supported.

Graphic primitives

- Markers
  - Have one or more vertices,
  - Have a type, a scale factor, and a color,
• Have a size, shape, and orientation independent of transformations.

**Triangulation**

• Have at least three vertices,
• Have nodal normals defined for shading,
• Have interior attributes – style, color, front and back material, texture and reflection ratio,

**Polylines or Segments**

• Have two or more vertices,
• Have the following attributes – type, width scale factor, color.

**Text**

• Has geometric and non-geometric attributes,
• Geometric attributes – character height, character up vector, text path, horizontal and vertical alignment, orientation, three-dimensional position, zoomable flag
• Non-geometric attributes – text font, character spacing, character expansion factor, color.

**Primitive arrays**

The different types of primitives could be presented with the following primitive arrays:

- `Graphic3d_ArrayOfPoints`,
- `Graphic3d_ArrayOfPolylines`,
- `Graphic3d_ArrayOfSegments`,
- `Graphic3d_ArrayOfTriangleFans`,
- `Graphic3d_ArrayOfTriangles`,
- `Graphic3d_ArrayOfTriangleStrips`.

The `Graphic3d_ArrayOfPrimitives` is a base class for these primitive arrays. Method `set Graphic3d_ArrayOfPrimitives::AddVertex` allows adding vertices to the primitive array with their attributes (color, normal, texture coordinates). You can also modify the values assigned to the vertex or query these values by the vertex index.

The following example shows how to define an array of points:

```cpp
// create an array
Handle(Graphic3d_ArrayOfPoints) anArray = new
```
Graphic3d_ArrayOfPoints (theVerticesMaxCount);

// add vertices to the array
anArray->AddVertex (10.0, 10.0, 10.0);
anArray->AddVertex (0.0, 10.0, 10.0);

// add the array to the structure
Handle(Graphic3d_Group) aGroup = thePrs->NewGroup();
aGroup->AddPrimitiveArray (anArray);
aGroup->SetGroupPrimitivesAspect (myDrawer->PointAspect()->Aspect());

If the primitives share the same vertices (polygons, triangles, etc.) then you can define them as indices of the vertices array. The method Graphic3d_ArrayOfPrimitives::AddEdge allows defining the primitives by indices. This method adds an "edge" in the range \[1, \text{VertexNumber()}\] in the array. It is also possible to query the vertex defined by an edge using method Graphic3d_ArrayOfPrimitives::Edge.

The following example shows how to define an array of triangles:

// create an array
Standard_Boolean hasNormals = false;
Standard_Boolean hasColors = false;
Standard_Boolean hasTextureCrds = false;
Handle(Graphic3d_ArrayOfTriangles) anArray = new Graphic3d_ArrayOfTriangles (theVerticesMaxCount, theEdgesMaxCount, hasNormals, hasColors, hasTextureCrds);

// add vertices to the array
anArray->AddVertex (-1.0, 0.0, 0.0); // vertex 1
anArray->AddVertex ( 1.0, 0.0, 0.0); // vertex 2
anArray->AddVertex ( 0.0, 1.0, 0.0); // vertex 3
anArray->AddVertex ( 0.0,-1.0, 0.0); // vertex 4

// add edges to the array
anArray->AddEdge (1); // first triangle
anArray->AddEdge (2);
anArray->AddEdge (3);
anArray->AddEdge (1); // second triangle
anArray->AddEdge (2);
anArray->AddEdge (4);

// add the array to the structure
Handle(Graphic3d_Group) aGroup = thePrs->NewGroup();
aGroup->AddPrimitiveArray (anArray);
aGroup->SetGroupPrimitivesAspect (myDrawerShadingAspect()->Aspect());

Text primitive

TKOpenGL toolkit renders text labels using texture fonts. Graphic3d text primitives have the following features:

- fixed size (non-zoomable) or zoomable,
- can be rotated to any angle in the view plane,
- support unicode charset.

The text attributes for the group could be defined with the Graphic3d_AspectText3d attributes group. To add any text to the graphic structure you can use the following methods:

```cpp
void Graphic3d_Group::Text (const Standard_CString theText, const Graphic3d_Vertex& thePoint, const Standard_Real theHeight, const Quantity_PlaneAngle theAngle, const Graphic3d_TextPath theTp, const Graphic3d_HorizontalTextAlignment theHta, const Graphic3d_VerticalTextAlignment theVta,
```

```cpp
```

```cpp
```
### Graphic3d_Group::Text

#### C++

```cpp
void Graphic3d_Group::Text (const Standard_CString theText,
    const Graphic3d_Vertex& thePoint,
    const Standard_Real theHeight,
    const Standard_Boolean theToEvalMinMax);
```

#### TC++

```cpp
void Graphic3d_Group::Text (const TCcollection_ExtendedString& theText,
    const Graphic3d_Vertex& thePoint,
    const Standard_Real theHeight,
    const Quantity_PlaneAngle theAngle,
    const Graphic3d_TextPath theTp,
    const Graphic3d_HorizontalTextAlignment theHta,
    const Graphic3d_VerticalTextAlignment theVta,
    const Standard_Boolean theToEvalMinMax);
```

The `theText` parameter is the text string, `thePoint` is the three-dimensional position of the text, `theHeight` is the text height, `theAngle` is the orientation of the text (at the moment, this parameter has no effect, but you can specify the text orientation through the `Graphic3d_AspectText3d` attributes). `theTp` parameter defines the text path, `theHta` is the horizontal alignment of the text, `theVta` is the vertical alignment of the text. You can pass FALSE as `theToEvalMinMax` if you do not want the graphic3d structure boundaries to be affected by the text position.

**Note** that the text orientation angle can be defined by `Graphic3d_AspectText3d` attributes.
See the example:

```cpp
// get the group
Handle(Graphic3d_Group) aGroup = thePrs->NewGroup();

// change the text aspect
Handle(Graphic3d_AspectText3d) aTextAspect = new Graphic3d_AspectText3d();
aTextAspect->SetTextZoomable (true);
aTextAspect->SetTextAngle (45.0);
aGroup->SetPrimitivesAspect (aTextAspect);

// add a text primitive to the structure
Graphic3d_Vertex aPoint (1, 1, 1);
aGroup->Text (Standard_CString ("Text"), aPoint, 16.0);
```

**Materials**

A `Graphic3d_MaterialAspect` is defined by:

- Transparency;
- Diffuse reflection – a component of the object color;
- Ambient reflection;
- Specular reflection – a component of the color of the light source;
- Refraction index.

The following items are required to determine the three colors of
reflection:

- Color;
- Coefficient of diffuse reflection;
- Coefficient of ambient reflection;
- Coefficient of specular reflection.

Textures

A *texture* is defined by a name. Three types of texture are available:

- 1D;
- 2D;
- Environment mapping.

Shaders

OCCT visualization core supports GLSL shaders. Shaders can be assigned to a generic presentation by its drawer attributes (Graphic3d aspects). To enable custom shader for a specific AIS_Shape in your application, the following API functions can be used:

```cpp
// Create shader program
Handle(Graphic3d_ShaderProgram) aProgram = new Graphic3d_ShaderProgram();

// Attach vertex shader
aProgram->AttachShader
    (Graphic3d_ShaderObject::CreateFromFile
    (Graphic3d_TOS_VERTEX, "<Path to VS>"));

// Attach fragment shader
aProgram->AttachShader
    (Graphic3d_ShaderObject::CreateFromFile
    (Graphic3d_TOS_FRAGMENT, "<Path to FS>"));

// Set values for custom uniform variables (if they are)
aProgram->PushVariable("MyColor", Graphic3d_Vec3
```
(0.0f, 1.0f, 0.0f));

// Set aspect property for specific AIS_Shape
theAISShape->Attributes()->ShadingAspect()->Aspect()->SetShaderProgram (aProgram);
Graphic attributes

Aspect package overview

The Aspect package provides classes for the graphic elements in the viewer:

- Groups of graphic attributes;
- Edges, lines, background;
- Window;
- Driver;
- Enumerations for many of the above.
3D view facilities

Overview

The V3d package provides the resources to define a 3D viewer and the views attached to this viewer (orthographic, perspective). This package provides the commands to manipulate the graphic scene of any 3D object visualized in a view on screen.

A set of high-level commands allows the separate manipulation of parameters and the result of a projection (Rotations, Zoom, Panning, etc.) as well as the visualization attributes (Mode, Lighting, Clipping, etc.) in any particular view.

The V3d package is basically a set of tools directed by commands from the viewer front-end. This tool set contains methods for creating and editing classes of the viewer such as:

- Default parameters of the viewer,
- Views (orthographic, perspective),
- Lighting (positional, directional, ambient, spot, headlight),
- Clipping planes,
- Instantiated sequences of views, planes, light sources, graphic structures, and picks,
- Various package methods.

A programming example

This sample TEST program for the V3d Package uses primary packages Xw and Graphic3d and secondary packages Visual3d, Aspect, Quantity and math.

```c++
// create a default display connection
Handle(Aspect_DisplayConnection) aDispConnection =
    new Aspect_DisplayConnection();
// create a Graphic Driver
Handle(OpenGl_GraphicDriver) aGraphicDriver = new
    OpenGl_GraphicDriver (aDispConnection);
```
// create a Viewer to this Driver
Handle(V3d_Viewer) VM = new V3d_Viewer
   (aGraphicDriver);
VM->SetDefaultBackgroundColor
   (Quantity_NOC_DARKVIOLET);
VM->SetDefaultViewProj (V3d_Xpos);
// Create a structure in this Viewer
Handle(Graphic3d_Structure) aStruct = new
   Graphic3d_Structure (VM->Viewer());

// Type of structure
aStruct->SetVisual (Graphic3d_TOS_SHADING);

// Create a group of primitives in this structure
Handle(Graphic3d_Group) aPrsGroup = new
   Graphic3d_Group (aStruct);

// Fill this group with one quad of size 100
Handle(Graphic3d_ArrayOfTriangleStrips) aTriangles =
   new Graphic3d_ArrayOfTriangleStrips (4);
aTriangles->AddVertex (-100./2., -100./2., 0.0);
aTriangles->AddVertex (-100./2., 100./2., 0.0);
aTriangles->AddVertex (100./2., -100./2., 0.0);
aTriangles->AddVertex (100./2., 100./2., 0.0);
aPrsGroup->AddPrimitiveArray (aTriangles);
aPrsGroup->SetGroupPrimitivesAspect (new
   Graphic3d_AspectFillArea3d());

// Create Ambient and Infinite Lights in this Viewer
Handle(V3d_AmbientLight) aLight1 = new
   V3d_AmbientLight (VM, Quantity_NOC_GRAY50);
Handle(V3d_DirectionalLight) aLight2 = new
   V3d_DirectionalLight (VM, V3d_XnegYnegZneg,
                        Quantity_NOC_WHITE);

// Create a 3D quality Window with the same DisplayConnection
Handle(Xw_Window) aWindow = new Xw_Window
    (aDispConnection, "Test V3d", 0.5, 0.5, 0.5, 0.5);

// Map this Window to this screen
aWindow->Map();

// Create a Perspective View in this Viewer
Handle(V3d_View) aView = new V3d_View (VM);
aView->Camera()->SetProjectionType
    (Graphic3d_Camera::Projection_Perspective);
// Associate this View with the Window
aView->SetWindow (aWindow);
// Display ALL structures in this View
VM->Viewer()->Display();
// Finally update the Visualization in this View
aView->Update();
// Fit view to object size
V->FitAll();

Define viewing parameters

View projection and orientation in OCCT V3d_View are driven by camera. The camera calculates and supplies projection and view orientation matrices for rendering by OpenGL. The allows to the user to control all projection parameters. The camera is defined by the following properties:

- **Eye** – defines the observer (camera) position. Make sure the Eye point never gets between the Front and Back clipping planes.
- **Center** – defines the origin of View Reference Coordinates (where camera is aimed at).
- **Direction** – defines the direction of camera view (from the Eye to the Center).
- **Distance** – defines the distance between the Eye and the Center.
- **Front Plane** – defines the position of the front clipping plane in View Reference Coordinates system.
- **Back Plane** – defines the position of the back clipping plane in View Reference Coordinates system.
- **ZNear** – defines the distance between the Eye and the Front plane.
- **ZFar** – defines the distance between the Eye and the Back plane.

Most common view manipulations (panning, zooming, rotation) are implemented as convenience methods of `V3d_View` class, however `Graphic3d_Camera` class can also be used directly by application developers:

Example:

```cpp
// rotate camera by X axis on 30.0 degrees
gp_Trsf aTrsf;
aTrsf.SetRotation (gp_Ax1 (gp_Pnt (0.0, 0.0, 0.0),
    gp_Dir (1.0, 0.0, 0.0)), 30.0);
aView->Camera()->Transform (aTrsf);
```

**Orthographic Projection**

The following code configures the camera for orthographic rendering:

```cpp
// Create an orthographic View in this Viewer
Handle(V3d_View) aView = new V3d_View (VM);
aView->Camera()->SetProjectionType
```
Perspective Projection

Field of view (FOVy) – defines the field of camera view by y axis in degrees (45° is default).

Perspective frustum

The following code configures the camera for perspective rendering:

```cpp
// Create a perspective View in this Viewer
Handle(V3d_View) aView = new V3d_View(VM);
aView->Camera()->SetProjectionType
    (Graphic3d_Camera::Projection_Perspective);
aView->Update();
```

Stereographic Projection

IOD – defines the intraocular distance (in world space units).

There are two types of IOD:
- **IODType_Absolute**: Intraocular distance is defined as an absolute value.
- **IODType_Relative**: Intraocular distance is defined relative to the camera focal length (as its coefficient).

**Field of view (FOV)** – defines the field of camera view by y axis in degrees (45° is default).

**ZFocus** – defines the distance to the point of stereographic focus.

![Stereographic projection diagram](image)

**Stereographic projection**

To enable stereo projection, your workstation should meet the following requirements:

- The graphic card should support quad buffering.
- You need active 3D glasses (LCD shutter glasses).
- The graphic driver needs to be configured to impose quad buffering for newly created OpenGL contexts; the viewer and the view should be created after that.
In stereographic projection mode the camera prepares two projection matrices to display different stereo-pictures for the left and for the right eye. In a non-stereo camera this effect is not visible because only the same projection is used for both eyes.

To enable quad buffering support you should provide the following settings to the graphic driver `opengl_caps`:

```cpp
Handle(OpenGL_GraphicDriver) aDriver = new OpenGL_GraphicDriver();
OpenGL_Caps& aCaps = aDriver->ChangeOptions();
aCaps.contextStereo = Standard_True;
```

The following code configures the camera for stereographic rendering:

```cpp
// Create a Stereographic View in this Viewer
Handle(V3d_View) aView = new V3d_View(VM);
aView->Camera()->SetProjectionType
    (Graphic3d_Camera::Projection_Stereo);
// Change stereo parameters
aView->Camera()->SetIOD (IODType_Absolute, 5.0);
// Finally update the Visualization in this View
aView->Update();
```

View frustum culling

The algorithm of frustum culling on CPU-side is activated by default for 3D viewer. This algorithm allows skipping the presentation outside camera at the rendering stage, providing better performance. The following features support this method:

- `Graphic3d_Structure::CalculateBoundBox()` is used to calculate axis-aligned bounding box of a presentation considering its transformation.
- `V3d_View::SetFrustumCulling` enables or disables frustum culling for the specified view.
- Classes `OpenGL_BVHClipPrimitiveSet` and `OpenGL_BVHTreeSelector` handle the detection of outer objects and usage of acceleration structure for frustum culling.
• **BVH_BinnedBuilder** class splits several objects with null bounding box.

**View background styles**

There are three types of background styles available for **V3d_View**: solid color, gradient color and image.

To set solid color for the background you can use the following method:

```cpp
void V3d_View::SetBackgroundColor (const Quantity_Color& theColor);
```

The gradient background style could be set up with the following method:

```cpp
void V3d_View::SetBgGradientColors (const Quantity_Color& theColor1, const Quantity_Color& theColor2, const Aspect_GradientFillMethod theFillStyle, const Standard_Boolean theToUpdate = false);
```

The **theColor1** and **theColor2** parameters define the boundary colors of interpolation, the **theFillStyle** parameter defines the direction of interpolation.

To set the image as a background and change the background image style you can use the following method:

```cpp
void V3d_View::SetBackgroundImage (const Standard_CString theFileName, const Aspect_FillMethod theFillStyle, const Standard_Boolean theToUpdate = false);
```

The **theFileName** parameter defines the image file name and the path to
it, the *theFillStyle* parameter defines the method of filling the background with the image. The methods are:

- *Aspect_FM_NONE* – draws the image in the default position;
- *Aspect_FM_CENTERED* – draws the image at the center of the view;
- *Aspect_FM_TILED* – tiles the view with the image;
- *Aspect_FM_STRETCH* – stretches the image over the view.

**Dumping a 3D scene into an image file**

The 3D scene displayed in the view can be dumped into image file with resolution independent from window size (using offscreen buffer). The *V3d_View* has the following methods for dumping the 3D scene:

```cpp
Standard_Boolean V3d_View::Dump (const Standard_CString theFile, const Image_TypeOfImage theBufferType);
```

Dumps the scene into an image file with the view dimensions. The raster image data handling algorithm is based on the *Image_AlienPixMap* class. The supported extensions are ".png", ".bmp", ".jpg" and others supported by *Freelmage* library. The value passed as *theBufferType* argument defines the type of the buffer for an output image *(RGB, RGBA, floating-point, RGBF, RGBA)*. Method returns TRUE if the scene has been successfully dumped.

```cpp
Standard_Boolean V3d_View::ToPixMap (Image_PixMap& theImage, const V3d_ImageDumpOptions& theParams);
```

Dumps the displayed 3d scene into a pixmap with a width and height passed through parameters structure *theParams*.

**Ray tracing support**

OCCT visualization provides rendering by real-time ray tracing technique.
It is allowed to switch easily between usual rasterization and ray tracing rendering modes. The core of OCCT ray tracing is written using GLSL shaders. The ray tracing has a wide list of features:

- Hard shadows
- Refractions
- Reflection
- Transparency
- Texturing
- Support of non-polygon objects, such as lines, text, highlighting, selection.
- Performance optimization using 2-level bounding volume hierarchy (BVH).

The ray tracing algorithm is recursive (Whitted's algorithm). It uses BVH effective optimization structure. The structure prepares optimized data for a scene geometry for further displaying it in real-time. The time-consuming re-computation of the BVH is not necessary for view operations, selections, animation and even editing of the scene by transforming location of the objects. It is only necessary when the list of displayed objects or their geometry changes. To make the BVH reusable it has been added into an individual reusable OCCT package TKMath/BVH.

There are several ray-tracing options that user can switch on/off:

- Maximum ray tracing depth
- Shadows rendering
- Specular reflections
- Adaptive anti aliasing
- Transparency shadow effects

Example:

```cpp
Graphic3d_RenderingParams& aParams = aView->ChangeRenderingParams();
// specifies rendering mode
aParams.Method = Graphic3d_RM_RAYTRACING;
// maximum ray-tracing depth
aParams.RaytracingDepth = 3;
```
// enable shadows rendering
aParams.IsShadowEnabled = true;
// enable specular reflections.
aParams.IsReflectionEnabled = true;
// enable adaptive anti-aliasing
aParams.IsAntialiasingEnabled = true;
// enable light propagation through transparent media.
aParams.IsTransparentShadowEnabled = true;
// update the view
aView->Update();

**Display priorities**

Structure display priorities control the order, in which structures are drawn. When you display a structure you specify its priority. The lower is the value, the lower is the display priority. When the display is regenerated, the structures with the lowest priority are drawn first. The structures with the same display priority are drawn in the same order as they have been displayed. OCCT supports eleven structure display priorities.

**Z-layer support**

OCCT features depth-arranging functionality called z-layer. A graphical presentation can be put into a z-layer. In general, this function can be used for implementing "bring to front" functionality in a graphical application.

Example:

```cpp
// set z-layer to an interactive object
Handle(AIS_InteractiveContext) theContext;
Handle(AIS_InteractiveObject) theInterObj;
Standard_Integer anId = 3;
aViewer->AddZLayer (anId);
theContext->SetZLayer (theInterObj, anId);
```
For each z-layer, it is allowed to:

- Enable / disable depth test for layer.
- Enable / disable depth write for layer.
- Enable / disable depth buffer clearing.
- Enable / disable polygon offset.

You can get the options using getter from V3d_Viewer. It returns Graphic3d_ZLayerSettings for a given LayerId.

Example:

```cpp
// change z-layer settings
Graphic3d_ZLayerSettings aSettings = aViewer->ZLayerSettings (anId);
aSettings.SetEnableDepthTest (true);
aSettings.SetEnableDepthWrite(true);
aSettings.SetClearDepth (true);
aSettings.SetPolygonOffset (Graphic3d_PolygonOffset());
aViewer->SetZLayerSettings (anId, aSettings);
```

Another application for Z-Layer feature is treating visual precision issues when displaying objects far from the World Center. The key problem with such objects is that visualization data is stored and manipulated with single precision floating-point numbers (32-bit). Single precision 32-bit floating-point numbers give only 6-9 significant decimal digits precision, while double precision 64-bit numbers give 15-17 significant decimal digits precision, which is sufficient enough for most applications.

When moving an Object far from the World Center, float number steadily eats precision. The camera Eye position adds leading decimal digits to the overall Object transformation, which discards smaller digits due to floating point number nature. For example, the object of size 0.0000123 moved to position 1000 has result transformation 1000.0000123, which overflows single precision floating point - considering the most optimistic scenario of 9 significant digits (but it is really not this case), the result number will be 1000.00001.

This imprecision results in visual artifacts of two kinds in the 3D Viewer:
• Overall per-vertex Object distortion. This happens when each vertex position has been defined within World Coordinate system.
• The object itself is not distorted, but its position in the World is unstable and imprecise - the object jumps during camera manipulations. This happens when vertices have been defined within Local Coordinate system at the distance small enough to keep precision within single precision float, however Local Transformation applied to the Object is corrupted due to single precision float.

The first issue cannot be handled without switching the entire presentation into double precision (for each vertex position). However, visualization hardware is much faster using single precision float number rather than double precision - so this is not an option in most cases. The second issue, however, can be negated by applying special rendering tricks.

So, to apply this feature in OCCT, the application:

• Defines Local Transformation for each object to fit the presentation data into single precision float without distortion.
• Spatially splits the world into smaller areas/cells where single precision float will be sufficient. The size of such cell might vary and depends on the precision required by application (e.g. how much user is able to zoom in camera within application).
• Defines a Z-Layer for each spatial cell containing any object.
• Defines the Local Origin property of the Z-Layer according to the center of the cell.

```
Graphic3d_ZLayerSettings aSettings = aViewer->ZLayerSettings (anId);
aSettings.SetLocalOrigin (400.0, 0.0, 0.0);
```

• Assigns a presentable object to the nearest Z-Layer.

Note that Local Origin of the Layer is used only for rendering - everything outside will be still defined in the World Coordinate System, including Local Transformation of the Object and Detection results. E.g., while moving the presentation between Z-layers with different Local Origins, the Object will stay at the same place - only visualization quality will vary.
Clipping planes

The ability to define custom clipping planes could be very useful for some tasks. OCCT provides such an opportunity.

The Graphic3d_ClipPlane class provides the services for clipping planes: it holds the plane equation coefficients and provides its graphical representation. To set and get plane equation coefficients you can use the following methods:

```cpp
Graphic3d_ClipPlane::Graphic3d_ClipPlane (const gp_Pln& thePlane)
void Graphic3d_ClipPlane::SetEquation (const gp_Pln& thePlane)
Graphic3d_ClipPlane::Graphic3d_ClipPlane (const Equation& theEquation)
void Graphic3d_ClipPlane::SetEquation (const Equation& theEquation)
gp_Pln Graphic3d_ClipPlane::ToPlane() const
```

The clipping planes can be activated with the following method:

```cpp
void Graphic3d_ClipPlane::SetOn (const Standard_Boolean theIsOn)
```

The number of clipping planes is limited. You can check the limit value via method Graphic3d_GraphicDriver::InquireLimit():

```cpp
// get the limit of clipping planes for the current view
Standard_Integer aMaxClipPlanes = aView->Viewer()->Driver()->InquireLimit
          (Graphic3d_TypeOfLimit_MaxNbClipPlanes);
```

Let us see for example how to create a new clipping plane with custom parameters and add it to a view or to an object:

```cpp
// create a new clipping plane
const Handle(Graphic3d_ClipPlane)& aClipPlane = new
```
Graphic3d_ClipPlane();
// change equation of the clipping plane
Standard_Real aCoeffA = ...
Standard_Real aCoeffB = ...
Standard_Real aCoeffC = ...
Standard_Real aCoeffD = ...
aClipPlane->SetEquation (gp_Pln (aCoeffA, aCoeffB,
aCoeffC, aCoeffD));
// set capping
aClipPlane->SetCapping (aCappingArg == "on");
// set the material with red color of clipping plane
Graphic3d_MaterialAspect aMat = aClipPlane->CappingMaterial();
Quantity_Color aColor (1.0, 0.0, 0.0,
    Quantity_TOC_RGB);
aMat.SetAmbientColor (aColor);
aMat.SetDiffuseColor (aColor);
aClipPlane->SetCappingMaterial (aMat);
// set the texture of clipping plane
Handle(Graphic3d_Texture2Dmanual) aTexture = ...
aTexture->EnableModulate();
aTexture->EnableRepeat();
aClipPlane->SetCappingTexture (aTexture);
// add the clipping plane to an interactive object
Handle(AIS_InteractiveObject) aIObj = ...
aIObj->AddClipPlane (aClipPlane);
// or to the whole view
aView->AddClipPlane (aClipPlane);
// activate the clipping plane
aClipPlane->SetOn(Standard_True);
// update the view
aView->Update();

Automatic back face culling

Back face culling reduces the rendered number of triangles (which improves the performance) and eliminates artifacts at shape boundaries.
However, this option can be used only for solid objects, where the interior is actually invisible from any point of view. Automatic back-face culling mechanism is turned on by default, which is controlled by `V3d_View::SetBackFacingModel()`.

The following features are applied in `StdPrs_ToolShadedShape::IsClosed()`, which is used for definition of back face culling in `ShadingAspect`:

- disable culling for free closed Shells (not inside the Solid) since reversed orientation of a free Shell is a valid case;
- enable culling for Solids packed into a compound;
- ignore Solids with incomplete triangulation.

Back face culling is turned off at TKOpenGL level in the following cases:

- clipping/capping planes are in effect;
- for translucent objects;
- with hatching presentation style.
Examples: creating a 3D scene

To create 3D graphic objects and display them in the screen, follow the procedure below:

1. Create attributes.
2. Create a 3D viewer.
3. Create a view.
4. Create an interactive context.
5. Create interactive objects.
6. Create primitives in the interactive object.
7. Display the interactive object.

Create attributes

Create colors.

```cpp
Quantity_Color aBlack (Quantity_NOC_BLACK);
Quantity_Color aBlue (Quantity_NOC_MATRABLUE);
Quantity_Color aBrown (Quantity_NOC_BROWN4);
Quantity_Color aFirebrick (Quantity_NOC_FIREBRICK);
Quantity_Color aForest (Quantity_NOC_FORESTGREEN);
Quantity_Color aGray (Quantity_NOC_GRAY70);
Quantity_Color aMyColor (0.99, 0.65, 0.31, Quantity_TOC_RGB);
Quantity_Color aBeet (Quantity_NOC_BEET);
Quantity_Color aWhite (Quantity_NOC_WHITE);
```

Create line attributes.

```cpp
Handle(Graphic3d_AspectLine3d) anAspectBrown = new Graphic3d_AspectLine3d();
Handle(Graphic3d_AspectLine3d) anAspectBlue = new Graphic3d_AspectLine3d();
Handle(Graphic3d_AspectLine3d) anAspectWhite = new Graphic3d_AspectLine3d();
anAspectBrown->SetColor (aBrown);
```
Create marker attributes.

Handle(Graphic3d_AspectMarker3d aFirebrickMarker = new Graphic3d_AspectMarker3d();
    // marker attributes
    aFirebrickMarker->SetColor(Firebrick);
    aFirebrickMarker->SetScale (1.0);
    aFirebrickMarker->SetType (Aspect_TOM_BALL);
    // or this
    // it is a preferred way (supports full-color images on modern hardware).
    aFirebrickMarker->SetMarkerImage (theImage)

Create facet attributes.

Handle(Graphic3d_AspectFillArea3d) aFaceAspect = new Graphic3d_AspectFillArea3d();
Graphic3d_MaterialAspect aBrassMaterial (Graphic3d_NOM_BRASS);
Graphic3d_MaterialAspect aGoldMaterial (Graphic3d_NOM_GOLD);
    aFaceAspect->SetInteriorStyle (Aspect_IS_SOLID);
    aFaceAspect->SetInteriorColor (aMyColor);
    aFaceAspect->SetDistinguishOn ();
    aFaceAspect->SetFrontMaterial (aGoldMaterial);
    aFaceAspect->SetBackMaterial (aBrassMaterial);
    aFaceAspect->SetEdgeOn();

Create text attributes.

Handle(Graphic3d_AspectText3d) aTextAspect = new Graphic3d_AspectText3d (aForest,
    Graphic3d_NOF_ASCII_MONO, 1.0, 0.0);

Create a 3D Viewer (a Windows example)
Create a default connection
Handle(Aspect_DisplayConnection) aDisplayConnection;

Create a graphic driver from default connection
Handle(OpenGl_GraphicDriver) aGraphicDriver = new OpenGl_GraphicDriver (aDisplayConnection);

Create a viewer
myViewer = new V3d_Viewer (aGraphicDriver);

Set parameters for V3d_Viewer
// defines default lights -
// positional-light 0.3 0.0 0.0
// directional-light V3d_XnegYposZpos
// directional-light V3d_XnegYneg
// ambient-light
a3DViewer->SetDefaultLights();

Activates all the lights defined in this viewer
a3DViewer->SetLightOn();

Set background color to black
a3DViewer->SetDefaultBackgroundColor (Quantity_NOC_BLACK);

Create a 3D view (a Windows example)

It is assumed that a valid Windows window may already be accessed via the method GetSafeHwnd() (as in case of MFC sample).

Create an interactive context
myAISContext = new AIS_InteractiveContext (myViewer);

You are now able to display interactive objects such as an AIS_Shape.
Create your own interactive object

Follow the procedure below to compute the presentable object:

1. Build a presentable object inheriting from `AIS_InteractiveObject` (refer to the Chapter on Presentable Objects).
2. Reuse the `Prs3d_Presentation` provided as an argument of the compute methods.

Note that there are two compute methods: one for a standard representation, and the other for a degenerated representation, i.e. in hidden line removal and wireframe modes.

Let us look at the example of compute methods

```cpp
void MyPresentableObject::Compute (const Handle(PrsMgr_PresentationManager3d)& thePrsManager,
                                      const Handle(Prs3d_Presentation)& thePrs,
                                      const Standard_Integer theMode)
(
     //...
)

void MyPresentableObject::Compute (const Handle(Prs3d_Projector)& theProjector,
                                      const Handle(Prs3d_Presentation)& thePrs)
(
     //...
)
```
Create primitives in the interactive object

Get the group used in Prs3d_Presentation.

Handle(Graphic3d_Group) aGroup = thePrs->NewGroup();

Update the group attributes.

aGroup->SetGroupPrimitivesAspect (anAspectBlue);

Create two triangles in aGroup.

Standard_Integer aNbTria = 2;
Handle(Graphic3d_ArrayOfTriangles) aTriangles = new Graphic3d_ArrayOfTriangles (3 * aNbTria, 0, true);
for (Standard_Integer aTriIter = 1; aTriIter <= aNbTria; ++aTriIter)
{
    aTriangles->AddVertex (aTriIter * 5., 0., 0., 1., 1., 1.);
    aTriangles->AddVertex (aTriIter * 5 + 5, 0., 0., 1., 1., 1.);
    aTriangles->AddVertex (aTriIter * 5 + 2.5, 5., 0., 1., 1., 1.);
}
aGroup->AddPrimitiveArray (aTriangles);
aGroup->SetGroupPrimitivesAspect (new Graphic3d_AspectFillArea3d());

Use the polyline function to create a boundary box for the thePrs structure in group aGroup.

Standard_Real Xm, Ym, Zm, XM, YM, ZM;
thePrs->MinMaxValues (Xm, Ym, Zm, XM, YM, ZM);
Handle(Graphic3d_ArrayOfPolylines) aPolylines = new Graphic3d_ArrayOfPolylines (16, 4);
aPolylines->AddBound (4);
aPolylines->AddVertex (Xm, Ym, Zm);
aPolylines->AddVertex (Xm, Ym, ZM);
aPolylines->AddVertex (XM, YM, ZM);
aPolylines->AddVertex (XM, YM, Zm);
aPolylines->AddBound (4);
aPolylines->AddVertex (XM, YM, Zm);
aPolylines->AddVertex (XM, YM, Zm);
aPolylines->AddVertex (XM, YM, ZM);
aPolylines->AddVertex (XM, YM, Zm);
aPolylines->AddBound (4);
aPolylines->AddVertex (XM, YM, ZM);
aPolylines->AddVertex (XM, YM, Zm);
aPolylines->AddVertex (XM, YM, ZM);
aPolylines->AddVertex (XM, YM, Zm);
aPolylines->AddBound (4);
aPolylines->AddVertex (XM, YM, Zm);
aPolylines->AddVertex (XM, YM, ZM);
aPolylines->AddVertex (XM, YM, Zm);
aPolylines->AddVertex (XM, YM, Zm);

aGroup->AddPrimitiveArray(aPolyline);
aGroup->SetGroupPrimitivesAspect (new Graphic3d_AspectLine3d());

Create text and markers in group aGroup.

static char* texte[3] =
{
   "Application title",
   "My company",
   "My company address."
};
Handle(Graphic3d_ArrayOfPoints) aPtsArr = new Graphic3d_ArrayOfPoints (2, 1);
aPtsArr->AddVertex (-40.0, -40.0, -40.0);
aPtsArr->AddVertex (40.0, 40.0, 40.0);
aGroup->AddPrimitiveArray (aPtsArr);
aGroup->SetGroupPrimitivesAspect (new Graphic3d_AspectText3d());

Graphic3d_Vertex aMarker (0.0, 0.0, 0.0);
for (int i = 0; i <= 2; i++)
{
    aMarker.SetCoord (-(Standard_Real )i * 4 + 30,
                     (Standard_Real )i * 4,
                     -(Standard_Real )i * 4);
    aGroup->Text (texte[i], Marker, 20.);
}

Mesh Visualization Services

MeshVS (Mesh Visualization Service) component extends 3D visualization capabilities of Open CASCADE Technology. It provides flexible means of displaying meshes along with associated pre- and post-processor data.

From a developer's point of view, it is easy to integrate the MeshVS component into any mesh-related application with the following guidelines:

- Derive a data source class from the `MeshVS_DataSource` class.
- Re-implement its virtual methods, so as to give the MeshVS component access to the application data model. This is the most important part of the job, since visualization performance is affected by performance of data retrieval methods of your data source class.
- Create an instance of `MeshVS_Mesh` class.
- Create an instance of your data source class and pass it to a `MeshVS_Mesh` object through the `SetDataSource()` method.
- Create one or several objects of `MeshVS_PrsBuilder`-derived classes (standard, included in the MeshVS package, or your custom ones).
- Each `PrsBuilder` is responsible for drawing a `MeshVS_Mesh` presentation in a certain display mode(s) specified as a `PrsBuilder` constructor's argument. Display mode is treated by MeshVS classes as a combination of bit flags (two least significant bits are used to encode standard display modes: wireframe, shading and shrink).
- Pass these objects to the `MeshVS_Mesh::AddBuilder()` method. `MeshVS_Mesh` takes advantage of improved selection highlighting mechanism: it highlights its selected entities itself, with the help of so called "highlighter" object. You can set one of `PrsBuilder` objects to act as a highlighter with the help of a corresponding argument of the `AddBuilder()` method.

Visual attributes of the `MeshVS_Mesh` object (such as shading color, shrink coefficient and so on) are controlled through `MeshVS_Drawer` object. It maintains a map "Attribute ID --> attribute value" and can be easily extended with any number of custom attributes.
In all other respects, MeshVS_Mesh is very similar to any other class derived from AIS_InteractiveObject and it should be used accordingly (refer to the description of AIS package in the documentation).
Usage of OCCT picking algorithm
DRAW Test Harness
Introduction

VIS component provides adaptation functionality for visualization of OCCT topological shapes by means of VTK library. This User's Guide describes how to apply VIS classes in application dealing with 3D visualization based on VTK library.

There are two ways to use VIS in the application:

- Use a **high-level API**. It is a simple scenario to use VTK viewer with displayed OCCT shapes. It considers usage of tools provided with VIS component such as a specific VTK data source, a picker class and specific VTK filters. Basically, in this scenario you enrich your custom VTK pipeline with extensions coming from VIS.

- Use a **low-level API**. It is an advanced scenario for the users with specific needs, which are not addressed by the higher-level utilities of VIS. It presumes implementation of custom VTK algorithms (such as filters) with help of low-level API of VIS component. This document describes both scenarios of VIS integration into application. To understand this document, it is necessary to be familiar with VTK and OCCT libraries.
Component Architecture

Common structure

VIS component consists of the following packages:

- **IVtk** – common interfaces which define the principal objects playing as foundation of VIS.
- **IVtkOCC** – implementation of interfaces related to CAD domain. The classes from this package deal with topological shapes, faceting and interactive selection facilities of OCCT;
- **IVtkVTK** – implementation of interfaces related to VTK visualization toolkit;
- **IVtkTools** – high-level tools designed for integration into VTK visualization pipelines.

Dependencies of VIS packages

The idea behind the mentioned organization of packages is separation of interfaces from their actual implementations by their dependencies from a particular library (OCCT, VTK). Besides providing of semantic separation, such splitting helps to avoid excessive dependencies on other OCCT toolkits and VTK.

- **IVtk** package does not depend on VTK libraries at all and needs
OCCT libraries only because of collections usage (TKernel library);
- Implementation classes from IVtkOCC package depend on OCCT libraries only and do not need VTK;
- IVtkVTK package depends on VTK libraries only and does not need any OCCT functionality except collections.

Dependencies of VIS packages

Basically, it is enough to use the first three packages in the end user's application (IVtk, IVtkOCC and IVtkVTK) to be able to work with OCCT shapes in VTK viewer. However, IVtkTools package is also provided as a part of the component to make the work more comfortable.
**IVtk package**

**IVtk** package contains the following classes:

- **IVtk_Interface** – Base class for all interfaces of the component. Provides inheritance for Handle (OCCT “smart pointer”) functionality.
- **IVtk_IShape** – Represents a 3D shape of arbitrary nature. Provides its ID property. Implementation of this interface should maintain unique IDs for all visualized shapes. These IDs can be easily converted into original shape objects at the application level.
- **IVtk_IShapeData** – Represents faceted data. Provides methods for adding coordinates and cells (vertices, lines, triangles).
- **IVtk_IShapeMesher** – Interface for faceting, i.e. constructing **IVtk_IShapeData** from **IVtk_IShape** input shape.
- **IVtk_IShapePickerAlgo** – Algorithmic interface for interactive picking of shapes in a scene. Provides methods for finding shapes and their parts (sub-shapes) at a given location according to the chosen selection mode.
- **IVtk_IView** – Interface for obtaining view transformation parameters. It is used by **IVtk_IShapePickerAlgo**.
**IVtkOCC package**

**IVtkOCC** package contains the implementation of classes depending on OCCT:

- **IVtkOCC_Shape** – Implementation of IVtk_IShape interface as a wrapper for TopoDS_Shape.
- **IVtkOCC_ShapeMesher** – Implementation of IVtk_IShapeMesher interface for construction of facets from TopoDS shapes.
- **IVtkOCC_ShapePickerAlgo** – Implementation of interactive picking algorithm. It provides enabling/disabling of selection modes for shapes (IVtk_IShape instances) and picking facilities for a given position of cursor.
- **IVtkOCC_ViewerSelector** – Interactive selector, which implements Pick() methods for the picking algorithm IVtkOCC_ShapePickerAlgo and connects to the visualization layer with help of abstract IView interface.

**IVtkOCC_ViewerSelector** is a descendant of OCCT native SelectMgr_ViewerSelector, so it implements OCCT selection mechanism for IVtkVTK_View (similarly to StdSelectViewerSelector3D which implements SelectMgr_ViewerSelector for OCCT native V3d_View). **IVtkOCC_ViewerSelector** encapsulates all projection transformations for the picking mechanism. These transformations are extracted from vtkCamera instance available via VTK Renderer. **IVtkOCC_ViewerSelector** operates with native OCCT SelectMgr_Selection entities. Each entity represents one selection mode of an OCCT selectable object. ViewerSelector is an internal class, so it is not a part of the public API.

- **IVtkOCC_SelectableObject** – OCCT shape wrapper used in the picking algorithm for computation of selection primitives of a shape for a chosen selection mode.
**IVtkVtk package**

*IVtkVTK* package contains implementation of classes depending on VTK:

- *IVtkVTK_ShapeData* – Implementation of *IVtk_IShapeData* interface for VTK polydata. This class also stores information related to sub-shape IDs and sub-shape mesh type *IVtk_MeshType* (free vertex, shared vertex, free edge, boundary edge, shared edge, wireframe face or shaded face). This information is stored in VTK data arrays for cells.
- *IVtkVTK_View* – Implementation of *IVtk_IView* interface for VTK viewer. This implementation class is used to connect *IVtkOCC_ViewerSelector* to VTK renderer.
**IVtkTools package**

**IVtkTools** package gives you a ready-to-use toolbox of algorithms facilitating the integration of OCCT shapes into visualization pipeline of VTK. This package contains the following classes:

- **IVtkTools_ShapeDataSource** – VTK polygonal data source for OCCT shapes. It inherits `vtkPolyDataAlgorithm` class and provides a faceted representation of OCCT shape for visualization pipelines.
- **IVtkTools_ShapeObject** – Auxiliary wrapper class for OCCT shapes to pass them through pipelines by means of VTK information keys.
- **IVtkTools_ShapePicker** – VTK picker for shape actors. Uses OCCT selection algorithm internally.
- **IVtkTools_DisplayModeFilter** – VTK filter for extracting cells of a particular mesh type according to a given display mode `IVtk_DisplayMode` (Wireframe or Shading).
- **IVtkTools_SubPolyDataFilter** – VTK filter for extracting the cells corresponding to a given set of sub-shape IDs.

Additionally, **IVtkTools** package contains auxiliary methods in **IVtkTools** namespace. E.g. there is a convenience function populating `vtkLookupTable` instances to set up a color scheme for better visualization of sub-shapes.
Using high-level API (simple scenario)

OCCT shape presentation in VTK viewer

To visualize an OCCT topological shape in VTK viewer, it is necessary to perform the following steps:

1. Create `IVtkOCC_Shape` instance (VIS wrapper for OCCT shape) and initialize it with `TopoDS_Shape` object containing the actual geometry:

   ```cpp
   TopoDS_Shape aShape;
   // Initialize aShape variable: e.g. load it from BREP file
   IVtkOCC_Shape::Handle aShapeImpl = new IVtkOCC_Shape(aShape);
   ```

2. Create VTK polygonal data source for the target OCCT topological shape and initialize it with created `IVtkOCC_Shape` instance. At this stage the faceter is implicitly plugged:

   ```cpp
   vtkSmartPointer<IVtkTools_ShapeDataSource> DS =
   vtkSmartPointer<IVtkTools_ShapeDataSource>::New();
   DS->SetShape(aShapeImpl);
   ```

3. Visualize the loaded shape in usual VTK way starting a pipeline from the newly created specific source:

   ```cpp
   vtkSmartPointer<vtkPolyDataMapper> Mapper =
   vtkSmartPointer<vtkPolyDataMapper>::New();
   Mapper->SetInputConnection(aDS->GetOutputPort());
   vtkSmartPointer<vtkActor> Actor =
   vtkSmartPointer<vtkActor>::New();
   ```
It is always possible to access the shape data source from VTK actor by means of dedicated methods from `IVtkTools_ShapeObject` class:

```cpp
IVtkTools_ShapeDataSource* DS =
    IVtkTools_ShapeObject::GetShapeSource(Actor);

IVtkOCC_Shape::Handle occShape =
    IVtkTools_ShapeObject::GetOccShape(Actor);
```

It is also possible to get a shape wrapper from the shape data source:

```cpp
IVtkOCC_Shape::Handle occShape = DS->GetShape();
```
Color schemes

Default OCCT color scheme

To colorize different parts of a shape according to the default OCCT color scheme, it is possible to configure the corresponding VTK mapper using a dedicated auxiliary function of *IVtkTools* namespace:

```
IVtkTools::InitShapeMapper(Mapper);
```

It is possible to get an instance of *vtkLookupTable* class with a default OCCT color scheme by means of the following method:

```
vtkSmartPointer<vtkLookupTable> Table =
    IVtkTools::InitLookupTable();
```

Custom color scheme

To set up application-specific colors for a shape presentation, use *InitShapeMapper* function with an additional argument passing a custom lookup table:

```
IVtkTools::InitShapeMapper(Mapper, Table);
```

Setting custom colors for sub-shapes

It is also possible to bind custom colors to any sub-shape type listed in *IVtk_MeshType* enumeration. For example, to access the color bound to *free edge* entities, the following calls are available in *IVtkTools* namespace:

```
SetLookupTableColor(aLookupTable, MT_FreeEdge, R, G, B);
SetLookupTableColor(aLookupTable, MT_FreeEdge, R, G, B, A);
GetLookupTableColor(aLookupTable, MT_FreeEdge, R, G, B);
```
GetLookupTableColor(aLookupTable, MT_FreeEdge, R, G, B, A);

Here $R, G, B$ are double values of red, green and blue components of a color from the range $[0, 1]$. The optional parameter $A$ stands for the alpha value (the opacity) as a double from the same range $[0, 1]$. By default alpha value is 1, i.e. a color is not transparent.

**Using color scheme of mapper**

As VTK color mapping approach is based on associating scalar data arrays to VTK cells, the coloring of shape components can be turned on/off in the following way:

```cpp
Mapper->ScalarVisibilityOn(); // use colors from lookup table
Mapper->ScalarVisibilityOff(); // use a color of actor’s property
```

For example, the scalar-based coloring can be disabled to bind a single color to the entire VTK actor representing the shape.
Display modes

The output of the shape data source can be presented in wireframe or shading display mode. A specific filter from class `IVtkTools_DisplayModeFilter` can be applied to select the display mode. The filter passes only the cells corresponding to the given mode. The set of available modes is defined by `IVtk_DisplayMode` enumeration.

For example, the shading representation can be obtained in the following way:

```cpp
vtkSmartPointer<IVtkTools_ShapeDataSource> DS = vtkSmartPointer<IVtkTools_ShapeDataSource>::New();

vtkSmartPointer<IVtkTools_DisplayModeFilter> DMFilter = vtkSmartPointer<IVtkTools_DisplayModeFilter>::New();
DMFilter->AddInputConnection(DS->GetOutputPort());
DMFilter->SetDisplayMode(DM_Shading);

vtkSmartPointer<vtkDataSetMapper> M = vtkSmartPointer<vtkDataSetMapper>::New();
M->SetInputConnection(DMFilter->GetOutputPort());
```

By default, the display mode filter works in a wireframe mode.

TIP: to make the shading representation smooth, use additional `vtkPolyDataNormals` filter. This filter must be applied after the display
mode filter.
Interactive selection

*IVtkTools* package provides *IVtkTools_ShapePicker* class to perform selection of OCCT shapes and sub-shapes in VTK viewer and access the picking results. The typical usage of *IVtkTools_ShapePicker* tool consists in the following sequence of actions:

1. Create a picker and set its renderer to your active VTK renderer:

   ```cpp
   vtkSmartPointer<IVtkTools_ShapePicker> aPicker =
   vtkSmartPointer<IVtkTools_ShapePicker>::New();
   aPicker->SetRenderer(aRenderer);
   ```

2. Activate the desired selection mode by choosing the corresponding sub-shape types from *IVtk_SelectionMode* enumeration. For example, the following call allows selection of edges on all selectable shape actors of the renderer:

   ```cpp
   aPicker->SetSelectionMode(SM_Edge);
   ```

   If it is necessary to limit selection by a particular shape actor, one can use the mentioned *SetSelectionMode* method with *IVtk_IShape* handle or *vtkActor* pointer as the first argument:

   ```cpp
   IVtk_IShape::Handle aShape = new
   IVtkOCC_Shape(occShape);
   aPicker->SetSelectionMode(aShape, SM_Edge);  // If shape handle is available
   aPicker->SetSelectionMode(anActor, SM_Edge);  // If shape actor is available
   ```

   Different selection modes can be turned on/off for a picker at the same time independently from each other:

   ```cpp
   aPicker->SetSelectionMode(SM_Edge);
   aPicker->SetSelectionMode(SM_Face);
   ```

   To turn off a selection mode, the additional optional Boolean parameter is used with *false* value, for example:

   ```cpp
   aPicker->SetSelectionMode(aShape, SM_Edge, false);
   ```
3. Call *Pick* method passing the mouse display coordinates:

```cpp
aPicker->Pick(x, y, 0);
```

By default, the renderer passed in the step 1 is used. In order to perform pick operation for another renderer an additional optional parameter can be specified:

```cpp
aPicker->Pick(x, y, 0, aRenderer);
```

4. Obtain the top-level picking results as a collection of picked VTK actors:

```cpp
tvtkActorCollection* anActorCollection = aPicker->GetPickedActors();
```

or as a collection of picked shape IDs:

```cpp
IVtk_ShapeIdList ids = aPicker->GetPickedShapesIds();
```

These methods return a single top picked actor or a shape by default. To get all the picked actors or shapes it is necessary to send “true” value in the optional Boolean parameter:

```cpp
anActorCollection = aPicker->GetPickedActors(true);
ids = aPicker->GetPickedShapesIds(true);
```

5. Obtain the picked sub-shape IDs:

```cpp
IVtk_ShapeIdList subShapeIds = aPicker->GetPickedSubShapesIds(shapeId);
```

This method also returns a single ID of a top-level picked sub-shape and has the same optional Boolean parameter to get all the picked sub-shapes of a shape:

```cpp
subShapeIds = aPicker->GetPickedSubShapesIds(shapeId, true);
```

It should be noted that it is more efficient to create a sole picker instance and feed it with the renderer only once. The matter is that the picking algorithm performs internal calculations each time the renderer or some of its parameters are changed. Therefore, it makes sense to minimize the number of such updates.

OCCT picking algorithm *IVtkTools_ShapePicker* calculates a new transformation matrix for building of projection each time some
parameters of a view are changed. Likewise, the shape selection primitives for each selection mode are built once an appropriate selection mode is turned on for this shape in `SetSelectionMode` method.

WARNING: VIS picker essentially works on the initial topological data structures rather than on the actually visualized actors. This peculiarity allows VIS to take advantage of standard OCCT selection mechanism, but puts strict limitations on the corresponding visualization pipelines. Once constructed, the faceted shape representation should not be morphed or translated anyhow. Otherwise, the picking results will lose their associativity with the source geometry. E.g. you should never use `vtkTransform` filter, but rather apply OCCT isometric transformation on the initial model in order to work on already relocated facet. These limitations are often acceptable for CAD visualization. If not, consider usage of a custom VTK-style picker working on the actually visualized actors.

Selection of sub-shapes

`IVtkTools_SubPolyDataFilter` is a handy VTK filter class which allows extraction of polygonal cells corresponding to the sub-shapes of the initial shape. It can be used to produce a `vtkPolyData` object from the input `vtkPolyData` object, using selection results from `IVTkTools_ShapePicker` tool.

For example, sub-shapes can be represented in VTK viewer in the following way:

```cpp
// Load a shape into data source (see 3.1)
...
vtkSmartPointer<IVtkTools_ShapeDataSource> DS =
```
DS->SetShape(shapeImpl);

// Create a new sub-polydata filter for sub-shapes filtering
vtkSmartPointer<IVtkTools_SubPolyDataFilter> subShapesFilter = IVtkTools_SubPolyDataFilter::New();

// Set a shape source as an input of the sub-polydata filter
subShapesFilter->SetInputConnection(DS->GetOutputPort());

// Get all picked sub-shapes ids of the shape from a picker (see 3.4)
IVtk_ShapeIdList subShapeIds = aPicker->GetPickedSubShapesIds(ds->GetId(), true);

// Set ids to the filter to pass only picked sub-shapes
subShapesFilter->SetData(subShapeIds);
subShapesFilter->Modified();

// Output the result into a mapper
vtkSmartPointer<vtkPolyDataMapper> amapper = vtkPolyDataMapper::New();
amapper->AddInputConnection(subShapesFilter->GetOutputPort());
...
Using of low-level API (advanced scenario)

Shape presentation

The usage of low-level tools is justified in cases when the utilities from IVtkTools are not enough.

The low-level scenario of VIS usage in VTK pipeline is shown in the figure below. The Mesher component produces shape facet (VTK polygonal data) using implementation of IShapeData interface. Then result can be retrieved from this implementation as a vtkPolyData instance.

![Diagram](image)

Low-level VIS usage with VTK

420

The visualization pipeline for OCCT shape presentation can be initialized as follows:

1. Create an instance of IShape class initialized by OCCT topological shape:

   ```cpp
   TopoDS_Shape aShape;
   // Load or create a TopoDS_Shape in the variable a Shape
   ...
   IVtkOCC_Shape::Handle aShapeImpl = new IVtkOCC_Shape(aShape);
   ```

2. Create an empty instance of IShapeData implementation for VTK:

   ```cpp
   IVtk_IShapeData::Handle aDataImpl = new IVtkVTK_ShapeData();
   ```
3 Create an instance of *IShapeMesher* implementation for OCCT (any faceter can be used at this stage):

```cpp
IVtk_IShapeMesher::Handle aMesher = new IVtkOCC_ShapeMesher();
```

4 Triangulate the OCCT topological shape by means of the Mesher and access the result:

```cpp
aMesher->Build (aShapeImpl, aDataImpl);
vtkPolyData* aPolyData = aDataImpl->GetVtkPolyData();
```

The resulting *vtkPolyData* instance can be used for initialization of VTK pipelines. *IVtkVTK.ShapeData* object is used to keep and pass the mapping between sub-shapes, their mesh types and the resulting mesh cells through a pipeline. It stores sub-shape IDs and mesh type in VTK data arrays for each generated cell. As a result, the generated VTK cells get the following data arrays populated:

- **SUBSHAPE_IDS** - array of `vtkIdTypeArray` type. It contains the shape IDs the corresponding cells were generated for. The name of this array is defined in `ARRNAME_SUBSHAPE_IDS` constant of `IVtkVTK.ShapeData` class.
- **MESH_TYPES** - array of `vtkShortArray` type. It contains the type tags of the shape parts the corresponding cells were generated for. The name of this array is defined in `ARRNAME_MESH_TYPES` constant of `IVtkVTK.ShapeData` class.
Usage of OCCT picking algorithm

It is possible to create a custom VTK picker for interactive selection of OCCT 3D shapes using an instance of the picking algorithm IVtk_IShapePickerAlgo.

Picking algorithm uses an instance of viewer selector (OCCT term), which manages picking along with activation and deactivation of selection modes. VIS component implements OCCT selection principle in IVtkOCC_ShapePickerAlgo and IVtkOCC_ViewerSelector classes. IVtkOCC_ViewerSelector is an internal class that implements OCCT selection mechanism applied in IVtkVTK_View.

IVtkOCC_ShapePickerAlgo has to be used to activate/deactivate selection modes for shapes IVtk_IShape. IVtkOCC_ShapePickerAlgo is the implementation of IVtk_IShapePickerAlgo interface.

The typical usage of IVtk_IShapePickerAlgo consists in the following sequence of actions:

1. Create an instance of the picker class:

   ```cpp
   IVtkOCC_ShapePickerAlgo::Handle Picker = new IVtkOCC_ShapePickerAlgo();
   ```

2. Set an instance of IVtk_IView class to the algorithm in order to define the viewer parameters:

   ```cpp
   IVtkVTK_View::Handle View = new IVtkVTK_View(Renderer);
   Picker->SetView(View);
   ```

3. Activate the desired selection modes using values from IVtk_SelectionMode enumeration. For example, the following call allows selection of edges:

   ```cpp
   TopoDS_Shape aShape;
   // Load or create a TopoDS_Shape in the variable aShape
   ...
   IVtk_IShape::Handle shapeImpl = new IVtkOCC_Shape(aShape);
   ```
myOccPickerAlgo->SetSelectionMode(occShape, SM_Edge);

Different selection modes can be turned on/off for a picker at the same time independently from each other. To turn off a selection mode the additional optional Boolean parameter is used with *false* value, for example:

```cpp
myOccPickerAlgo->SetSelectionMode(occShape, SM_Edge, false);
```

4. Call *Pick* method passing the mouse coordinates:

```cpp
myOccPickerAlgo->Pick(x, y);
```

5. Obtain top-level picking results as IDs of the picked top-level shapes:

```cpp
IVtk_ShapeIdList ids = myOccPickerAlgo->ShapesPicked();
```

6. Obtain IDs of the picked sub-shapes:

```cpp
IVtk_ShapeIdList subShapeIds
    = myOccPickerAlgo->SubShapesPicked(shapeId);
```
DRAW Test Harness

TKIVtkDraw toolkit contains classes for embedding VIS functionality into DRAW Test Harness with possibility of simple interactions, including detection and highlighting.

- IVtkDraw_HighlightAndSelectionPipeline – Creates VTK pipeline with OCCT shape data source and properly initialized VIS filters.
- IVtkDraw_Interactor – Controls simple interactive actions, such as detection and selection of the displayed shapes.
IGES Support

Table of Contents

- Introduction
- Reading IGES
  - Procedure
  - Domain covered
    - Translatable entities
    - Attributes
    - Administrative data
  - Description of the process
    - Loading the IGES file
    - Checking the IGES file
    - Setting translation parameters
    - Selecting entities
    - Performing the IGES file translation
    - Getting the translation results
- Mapping of IGES entities to Open CASCADE Technology shapes
  - Points
Curves
Surfaces
Boundary Representation
Solid Entities
Structure Entities
Subfigures
Transformation Matrix
Messages
Tolerance management
Values used for tolerances during reading IGES
Initial setting of tolerances in translating objects
Transfer process
Code architecture
Example
Writing IGES
Procedure
Domain covered
Description of the process
Initializing the process
Setting the translation parameters
Performing the Open CASCADE Technology shape translation
Writing the IGES file
Mapping Open CASCADE Technology shapes to IGES entities

- Curves
- Surfaces
- Topological entities -- Translation in Face mode
- Topological entities -- Translation in BRep mode

- Tolerance management
- Setting resolution in an IGES file

- Code architecture
- Graph of calls
- Example

Using XSTEPDRAW

- Setting interface parameters
- Reading IGES files
- Analyzing the transferred data
  - Checking file contents
  - Estimating the results of reading IGES

- Writing an IGES file
- Reading from and writing to IGES
  - Reading from IGES
  - Writing to IGES
Introduction

The IGES interface reads IGES files and translates them to Open CASCADE Technology models. The interface is able to translate one entity, a group of entities or a whole file. Before beginning a translation, you can set a range of parameters to manage the translation process. If you like, you can also check file consistency before translation. The IGES interface also translates OCCT models to IGES files.

Other kinds of data such as colors and names can be read or written with the help of XDE tools IGESCAFControl_Reader and IGESCAFControl_Writer.

Please, note:

- an IGES model is an IGES file that has been loaded into memory.
- an IGES entity is an entity in the IGES normal sense.
- a root entity is the highest level entity of any given type, e.g. type 144 for surfaces and type 186 for solids. Roots are not referenced by other entities.

This manual mainly explains how to convert an IGES file to an Open CASCADE Technology (OCCT) shape and vice versa. It provides basic documentation on conversion. For advanced information on conversion, see our E-learning & Training offerings.

IGES files produced in accordance with IGES standard versions up to and including version 5.3 can be read. IGES files that are produced by this interface conform to IGES version 5.3 (Initial Graphics Exchange Specification, IGES 5.3. ANS US PRO/IPO-100-1996).

This manual principally deals with two OCCT classes:

- The Reader class, which loads IGES files and translates their contents to OCCT shapes,
- The Writer class, which translates OCCT shapes to IGES entities and then writes these entities to IGES files.

File translation is performed in the programming mode, via C++ calls, and
the resulting OCCT objects are shapes.

All definitions in IGES version 5.3 are recognized but only 3D geometric entities are translated. When the processor encounters data, which is not translated, it ignores it and writes a message identifying the types of data, which was not handled. This message can be written either to a log file or to screen output.

**Shape Healing** toolkit provides tools to heal various problems, which may be encountered in translated shapes, and to make them valid in Open CASCADE. The Shape Healing is smoothly connected to IGES translator using the same API, only the names of API packages change.
Reading IGES

Procedure

You can translate an IGES file to an OCCT shape by following the steps below:

1. Load the file,
2. Check file consistency,
3. Set the translation parameters,
4. Perform the file translation,
5. Fetch the results.
Domain covered

Translatable entities

The types of IGES entities, which can be translated, are:

- Points
- Lines
- Curves
- Surfaces
- B-Rep entities
- Structure entities (groups). Each entity in the group outputs a shape. There can be a group of groups.
- Subfigures. Each entity defined in a sub-figure outputs a shape
- Transformation Matrix.

Note that all non-millimeter length unit values in the IGES file are converted to millimeters.

Attributes

Entity attributes in the Directory Entry Section of the IGES file (such as layers, colors and thickness) are translated to Open CASCADE Technology using XDE.

Administrative data

Administrative data, in the Global Section of the IGES file (such as the file name, the name of the author, the date and time a model was created or last modified) is not translated to Open CASCADE Technology. Administrative data can, however, be consulted in the IGES file.
Description of the process

Loading the IGES file

Before performing any other operation, you have to load the file using the syntax below.

```plaintext
IGESControl_Reader reader;
IFSelect_ReturnStatus stat =
    reader.ReadFile("filename.igs");
```

The loading operation only loads the IGES file into computer memory; it does not translate it.

Checking the IGES file

This step is not obligatory. Check the loaded file with:

```plaintext
Standard_Boolean ok = reader.Check(Standard_True);
```

The variable “ok is True” is returned if no fail message was found; “ok is False” is returned if there was at least one fail message.

```plaintext
reader.PrintCheckLoad (failsonly, mode);
```

Error messages are displayed if there are invalid or incomplete IGES entities, giving you information on the cause of the error.

```plaintext
Standard_Boolean failsonly = Standard_True or Standard_False;
```

If you give True, you will see fail messages only. If you give False, you will see both fail and warning messages.

Your analysis of the file can be either message-oriented or entity-oriented. Choose your preference with `IFSelect_PrintCount mode = IFSelect_xxx`, where `xxx` can be any of the following:
- *ItemsByEntity* gives a sequential list of all messages per IGES entity.
- *CountByItem* gives the number of IGES entities with their types per message.
- *ShortByItem* gives the number of IGES entities with their types per message and displays rank numbers of the first five IGES entities per message.
- *ListByItem* gives the number of IGES entities with their type and rank numbers per message.
- *EntitiesByItem* gives the number of IGES entities with their types, rank numbers and Directory Entry numbers per message.

### Setting translation parameters

The following parameters can be used to translate an IGES file to an OCCT shape. If you give a value that is not within the range of possible values, it will be ignored.

**read.iges.bspline.continuity**

manages the continuity of BSpline curves (IGES entities 106, 112 and 126) after translation to Open CASCADE Technology (Open CASCADE Technology requires that the curves in a model be at least C1 continuous; no such requirement is made by IGES).

- 0: no change; the curves are taken as they are in the IGES file. C0 entities of Open CASCADE Technology may be produced.
- 1: if an IGES BSpline, Spline or CopiousData curve is C0 continuous, it is broken down into pieces of C1 continuous Geom_BSplineCurve.
- 2: This option concerns IGES Spline curves only. IGES Spline curves are broken down into pieces of C2 continuity. If C2 cannot be ensured, the Spline curves will be broken down into pieces of C1 continuity.

Read this parameter with:

```plaintext
Standard_Integer ic = Interface_Static::IVal("read.iges.bspline.continuity");
```
Modify this value with:

```cpp
if (!Interface_Static::SetIVal("read.iges.bspline.continuity",2))
  .. error ..;
```

Default value is 1.

This parameter does not change the continuity of curves that are used in the construction of IGES BRep entities. In this case, the parameter does not influence the continuity of the resulting OCCT curves (it is ignored).

**read.precision.mode**

reads the precision value.

- File (0) the precision value is read in the IGES file header (default).
- User (1) the precision value is that of the read.precision.val parameter.

Read this parameter with:

```cpp
Standard_Integer ic = Interface_Static::IVal("read.precision.mode");
```

Modify this value with:

```cpp
if (!Interface_Static::SetIVal("read.precision.mode",1))
  .. error ..;
```

Default value is *File (0)*.

**read.precision.val**

User defined precision value. This parameter gives the precision for shape construction when the read.precision.mode parameter value is 1. By default it is 0.0001, but can be any real positive (non null) value.
This value is in the measurement unit defined in the IGES file header.

Read this parameter with:

```cpp
Standard_Real rp =
    Interface_Static::RVal("read.precision.val");
```

Modify this parameter with:

```cpp
if (!Interface_Static::SetRVal
    ("read.precision.val",0.001))
.. error ..;
```

Default value is 0.0001.

The value given to this parameter is a target value that is applied to `TopoDS_Vertex`, `TopoDS_Edge` and `TopoDS_Face` entities. The processor does its best to reach it. Under certain circumstances, the value you give may not be attached to all of the entities concerned at the end of processing. IGES-to-OCCT translation does not improve the quality of the geometry in the original IGES file. This means that the value you enter may be impossible to attain the given quality of geometry in the IGES file.

Value of tolerance used for computation is calculated by multiplying the value of `read.precision.val` and the value of coefficient of transfer from the file units to millimeters.

**read.maxprecision.mode**

defines the mode of applying the maximum allowed tolerance. Its possible values are:

- **Preferred(0)** maximum tolerance is used as a limit but sometimes it can be exceeded (currently, only for deviation of a 3D curve of an edge from its pcurves and from vertices of such edge) to ensure shape validity;
- **Forced(1)** maximum tolerance is used as a rigid limit, i.e. it can not be exceeded and, if this happens, tolerance is trimmed to suit the maximum-allowable value.
Read this parameter with:

```plaintext
Standard_Integer mv = Interface_Static::IVal("read.maxprecision.mode") ;
```

Modify this parameter with:

```plaintext
if (!Interface_Static::SetIVal("read.maxprecision.mode",1))
   .. error ..;
```

Default value is *Preferred (0)*.

**read.maxprecision.val**

defines the maximum allowable tolerance (in mm) of the shape. It should be not less than the basis value of tolerance set in processor (either Resolution from the file or *read.precision.val*). Actually, the maximum between *read.maxprecision.val* and basis tolerance is used to define maximum allowed tolerance. Read this parameter with:

```plaintext
Standard_Real rp = Interface_Static::RVal("read.maxprecision.val");
```

Modify this parameter with:

```plaintext
if (!Interface_Static::SetRVal("read.maxprecision.val",0.1))
   .. error ..;
```

Default value is 1.

**read.stdsameparameter.mode**

defines the using of *BRepLib::SameParameter*. Its possible values are:

- 0 (Off) – *BRepLib::SameParameter* is not called,
1 (On) – \textit{BRepLib::SameParameter} is called. \textit{BRepLib::SameParameter} is used through \textit{ShapeFix\_Edge::SameParameter}. It ensures that the resulting edge will have the lowest tolerance taking pcurves either unmodified from the IGES file or modified by \textit{BRepLib::SameParameter}. Read this parameter with:

```plaintext
Standard\_Integer\ mv = Interface\_Static::IVal("read.stdsameparameter.mode");
```

Modify this parameter with:

```plaintext
if (!Interface\_Static::SetIVal("read.stdsameparameter.mode",1))
  .. error ..;
```

Default value is 0 (Off).

\textit{read.surfacecurve.mode}

preference for the computation of curves in case of 2D/3D inconsistency in an entity which has both 2D and 3D representations.

Here we are talking about entity types 141 (Boundary), 142 (CurveOnSurface) and 508 (Loop). These are entities representing a contour lying on a surface, which is translated to a \textit{TopoDS\_Wire}, formed by \textit{TopoDS\_Edges}. Each \textit{TopoDS\_Edge} must have a 3D curve and a 2D curve that reference the surface.

The processor also decides to re-compute either the 3D or the 2D curve even if both curves are translated successfully and seem to be correct, in case there is inconsistency between them. The processor considers that there is inconsistency if any of the following conditions is satisfied:

- the number of sub-curves in the 2D curve is different from the number of sub-curves in the 3D curve. This can be either due to different numbers of sub-curves given in the IGES file or because of splitting of curves during translation.
- 3D or 2D curve is a Circular Arc (entity type 100) starting and ending in the same point (note that this case is incorrect according to the IGES standard).
The parameter \textit{read.surfacecurve.mode} defines which curve (3D or 2D) is used for re-computing the other one:

- \textit{Default(0)} use the preference flag value in the entity's Parameter Data section. The flag values are:
  - 0: no preference given,
  - 1: use 2D for 142 entities and 3D for 141 entities,
  - 2: use 3D for 142 entities and 2D for 141 entities,
  - 3: both representations are equally preferred.
- \textit{2DUse_Preferred (2)}: the 2D is used to rebuild the 3D in case of their inconsistency,
- \textit{2DUse_Forced (-2)}: the 2D is always used to rebuild the 3D (even if 3D is present in the file),
- \textit{3DUse_Preferred (3)}: the 3D is used to rebuild the 2D in case of their inconsistency,
- \textit{3DUse_Forced (-3)}: the 3D is always used to rebuild the 2D (even if 2D is present in the file),

If no preference is defined (if the value of \textit{read.surfacecurve.mode} is \textit{Default} and the value of the preference flag in the entity's Parameter Data section is 0 or 3), an additional analysis is performed.

The 3D representation is preferred to the 2D in two cases:

- if 3D and 2D contours in the file have a different number of curves,
- if the 2D curve is a Circular Arc (entity type 100) starting and ending in the same point and the 3D one is not.

In any other case, the 2D representation is preferred to the 3D.

If either a 3D or a 2D contour is absent in the file or cannot be translated, then it is re-computed from another contour. If the translation of both 2D and 3D contours fails, the whole curve (type 141 or 142) is not translated. If this curve is used for trimming a face, the face will be translated without this trimming and will have natural restrictions.

Read this parameter with:

```cpp
Standard_Integer ic = Interface_Static::IVal("read.surfacecurve.mode") ;
```
Modify this value with:

```cpp
if (!Interface_Static::SetIVal
    ("read.surfacecurve.mode",3))
  .. error ..;
```

Default value is Default (0).

**read.encoderegularity.angle**

This parameter is used within the `BRepLib::EncodeRegularity()` function which is called for a shape read from an IGES or a STEP file at the end of translation process. This function sets the regularity flag of an edge in a shell when this edge is shared by two faces. This flag shows the continuity, which these two faces are connected with at that edge.

Read this parameter with:

```cpp
Standard_Real era = Interface_Static::RVal("read.encoderegularity.angle");
```

Modify this parameter with:

```cpp
if (!Interface_Static::SetRVal("read.encoderegularity.angle",0.1))
  .. error ..;
```

Default value is 0.01.

**read.iges.bspline.approxd1.mode**

This parameter is obsolete (it is rarely used in real practice). If set to True, it affects the translation of bspline curves of degree 1 from IGES: these curves (which geometrically are polylines) are split by duplicated points, and the translator attempts to convert each of the obtained parts to a bspline of a higher continuity.

Read this parameter with:
Standard_Real bam = 
    Interface_Static::CVal("read.iges.bspline.approx
d1.mode");

Modify this parameter with:

```cpp
if (!Interface_Static::SetRVal
    ("read.encoderegularity.angle","On"))
  .. error ..;
```

Default value is Off.

**read.iges.resource.name and read.iges.sequence**

These two parameters define the name of the resource file and the name of the sequence of operators (defined in that file) for Shape Processing, which is automatically performed by the IGES translator. The Shape Processing is a user-configurable step, which is performed after the translation and consists in application of a set of operators to a resulting shape. This is a very powerful tool allowing to customize the shape and to adapt it to the needs of a receiving application. By default, the sequence consists of a single operator *ShapeFix* that calls Shape Healing from the IGES translator.

Please find an example of the resource file for IGES (which defines parameters corresponding to the sequence applied by default, i.e. if the resource file is not found) in the Open CASCADE Technology installation, by the path `CASROOT%/src/XSTEPResource/IGES`.

In order for the IGES translator to use that file, you have to define the environment variable *CSF_IGESDefaults*, which should point to the directory where the resource file resides. Note that if you change parameter *read.iges.resource.name*, you should change the name of the resource file and the name of the environment variable correspondingly. The variable should contain a path to the resource file.

Default values:

- *read.iges.resource.name* – IGES,
- *read.iges.sequence* – FromIGES.
xstep.cascade.unit

This parameter defines units to which a shape should be converted when translated from IGES or STEP to CASCADE. Normally it is MM; only those applications that work internally in units other than MM should use this parameter.

Default value is MM.

Selecting entities

A list of entities can be formed by invoking the method

IGESControl_Reader::GiveList.

```cpp
Handle(TColStd_HSequenceOfTransient) list = 
    reader.GiveList();
```

Several predefined operators can be used to select a list of entities of a specific type. To make a selection, you use the method

IGESControl_Reader::GiveList with the selection type in quotation marks as an argument. You can also make cumulative selections. For example, you would use the following syntax:

1. Requesting the faces in the file:
   ```cpp
   faces = Reader.GiveList("iges-faces");
   ```
2. Requesting the visible roots in the file:
   ```cpp
   visibles = Reader.GiveList(iges-visible-roots);
   ```
3. Requesting the visible faces:
   ```cpp
   visfac = Reader.GiveList(iges-visible-roots,faces);
   ```

Using a signature, you can define a selection dynamically, filtering the string by means of a criterion. When you request a selection using the method GiveList, you can give either a predefined selection or a selection by signature. You make your selection by signature using the predefined signature followed by your criterion in parentheses as shown in the example below. The syntaxes given are equivalent to each other.

```cpp
faces = Reader.GiveList("xst-
```
type(SurfaceOfRevolution);
faces = Reader.GiveList("iges-type(120)");

You can also look for:

- values returned by your signature which match your criterion exactly
  
  ```
  faces = Reader.GiveList("xst-type(=SurfaceOfRevolution));
  ```

- values returned by your signature which do not contain your criterion
  
  ```
  faces = Reader.GiveList("xst-type(!SurfaceOfRevolution));
  ```

- values returned by your signature which do not exactly match your criterion.
  
  ```
  faces = Reader.GiveList("xst-type(!=SurfaceOfRevolution)");
  ```

**List of predefined operators that can be used:**

- **xst-model-all** – selects all entities.
- **xst-model-roots** – selects all roots.
- **xst-transferrable-all** – selects all translatable entities.
- **xst-transferrable-roots** – selects all translatable roots (default).
- **xst-sharing + <selection>** – selects all entities sharing at least one entity selected by <selection>.
- **xst-shared + <selection>** – selects all entities shared by at least one entity selected by <selection>.
- **iges-visible-roots** – selects all visible roots, whether translatable or not.
- **iges-visible-transf-roots** – selects all visible and translatable roots.
- **iges-blanked-roots** – selects all blank roots, whether translatable or not.
- **iges-blanked-transf-roots** – selects all blank and translatable roots.
- **iges-status-independant** – selects entities whose IGES Subordinate Status = 0.
- **iges-bypass-group** – selects all root entities. If a root entity is a group (402/7 or 402/9), the entities in the group are selected.
- **iges-bypass-subfigure** – selects all root entities. If a root entity is a subfigure definition (308), the entities in the subfigure definition are
selected.

- *iges-bypass-group-subfigure* – selects all root entities. If a root entity is a group (402/7 or 402/9) or a subfigure definition (308), the entities in the group and in the subfigure definition are selected.
- *iges-curves-3d* – selects 3D curves, whether they are roots or not (e.g. a 3D curve on a surface).
- *iges-basic-geom* – selects 3D curves and untrimmed surfaces.
- *iges-faces* – selects face-supporting surfaces (trimmed or not).
- *iges-surfaces* – selects surfaces not supporting faces (i.e. with natural bounds).
- *iges-basic-curves-3d* – selects the same entities as *iges-curves-3d*. Composite Curves are broken down into their components and the components are selected.

**Performing the IGES file translation**

Perform translation according to what you want to translate:

1. Translate an entity identified by its rank with:
   ```
   Standard_Boolean ok = reader.Transfer(rank);
   ```

2. Translate an entity identified by its handle with:
   ```
   Standard_Boolean ok = reader.TransferEntity(ent);
   ```

3. Translate a list of entities in one operation with:
   ```
   Standard_Integer nbtrans = reader.TransferList(list);
   reader.IsDone();
   ```

   where *nbtrans* returns the number of items in the list that produced a shape and *reader.IsDone()* indicates whether at least one entity was translated.

4. Translate a list of entities, entity by entity:
   ```
   Standard_Integer i,nb = list-Length();
   for (i = 1; i <= nb; i ++) {
     Handle(Standard_Transient) ent = list-Value(i);
     Standard_Boolean OK = reader.TransferEntity(ent);
   }
   ```
5. Translate the whole file (all entities or only visible entities) with:

```plaintext
Standard_Boolean onlyvisible = Standard_True or Standard_False;
reader.TransferRoots(onlyvisible)
```

## Getting the translation results

Each successful translation operation outputs one shape. A series of translations gives a series of shapes. Each time you invoke `TransferEntity, Transfer` or `Transferlist`, their results are accumulated and `NbShapes` increases. You can clear the results (Clear function) between two translation operations, if you do not do this, the results from the next translation will be added to the accumulation. `TransferRoots` operations automatically clear all existing results before they start.

```plaintext
Standard_Integer nbs = reader.NbShapes();
```

returns the number of shapes recorded in the result.

```plaintext
TopoDS_Shape shape = reader.Shape(num);
```

returns the result `num`, where `num` is an integer between 1 and `NbShapes`.

```plaintext
TopoDS_Shape shape = reader.Shape();
```

returns the first result in a translation operation.

```plaintext
TopoDS_Shape shape = reader.OneShape();
```

returns all results in a single shape which is:

- a null shape if there are no results,
- in case of a single result, a shape that is specific to that result,
- a compound that lists the results if there are several results.

```plaintext
reader.Clear();
```

erases the existing results.

```plaintext
reader.PrintTransferInfo (failsonly, mode);
```
displays the messages that appeared during the last invocation of *Transfer* or *TransferRoots*.

If *failonly* is *IFSelect_FailOnly*, only fail messages will be output, if it is *IFSelect_FailAnd Warn*, all messages will be output. Parameter “mode” can have *IFSelect_xxx* values where *xxx* can be:

- **GeneralCount** – gives general statistics on the transfer (number of translated IGES entities, number of fails and warnings, etc)
- **CountByItem** – gives the number of IGES entities with their types per message.
- **ListByItem** – gives the number of IGES entities with their type and DE numbers per message.
- **ResultCount** – gives the number of resulting OCCT shapes per type.
- **Mapping** – gives mapping between roots of the IGES file and the resulting OCCT shape per IGES and OCCT type.
Mapping of IGES entities to Open CASCADE Technology shapes

*NOTE* that IGES entity types that are not given in the following tables are not translatable.

### Points

<table>
<thead>
<tr>
<th>IGES entity type</th>
<th>CASCADE shape</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>116: Point</td>
<td>TopoDS_Vertex</td>
<td></td>
</tr>
</tbody>
</table>

### Curves

Curves, which form the 2D of face boundaries, are translated as *Geom2D_Curves* (Geom2D circles, etc.).

<table>
<thead>
<tr>
<th>IGES entity type</th>
<th>CASCADE shape</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>100: Circular Arc</td>
<td>TopoDS_Edge</td>
<td>The geometrical support is a <em>Geom_Circle</em> or a <em>Geom_TrimmedCurve</em> (if the arc is not closed).</td>
</tr>
<tr>
<td>102: Composite Curve</td>
<td>TopoDS_Wire</td>
<td>The resulting shape is always a <em>TopoDS_Wire</em> that is built from a set of <em>TopoDS_Edges</em>. Each <em>TopoDS_Edge</em> is connected to the preceding and to the following edge by a common <em>TopoDS_Vertex</em>.</td>
</tr>
<tr>
<td>104: Conic Arc</td>
<td>TopoDS_Edge</td>
<td>The geometric support depends on whether the IGES entity's form is 0 (<em>Geom_Circle</em>), 1 (<em>Geom_Ellipse</em>), 2 (<em>Geom_Hyperbola</em>), or 3 (<em>Geom_Parabola</em>). A <em>Geom_TrimmedCurve</em> is output if the arc is not closed.</td>
</tr>
<tr>
<td>IGES Entity</td>
<td>OCCT Shape(s)</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>106: Copious Data</td>
<td>TopoDS_Edge or TopoDS_Wire</td>
<td>IGES entity Copious Data (type 106, forms 1-3) is translated just as the IGES entities Linear Path (106/11-13) and the Simple Closed Planar Curve (106/63). Vectors applying to forms other than 11,12 or 63 are ignored. The <code>Geom_BSplineCurve</code> (geometrical support) has C0 continuity. If the Copious Data has vectors (DataType = 3) they will be ignored.</td>
</tr>
<tr>
<td>110: Line</td>
<td>TopoDS_Edge</td>
<td>The supporting curve is a <code>Geom_TrimmedCurve</code> whose basis curve is a <code>Geom_Line</code>.</td>
</tr>
<tr>
<td>112: Parametric Spline Curve</td>
<td>TopoDS_Edge or TopoDS_Wire</td>
<td>The geometric support is a <code>Geom_BSplineCurve</code>.</td>
</tr>
<tr>
<td>126: BSpline Curve</td>
<td>TopoDS_Edge or TopoDS_Wire</td>
<td>The resulting shape is a <code>TopoDS_Edge</code> or a <code>TopoDS_Wire</code> (depending on the translation of the basis curve) whose geometrical support is a <code>Geom_OffsetCurve</code> built from a basis <code>Geom_Curve</code>. Limitation: The IGES Offset Type value must be 1.</td>
</tr>
<tr>
<td>130: Offset Curve</td>
<td>TopoDS_Edge or TopoDS_Wire</td>
<td>Same behavior as for the Curve On Surface (see below). The translation of a non-referenced Boundary IGES entity in a <code>BoundedSurface</code> IGES entity outputs a <code>TopoDS_Edge</code> or a <code>TopoDS_Wire</code> with a <code>Geom_Curve</code>.</td>
</tr>
<tr>
<td>141: Boundary</td>
<td>TopoDS_Wire</td>
<td>Each <code>TopoDS_Edge</code> is defined by a 3D curve and by a 2D curve that references the surface.</td>
</tr>
</tbody>
</table>

The type of OCCT shapes (either `TopoDS_Edges` or `TopoDS_Wires`) that result from the translation of IGES entities 106, 112 and 126 depends on the continuity of the curve in the IGES file and the value of the
Surfaces

Translation of a surface outputs either a `TopoDS_Face` or a `TopoDS_Shell`. If a `TopoDS_Face` is output, its geometrical support is a `Geom_Surface` and its outer and inner boundaries (if it has any) are `TopoDS_Wires`.

<table>
<thead>
<tr>
<th>IGES entity type</th>
<th>CASCADE shape</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>108: Plane</td>
<td>TopoDS_Face</td>
<td>The geometrical support for the <code>TopoDS_Face</code> is a <code>Geom_Plane</code> and the orientation of its <code>TopoDS_Wire</code> depends on whether it is an outer <code>TopoDS_Wire</code> or whether it is a hole.</td>
</tr>
<tr>
<td>114: Parametric Spline Surface</td>
<td>TopoDS_Face</td>
<td>The geometrical support of a <code>TopoDS_Face</code> is a <code>Geom_BSplineSurface</code>.</td>
</tr>
<tr>
<td>118: Ruled Surface</td>
<td>TopoDS_Face or TopoDS_Shell</td>
<td>The translation of a Ruled Surface outputs a <code>TopoDS_Face</code> if the profile curves become <code>TopoDS_Edges</code>, or a <code>TopoDS_Shell</code> if the profile curves become <code>TopoDS_Wires</code>. Limitation: This translation cannot be completed when these two <code>TopoDS_Wires</code> are oriented in different directions.</td>
</tr>
<tr>
<td>120: Surface Of Revolution</td>
<td>TopoDS_Face or TopoDS_Shell</td>
<td>The translation of a Surface Of Revolution outputs: a <code>TopoDS_Face</code> if the generatrix becomes a <code>TopoDS_Edge</code>, a <code>TopoDS_Shell</code> if the generatrix becomes a <code>TopoDS_Wire</code>. The geometrical support may be: <code>Geom_CylindricalSurface</code>, <code>Geom_ConicalSurface</code>, <code>Geom_SphericalSurface</code>, <code>Geom_ToroidalSurface</code> or a ...</td>
</tr>
<tr>
<td>122: Tabulated Cylinder</td>
<td>TopoDS_Face or TopoDS_Shell</td>
<td>The translation outputs a <code>TopoDS_Face</code> if the base becomes a <code>TopoDS_Edge</code> or a <code>TopoDS_Shell</code> if the base becomes a <code>TopoDS_Wire</code>. The geometrical support may be <code>Geom_Plane</code>, <code>Geom_CylindricalSurface</code> or a <code>Geom_SurfaceOfLinearExtrusion</code> depending on the result of the CASCADE computation (based on the generatrix type). The <code>Geom_Surface</code> geometrical support is limited according to the generatrix.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>128: BSpline Surface</td>
<td>TopoDS_Face</td>
<td>The geometrical support of the <code>TopoDS_Face</code> is a <code>Geom_BsplineSurface</code>.</td>
</tr>
<tr>
<td>140: Offset Surface</td>
<td>TopoDS_Face</td>
<td>The translation of an Offset Surface outputs a <code>TopoDS_Face</code> whose geometrical support is a <code>Geom_OffsetSurface</code>. Limitations: For OCCT algorithms, the original surface must be C1-continuous so that the <code>Geom_OffsetSurface</code> can be created. If the basis surface is not C1-continuous, its translation outputs a <code>TopoDS_Shell</code> and only the first <code>TopoDS_Face</code> in the <code>TopoDS_Shell</code> is offset.</td>
</tr>
<tr>
<td>143: Bounded Surface</td>
<td>TopoDS_Face or TopoDS_Shell</td>
<td>If the basis surface outputs a <code>TopoDS_Shell</code> (that has more than one <code>TopoDS_Face</code>), the IGES boundaries are not translated. Limitations: If the bounding curves define holes, natural bounds are not created. If the orientation of the contours is wrong, it is not corrected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For the needs of interface processing, the basis surface must be a face. Shells are only processed if they are single-face. The</td>
</tr>
</tbody>
</table>
144: Trimmed Surface  
TopoDS_Face or TopoDS_Shell  
contours (wires that are correctly oriented according to the definition of the IGES 142: Curve On Surface entity) are added to the face that is already created. If the orientation of the contours is wrong, it is corrected.

190: Plane Surface  
TopoDS_Face  
This type of IGES entity can only be used in BRep entities in place of an IGES 108 type entity. The geometrical support of the face is a Geom_Plane.

### Boundary Representation Solid Entities

<table>
<thead>
<tr>
<th>IGES entity type</th>
<th>CASCADE shape</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>186: ManifoldSolid</td>
<td>TopoDS_Solid</td>
<td></td>
</tr>
<tr>
<td>514: Shell</td>
<td>TopoDS_Shell</td>
<td></td>
</tr>
<tr>
<td>510: Face</td>
<td>TopoDS_Face</td>
<td>This is the lowest IGES entity in the BRep structure that can be specified as a starting point for translation.</td>
</tr>
<tr>
<td>508: Loop</td>
<td>TopoDS_Wire</td>
<td></td>
</tr>
<tr>
<td>504: Edge List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>502: Vertex List</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Structure Entities

<table>
<thead>
<tr>
<th>IGES entity type</th>
<th>CASCADE shape</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>402/1: Associativity Instance:</td>
<td>TopoDS_Compound</td>
<td></td>
</tr>
<tr>
<td>Group with back pointers</td>
<td>TopoDS_Compound</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>402/7: Associativity Instance: Group without back pointers</td>
<td>TopoDS_Compound</td>
<td></td>
</tr>
<tr>
<td>402/9: Associativity Instance: Single Parent</td>
<td>TopoDS_Face</td>
<td></td>
</tr>
</tbody>
</table>

The translation of a SingleParent entity is only performed for 402 form 9 with entities 108/1 and 108/-1. The geometrical support for the TopoDS_Face is a Geom_Plane with boundaries: the parent plane defines the outer boundary; the child planes define the inner boundaries.

### Subfigures

<table>
<thead>
<tr>
<th>IGES entity type</th>
<th>CASCADE shape</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>308: Subfigure Definition</td>
<td>TopoDS_Compound</td>
<td>This IGES entity is only translated when there are no Singular Subfigure Instance entities.</td>
</tr>
<tr>
<td>408: Singular Subfigure Instance</td>
<td>TopoDS_Compound</td>
<td>This shape has the Subfigure Definition Compound as its origin and is positioned in space by its translation vector and its scale factor.</td>
</tr>
</tbody>
</table>

### Transformation Matrix
<table>
<thead>
<tr>
<th>type</th>
<th>Geom_Transformation</th>
<th>This entity is never translated alone. It must be included in the definition of another entity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>124: Transformation Matrix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Messages

Messages are displayed concerning the normal functioning of the processor (transfer, loading, etc.). You must declare an include file:

```cpp
#include <Interface_DT.hxx>
```

You have the choice of the following options for messages:

```cpp
IDT_SetLevel (level);
```

level modifies the level of messages:

- 0: no messages
- 1: raise and fail messages are displayed, as are messages concerning file access,
- 2: warnings are also displayed.

```cpp
IDT_SetFile ("tracefile.log");
```

prints the messages in a file,

```cpp
IDT_SetStandard();
```

restores screen output.
Tolerance management

Values used for tolerances during reading IGES

During the transfer of IGES to Open CASCADE Technology several parameters are used as tolerances and precisions for different algorithms. Some of them are computed from other using specific functions.

3D (spatial) tolerances

- Package method `Precision::Confusion` equal to $10^{-7}$ is used as a minimal distance between points, which are considered distinct.
- Resolution in the IGES file is defined in the Global section of an IGES file. It is used as a fundamental value of precision during the transfer.
- User-defined variable `read.precision.val` can be used instead of resolution from the file when parameter `read.precision.mode` is set to 1 (“User”).
- Field `EpsGeom` of the class `IGESToBRep_CurveAndSurface` is a basic precision for translating an IGES object. It is set for each object of class `IGESToBRep_CurveAndSurface` and its derived classes. It is initialized for the root of transfer either by value of resolution from the file or by value of `read.precision.val`, depending on the value of `read.precision.mode` parameter. It is returned by call to method `IGESToBRep_CurvAndSurface::GetEpsGeom`. As this value belongs to measurement units of the IGES file, it is usually multiplied by the coefficient `UnitFactor` (returned by method `IGESToBRep_CurvAndSurface::GetUnitFactor`) to convert it to Open CASCADE Technology units.
- Field `MaxTol` of the class `IGESToBRep_CurveAndSurface` is used as the maximum tolerance for some algorithms. Currently, it is computed as the maximum between 1 and `GetEpsGeom * GetUnitFactor`. This field is returned by method `IGESToBRep_CurvAndSurface::GetMaxTol`.

2D (parametric) tolerances
- Package method `Precision::PConfusion` equal to $0.01 \times \text{Precision::Confusion}$, i.e. $10^{-9}$. It is used to compare parametric bounds of curves.

- Field `EpsCoeff` of the class `IGESToBRep_CurveAndSurface` is a parametric precision for translating an IGES object. It is set for each object of class `IGESToBRep_CurveAndSurface` and its derived classes. Currently, it always has its default value $10^{-6}$. It is returned by call to method `IGESToBRep_CurvAndSurface::GetEpsCoeff`. This value is used for translating 2d objects (for instance, parametric curves).

- Methods `UResolution(tolerance3d)` and `VResolution(tolerance3d)` of the class `GeomAdaptor_Surface` or `BRepAdaptor_Surface` return tolerance in parametric space of a surface computed from 3D tolerance. When one tolerance value is to be used for both U and V parametric directions, the maximum or the minimum value of `UResolution` and `VResolution` is used.

- Methods `Resolution(tolerance3d)` of the class `GeomAdaptor_Curve` or `BRepAdaptor_Curve` return tolerance in the parametric space of a curve computed from 3D tolerance.

**Zero-dimensional tolerances**

- Field `Epsilon` of the class `IGESToBRep_CurveAndSurface` is set for each object of class `IGESToBRep_CurveAndSurface` and returned by call to method `GetEpsilon`. It is used in comparing angles and converting transformation matrices. In most cases, it is reset to a fixed value ($10^{-5} - 10^{-3}$) right before use. The default value is $10^{-4}$.

**Initial setting of tolerances in translating objects**

Transfer starts from one entity treated as a root (either the actual root in the IGES file or an entity selected by the user). The function which performs the transfer (that is `IGESToBRep_Actor::Transfer` or `IGESToBRep_Reader::Transfer`) creates an object of the type `IGESToBRep_CurveAndSurface`, which is intended for translating geometry.

This object contains three tolerances: `Epsilon`, `EpsGeom` and `EpsCoeff`. 
Parameter \textit{Epsilon} is set by default to value $10^{-4}$. In most cases when it is used in the package \textit{IGESToBRep}, it is reset to a fixed value, either $10^{-5}$ or $10^{-4}$ or $10^{-3}$. It is used as precision when comparing angles and transformation matrices and does not have influence on the tolerance of the resulting shape.

Parameter \textit{EpsGeom} is set right after creating a \textit{IGESToBRep\_CurveAndSurface} object to the value of resolution, taken either from the Global section of an IGES file, or from the \texttt{XSTEP.readprecision.val} parameter, depending on the value of \texttt{XSTEP.readprecision.mode}.

Parameter \textit{EpsCoeff} is set by default to $10^{-6}$ and is not changed.

During the transfer of a shape, new objects of type \textit{IGESToBRep\_CurveAndSurface} are created for translating subshapes. All of them have the same tolerances as the root object.

\section*{Transfer process}

\subsection*{Translating into Geometry}

Geometrical entities are translated by classes \textit{IGESToBRep\_BasicCurve} and \textit{IGESToBRep\_BasicSurface}. Methods of these classes convert curves and surfaces of an IGES file to Open CASCADE Technology geometry objects: \texttt{Geom\_Curve}, \texttt{Geom\_Surface}, and \texttt{Geom\_Transformation}.

Since these objects are not BRep objects, they do not have tolerances. Hence, tolerance parameters are used in these classes only as precisions: to detect specific cases (e.g., to distinguish a circle, an ellipse, a parabola and a hyperbola) and to detect bad cases (such as coincident points).

Use of precision parameters is reflected in the following classes:

- \textit{IGESToBRep\_BasicCurve} – all parameters and points are compared with precision \textit{EpsGeom}. All transformations (except \texttt{IGESToBRep\_BasicCurve::TransferTransformation}) are fulfilled with precision \textit{Epsilon} which is set to $10^{-3}$ (in the
**IGESToBRep_BasicCurve::TransferTransformation** the value $10^{-5}$ is used.

- **IGESToBRep_BasicCurve::TransferBSplineCurve** – all weights of BSplineCurve are assumed to be more than `Precision::PConfusion` (else the curve is not translated).

- **IGESToBRep_BasicSurface** – all parameters and points are compared with precision `EpsGeom`. All transformations are fulfilled with precision `Epsilon`, which is set to $10^{-3}$.

- **IGESToBRep_BasicSurface::TransferBSplineSurface** – all weights of BSplineSurface are assumed to be more than `Precision::PConfusion` (else the surface is not translated).

**Translating into Topology**

IGES entities represented as topological shapes and geometrical objects are translated into OCCT shapes by use of the classes `IGESToBRep_TopoCurve`, `IGESToBRep_TopoSurface`, `IGESToBRep_BRepEntity` and `ShapeFix_Wire`.

Class **IGESToBRep_BRepEntity** is intended for transferring BRep entities (IGES version is 5.1 or greater) while the two former are used for translating geometry and topology defined in IGES versions prior to 5.1. Methods from **IGESToBRep_BRepEntity** call methods from **IGESToBRep_TopoCurve** and **IGESToBRep_TopoSurface**, while those call methods from **IGESToBRep_BasicCurve** and **IGESToBRep_BasicSurface** to translate IGES geometry into OCCT geometry.

Although the IGES file contains only one parameter for tolerance in the Global Section, OCCT shapes are produced with different tolerances. As a rule, updating the tolerance is fulfilled according to local distances between shapes (distance between vertices of adjacent edges, deviation of edge’s 3D curve and its parametric curve and so on) and may be less or greater than precision in the file.

The following classes show what default tolerances are used when creating shapes and how they are updated during transfer.

**Class IGESToBRep_TopoCurve**
All methods are in charge of transferring curves from IGES curve entities (TransferCompositeCurve, Transfer2dCompositeCurve, TransferCurveOnFace, TransferBoundaryOnFace, TransferOffsetCurve, TransferTopoBasicCurve) if an entity has transformation call to IGESData_ToolLocation::ConvertLocation with Epsilon value set to $10^{-4}$.

- **IGESToBRep_TopoCurve::TransferPoint** – vertex is constructed from a Point entity with tolerance $EpsGeom*UnitFactor$.
- **IGESToBRep_TopoCurve::Transfer2dPoint** – vertex is constructed from a Point entity with tolerance $EpsCoeff$.
- **IGESToBRep_TopoCurve::TransferCompositeCurveGeneral** – obtains shapes (edges or wires) from other methods and adds them into the resulting wire. Two adjacent edges of the wire can be connected with tolerance up to $MaxTol$.
- **IGESToBRep_TopoCurve::TransferCurveOnFace** and **IGESToBRep_TopoCurve::TransferBoundaryOnFace** build a wire from 3D and 2D representations of a curve on surface. Edges and vertices of the wire cannot have tolerance larger than $MaxTol$. The value $EpsGeom*UnitFactor$ is passed into ShapeFix_Wire::SetPrecision and $MaxTol$ is passed into ShapeFix_Wire::MaxTolerance. To find out how these parameters affect the resulting tolerance changes, please, refer to class ShapeFix_Wire.
- **IGESToBRep_TopoCurve::TransferTopoBasicCurve** and **IGESToBRep_TopoCurve::Transfer2dTopoBasicCurve** – the boundary vertices of an edge (or a wire if a curve was of C0 continuity) translated from a basis IGES curve ($BSplineCurve$, CopiousData, Line, etc.) are built with tolerance $EpsGeom*UnitFactor$, the edge tolerance is $Precision::Confusion$. If a curve was divided into several edges, the common vertices of such adjacent edges have tolerance $Precision::Confusion$.

**Class IGESToBRep_TopoSurface**

All faces created by this class have tolerance $Precision::Confusion$.

**Class IGESToBRep_BRepEntity**

- **IGESToBRep_BRepEntity::TransferVertex** – the vertices from the VertexList entity are constructed with tolerance
\textit{EpsGeom*UnitFactor}.

- \textit{IGESToBRep_BRepEntity::TransferEdge} – the edges from the \textit{EdgeList} entity are constructed with tolerance \textit{Precision::Confusion}.
- \textit{IGESToBRep_BRepEntity::TransferLoop} – this function works like \textit{IGESToBRep_TopoCurve::TransferCurveOnFace} and \textit{IGESToBRep_TopoCurve::TransferBoundaryOnFace}.
- \textit{IGESToBRep_BRepEntity::TransferFace} – the face from the \textit{Face} IGES entity is constructed with tolerance \textit{Precision::Confusion}.

\textbf{Shape Healing classes}

After performing a simple mapping, shape-healing algorithms are called (class \textit{ShapeFix\_Shape}) by \textit{IGESToBRep\_Actor::Transfer()}\textit{)}. Shape-healing algorithm performs the correction of the resulting OCCT shape. Class \textit{ShapeFix\_Wire} can increase the tolerance of a shape. This class is used in \textit{IGESToBRep\_BRepEntity::TransferLoop}, \textit{IGESToBRep\_TopoCurve::TransferBoundaryOnFace} and \textit{IGESToBRep\_TopoCurve::TransferCurveOnFace} for correcting a wire. The maximum possible tolerance applied to the edges or vertices after invoking the methods of this class is \textit{MaxTolerance} (set by method \textit{ShapeFix\_Wire::MaxTolerance()} ).
Code architecture

The following diagram illustrates the structure of calls in reading IGES. The highlighted classes produce OCCT geometry.

The structure of calls in reading IGES
Example

```cpp
#include "IGESControl_Reader.hxx"
#include "TColStd_HSequenceOfTransient.hxx"
#include "TopoDS_Shape.hxx"
{
    IGESControl_Reader myIgesReader;
    Standard_Integer nIgesFaces, nTransFaces;

    myIgesReader.ReadFile ("MyFile.igs");
    //loads file MyFile.igs

    Handle(TColStd_HSequenceOfTransient) myList =
        myIgesReader.GiveList("iges-faces");
    //selects all IGES faces in the file and puts them
    //into a list called //MyList,

    nIgesFaces = myList-Length();
    nTransFaces = myIgesReader.TransferList(myList);
    //translates MyList,

    cout<<"IGES Faces: "<<nIgesFaces<<"  Transferred:"
        <<nTransFaces<<endl;
    TopoDS_Shape sh = myIgesReader.OneShape();
    //and obtains the results in an OCCT shape.
}
```
Writing IGES

Procedure

You can translate OCCT shapes to IGES entities in the following steps:

1. Initialize the process.
2. Set the translation parameters,
3. Perform the model translation,
4. Write the output IGES file.

You can translate several shapes before writing a file. Each shape will be a root entity in the IGES model.
Domain covered

There are two families of OCCT objects that can be translated:

- geometrical,
- topological.
Description of the process

Initializing the process

Choose the unit and the mode you want to use to write the output file as follows:

- **IGESControl_Controller::Init** performs standard initialization. Returns False if an error occurred.
- **IGESControl_Writer writer** uses the default unit (millimeters) and the default write mode (Face).
- **IGESControl_Writer writer (UNIT)** uses the Face write mode and any of the units that are accepted by IGES.
- **IGESControl_Writer writer (UNIT,modecr)** uses the unit (accepted by IGES) and the write mode of your choice.
  - 0: Faces,
  - 1: BRep The result is an **IGESControl_Writer** object.

Setting the translation parameters

The following parameters are used for the OCCT-to-IGES translation.

- **write.iges.brep.mode**: allows choosing the write mode:
  - "Faces" (0): OCCT *TopoDS_Faces* will be translated into IGES 144 (Trimmed Surface) entities, no BRep entities will be written to the IGES file,
  - "BRep" (1): OCCT *TopoDS_Faces* will be translated into IGES 510 (Face) entities, the IGES file will contain BRep entities.

Read this parameter with:

```cpp
Standard_Integer byvalue = Interface_Static::IVal("write.iges.brep.mode");
```

Modify this parameter with:

```cpp
Interface_Static::SetIVal("write.iges.brep.mode", 1);
```

Default value is "Faces" (0).

- **write.convertsurface.mode** when writing to IGES in the BRep mode,
this parameter indicates whether elementary surfaces (cylindrical, conical, spherical, and toroidal) are converted into corresponding IGES 5.3 entities (if the value of a parameter value is On), or written as surfaces of revolution (by default).

- **write.iges.unit:** allows choosing the unit. The default unit for Open CASCADE Technology is "MM" (millimeter). You can choose to write a file into any unit accepted by IGES.
  - Read this parameter with `Standard_String byvalue = Interface_Static::CVal("write.iges.unit");`
  - Modify this parameter with `Interface_Static::SetCVal("write.iges.unit", "INCH");`

- **write.iges.header.autor:** gives the name of the author of the file. The default value is the system name of the user.
  - Read this parameter with `Standard_String byvalue = Interface_Static::CVal("write.iges.header.author");`
  - Modify this parameter with `Interface_Static::SetCVal("write.iges.header.author", "name");`

- **write.iges.header.company:** gives the name of the sending company. The default value is "" (empty).
  - Read this parameter with `Standard_String byvalue = Interface_Static::CVal("write.iges.header.company");`
  - Modify this parameter with `Interface_Static::SetCVal("write.iges.header.company", "Open CASCADE");`

- **write.iges.header.product:** gives the name of the sending product. The default value is "CAS.CADE IGES processor Vx.x", where x.x means the current version of Open CASCADE Technology.
  - Read this parameter with `Standard_String byvalue = Interface_Static::CVal("write.iges.header.product");`
  - Modify this parameter with `Interface_Static::SetCVal("write.iges.header.product", "product name");`

- **write.iges.header.receiver:** – gives the name of the receiving company. The default value is "" (empty).
  - Read this parameter with `Standard_String byvalue = Interface_Static::CVal("write.iges.header.receiver");`
  - Modify this parameter with `Interface_Static::SetCVal("write.iges.header.receiver", "receiver name");`

- **write.precision.mode:** specifies the mode of writing the resolution value into the IGES file.
  - "Least" (-1): resolution value is set to the minimum tolerance of all edges and all vertices in an OCCT shape.
- "Average" (0): resolution value is set to average between the average tolerance of all edges and the average tolerance of all vertices in an OCCT shape. This is the default value.
- "Greatest" (1): resolution value is set to the maximum tolerance of all edges and all vertices in an OCCT shape.
- "Session" (2): resolution value is that of the write.precision.val parameter.
- Read this parameter with `Standard_Integer ic = Interface_Static::IVal("write.precision.mode");`
- Modify this parameter with `if (!Interface_Static::SetIVal("write.precision.mode",1)) .. error ..`

- `write.precision.val`: is the user precision value. This parameter gives the resolution value for an IGES file when the `write.precision.mode` parameter value is 1. It is equal to 0.0001 by default, but can take any real positive (non null) value.

Read this parameter with:

```plaintext
Standard_Real rp = Interface_Static::RVal("write.precision.val");
```

Modify this parameter with:

```plaintext
if (!Interface_Static::SetRVal("write.precision.val",0.01)) .. error ..
```

Default value is 0.0001.

**write.iges.resource.name**

and

**write.iges.sequence**

are the same as the corresponding read.iges.* parameters, please, see above. Note that the default sequence for writing contains *DirectFaces* operator, which converts elementary surfaces based on left-hand axes.
(valid in CASCADE) to right-hand axes (which are valid only in IGES).

Default values:

```
write.iges.resource.name - IGES,
write.iges.sequence - ToIGES.
```

**Performing the Open CASCADE Technology shape translation**

You can perform the translation in one or several operations. Here is how you translate topological and geometrical objects:

```
Standard_Boolean ok = writer.AddShape (TopoDS_Shape);
```

ok is True if translation was correctly performed and False if there was at least one entity that was not translated.

```
Standard_Boolean ok = writer.AddGeom (geom);
```

where `geom` is `Handle(Geom_Curve)` or `Handle(Geom_Surface)`. `ok` is True if the translation was correctly performed and False if there was at least one entity whose geometry was not among the allowed types.

**Writing the IGES file**

Write the IGES file with:

```
Standard_Boolean ok = writer.Write ("filename.igs");
```

to give the file name.

```
Standard_Boolean ok = writer.Write (S);
```

where `S` is `Standard_OStream` `ok` is True if the operation was correctly performed and False if an error occurred (for instance, if the processor could not create the file).
Mapping Open CASCADE Technology shapes to IGES entities

Translated objects depend on the write mode that you chose. If you chose the Face mode, all of the shapes are translated, but the level of topological entities becomes lower (geometrical one). If you chose the BRep mode, topological OCCT shapes become topological IGES entities.

**Curves**

<table>
<thead>
<tr>
<th>CASCADE shape</th>
<th>IGES entity type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geom_BsplineCurve</td>
<td>126: BSpline Curve</td>
<td></td>
</tr>
<tr>
<td>Geom_BezierCurve</td>
<td>126: BSpline Curve</td>
<td></td>
</tr>
<tr>
<td>Geom_TrimmedCurve</td>
<td>All types of translatable IGES curves</td>
<td>The type of entity output depends on the type of the basis curve. If the curve is not trimmed, limiting points will be defined by the CASCADE RealLast value.</td>
</tr>
<tr>
<td>Geom_Circle</td>
<td>100: Circular Arc or 126: BSpline Curve</td>
<td>A BSpline Curve is output if the Geom_Circle is closed</td>
</tr>
<tr>
<td>Geom_Ellipse</td>
<td>104: Conic Arc or 126: BSpline Curve</td>
<td>A Conic Arc has Form 1. A BSpline Curve is output if the Geom_Ellipse is closed.</td>
</tr>
<tr>
<td>Geom_Hyperbola</td>
<td>104: Conic Arc</td>
<td>Form 2</td>
</tr>
</tbody>
</table>
### Surfaces

<table>
<thead>
<tr>
<th>CASCADE shapes</th>
<th>IGES entity type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geom_BSplineSurface</td>
<td>128: BSpline Surface</td>
<td>The type of entity output depends on the type of the basis surface. If the surface is not trimmed and has infinite edges/sides, the coordinates of the sides in IGES will be limited to the CASCADE RealLast value.</td>
</tr>
<tr>
<td>Geom_BezierSurface</td>
<td>128: BSpline Surface</td>
<td>A BSpline Surface (of degree 1 in U and V) is output if you are working in the face mode. A Plane Surface is output if you are working in the BRep</td>
</tr>
<tr>
<td>Geom_RectangularTrimmedSurface</td>
<td>All types of translatable IGES surfaces.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1:**

<table>
<thead>
<tr>
<th>Geom_Parabola</th>
<th>104: Conic Arc</th>
<th>Form 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geom_Line</td>
<td>110: Line</td>
<td></td>
</tr>
<tr>
<td>Geom_OffsetCurve</td>
<td>130: Offset Curve</td>
<td></td>
</tr>
<tr>
<td>CASCADE shapes</td>
<td>IGES entity type</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Single TopoDS_Vertex</td>
<td>116: 3D Point</td>
<td></td>
</tr>
<tr>
<td>TopoDS_Vertex in a TopoDS_Edge</td>
<td>No equivalent</td>
<td>Not transferred.</td>
</tr>
<tr>
<td>TopoDS_Edge</td>
<td>All types of translatable IGES curves</td>
<td>The output IGES curve will be the one that corresponds to the Open CASCADE Technology definition.</td>
</tr>
</tbody>
</table>
| Single                                  | 102:             | Each TopoDS_Edge in the
<table>
<thead>
<tr>
<th>TopoDS_Wire</th>
<th>Composite Curve</th>
<th>TopoDS_Wire results in a curve.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopoDS_Wire in a TopoDS_Face</td>
<td>142: Curve On Surface</td>
<td>Both curves (3D and pcurve) are transferred if they are defined and result in a simple curve or a composite curve depending on whether there is one or more edges in the wire. Note: if the basis surface is a plane (108), only the 3D curve is used.</td>
</tr>
<tr>
<td>TopoDS_Face</td>
<td>144: Trimmed Surface</td>
<td></td>
</tr>
<tr>
<td>TopoDS_Shell</td>
<td>402: Form 1 Group or no equivalent</td>
<td>Group is created only if TopoDS_Shell contains more than one TopoDS_Face. The IGES group contains Trimmed Surfaces.</td>
</tr>
<tr>
<td>TopoDS_Solid</td>
<td>402: Form 1 Group or no equivalent</td>
<td>Group is created only if TopoDS_Solid contains more than one TopoDS_Shell. One IGES entity is created per TopoDS_Shell.</td>
</tr>
<tr>
<td>TopoDS_CompSolid</td>
<td>402: Form 1 Group or no equivalent</td>
<td>Group is created only if TopoDS_CompSolid contains more than one TopoDS_Solid. One IGES entity is created per TopoDS_Solid.</td>
</tr>
<tr>
<td>TopoDS_Compound</td>
<td>402: Form 1 Group or no equivalent</td>
<td>Group is created only if TopoDS_Compound contains more than one item. One IGES entity is created per TopoDS_Shape in the TopoDS_Compound. If TopoDS_Compound is nested into another TopoDS_Compound, it is not mapped.</td>
</tr>
</tbody>
</table>

**Topological entities -- Translation in BRep mode**

<table>
<thead>
<tr>
<th>CASCADE</th>
<th>IGES entity</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shapes</td>
<td>type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Single TopoDS_Vertex</td>
<td>No equivalent</td>
<td>Not transferred.</td>
</tr>
<tr>
<td>TopoDS_Vertex in a TopoDS_Edge</td>
<td>One item in a 502: VertexList</td>
<td></td>
</tr>
<tr>
<td>TopoDS_Edge</td>
<td>No equivalent</td>
<td>Not transferred as such. This entity serves as a part of a Loop entity.</td>
</tr>
<tr>
<td>TopoDS_Edge in a TopoDS_Wire</td>
<td>One item in a 504: EdgeList</td>
<td></td>
</tr>
<tr>
<td>TopoDS_Wire</td>
<td>508: Loop</td>
<td></td>
</tr>
<tr>
<td>TopoDS_Face</td>
<td>510: Face</td>
<td>If the geometrical support of the face is a plane, it will be translated as a 190 entity PlaneSurface.</td>
</tr>
<tr>
<td>TopoDS_Shell</td>
<td>514: Shell</td>
<td></td>
</tr>
<tr>
<td>TopoDS_Solid</td>
<td>186: Manifold Solid</td>
<td></td>
</tr>
<tr>
<td>TopoDS_CompSolid</td>
<td>402 Form1 Group or no equivalent</td>
<td>Group is created only if TopoDS_Compound contains more than one item. One IGES Manifold Solid is created for each TopoDS_Solid in the TopoDS_CompSolid.</td>
</tr>
<tr>
<td>TopoDS_Compound</td>
<td>402 Form1 Group or no equivalent</td>
<td>Group is created only if TopoDS_Compound contains more than one item. One IGES entity is created per TopoDS_Shape in the TopoDS_Compound. If TopoDS_Compound is nested into another TopoDS_Compound it is not mapped.</td>
</tr>
</tbody>
</table>
Tolerance management

Setting resolution in an IGES file

There are several possibilities to set resolution in an IGES file. They are controlled by write.precision.mode parameter; the dependence between the value of this parameter and the set resolution is described in paragraph Setting the translation parameters.

If the value of parameter write.precision.mode is -1, 0 or 1, resolution is computed from tolerances of sub-shapes inside the shape to be translated. In this computation, only tolerances of TopoDS_Edges and TopoDS_Vertices participate since they reflect the accuracy of the shape. TopoDS_Faces are ignored in computations since their tolerances may have influence on resulting computed resolution while IGES resolution mainly concerns points and curves but not surfaces.
Code architecture

Graph of calls

The following diagram illustrates the class structure in writing IGES. The highlighted classes are intended to translate geometry.

The class structure in writing IGES
Example

```cpp
#include <IGESControl_Controller.hxx>
#include <IGESControl_Writer.hxx>
#include <TopoDS_Shape.hxx>
Standard_Integer main()
{
    IGESControl_Controller::Init();
    IGESControl_Writer ICW (;MM;, 0);
    // creates a writer object for writing in Face mode
    // with millimeters
    TopoDS_Shape sh;
    ICW.AddShape (sh);
    // adds shape sh to IGES model
    ICW.ComputeModel();
    Standard_Boolean OK = ICW.Write (;MyFile.igs;);
    // writes a model to the file MyFile.igs
}
```
Using XSTEPDRAW

XSTEPDRAW UL is intended for creating executables for testing XSTEP interfaces interactively in the DRAW environment. It provides an additional set of DRAW commands specific for the data exchange tasks, which allow loading and writing data files and analysis of resulting data structures and shapes.

In the description of commands, square brackets ([]) are used to indicate optional parameters. Parameters given in the angle brackets (<>) and sharps (#) are to be substituted by an appropriate value. When several exclusive variants are possible, vertical dash (|) is used.
Setting interface parameters

A set of parameters for importing and exporting IGES files is defined in the XSTEP resource file. In XSTEPDRAW, these parameters can be viewed or changed using command

```
Draw> param [<parameter_name> [<value>]]
```

Command `param` with no arguments gives a list of all parameters with their values. When argument `parameter_name` is specified, information about this parameter is printed (current value and short description).

The third argument is used to set a new value of the given parameter. The result of the setting is printed immediately.

During all interface operations, the protocol of the process (fail and warning messages, mapping of the loaded entities into OCCT shapes etc.) can be output to the trace file. Two parameters are defined in the DRAW session: trace level (integer value from 0 to 9, default is 0), and trace file (default is a standard output).

Command `xtrace` is intended to view and change these parameters:

- `Draw> xtrace` – prints current settings (e.g.: "Level=0 - Standard Output");
- `Draw> xtrace #` – sets the trace level to the value #;
- `Draw> xtrace tracefile.log` – sets the trace file as `tracefile.log`;
- `Draw xtrace` – directs all messages to the standard output.
Reading IGES files

For a description of parameters used in reading an IGES file refer to Setting the translation parameters.

These parameters are set by command `param`:

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision for input entities</td>
<td><code>read.precision.mode</code></td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td><code>read.precision.val</code></td>
<td><code>real</code></td>
</tr>
<tr>
<td>Continuity of B splines</td>
<td><code>read.iges.bspline.continuity</code></td>
<td>0-2</td>
</tr>
<tr>
<td>Surface curves</td>
<td><code>read.surfacecurve.mode</code></td>
<td>2, 3 or 0</td>
</tr>
</tbody>
</table>

It is possible either only to load an IGES file into memory (i.e. to fill the model with data from the file), or to read it (i.e. to load and convert all entities to OCCT shapes).

Loading is done by the command

```
Draw> xload <file_name>
```

Once the file is loaded, it is possible to investigate the structure of the loaded data. To learn how to do it see Analyzing the transferred.

Reading of an IGES file is done by the command

```
Draw> igesbrep <file_name> <result_shape_name> [<selection>]
```

Here a dot can be used instead of a filename if the file is already loaded by `xload` or `igesbrep` command. In that case, only conversion of IGES entities to OCCT shapes will be done.

Command `igesbrep` will interactively ask the user to select a set of entities to be converted:
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>End</td>
<td>finish conversion and exit igesbrep</td>
</tr>
<tr>
<td>1</td>
<td>Visible roots</td>
<td>convert only visible roots</td>
</tr>
<tr>
<td>2</td>
<td>All roots</td>
<td>convert all roots</td>
</tr>
<tr>
<td>3</td>
<td>One entity</td>
<td>convert entity with number provided by the user</td>
</tr>
<tr>
<td>4</td>
<td>Selection</td>
<td>convert only entities contained in selection</td>
</tr>
</tbody>
</table>

After the selected set of entities is loaded the user will be asked how loaded entities should be converted into OCCT shapes (e.g., one shape per root or one shape for all the entities). It is also possible to save loaded shapes in files, and to cancel loading.

The second parameter of the \textit{igesbrep} command defines the name of the loaded shape. If several shapes are created, they will get indexed names. For instance, if the last parameter is ‘s’, they will be \texttt{s\_1, \ldots s\_N}.

\texttt{xst-transferrable-roots} by default. An asterisk ‘*’ can be specified instead of \textit{iges-visible-transf-roots}. For possible values of \texttt{selection} refer to Selecting entities section.

Instead of \textit{igesbrep} it is possible to use commands:

\begin{verbatim}
Draw> trimport <file_name> <result_shape_name> <selection>
\end{verbatim}

which outputs the result of translation of each selected entity into one shape, or

\begin{verbatim}
Draw> trimpcomp <file_name> <result_shape_name> <selection>
\end{verbatim}

which outputs the result of translation of all selected entities into one shape (\texttt{TopoDS_Compound} for several entities).

An asterisk ‘*’ can be specified instead of \texttt{selection}, it means \texttt{xst-transferrable-roots}.

During the IGES translation, a map of correspondence between IGES entities and OCCT shapes is created. The following commands are available:
• *Draw> tpent #* – provides information on the result of translation of the given IGES entity;
• *Draw> tpdraw #* – creates an OCCT shape corresponding to an IGES entity;
• *Draw> fromshape <shape_name>* – provides the number of an IGES entity corresponding to an OCCT shape;
• *Draw> tpclear* – clears the map of correspondences between IGES entities and OCCT shapes.
Analyzing the transferred data

The procedure of analysis of the data import can be divided into two stages:

1. Checking the file contents;
2. Estimation of translation results (conversion and validated ratios).

Checking file contents

General statistics on the loaded data can be obtained by using command

```
Draw> data <symbol>
```

The information printed by this command depends on the symbol specified:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>Prints information contained in the header of the file (Start and Global sections)</td>
</tr>
<tr>
<td>c or f</td>
<td>Runs check procedure of the integrity of the loaded data and prints the resulting statistics (f works only with fails while c with both fail and warning messages)</td>
</tr>
<tr>
<td>t</td>
<td>The same as c or f, with a list of failed or warned entities</td>
</tr>
<tr>
<td>m or l</td>
<td>The same as t but also prints a status for each entity</td>
</tr>
<tr>
<td>e</td>
<td>Lists all entities of the model with their numbers, types, status of validity etc.</td>
</tr>
<tr>
<td>r</td>
<td>The same as e but lists only root entities</td>
</tr>
</tbody>
</table>

There is a set of special objects, which can be used to operate with the loaded model. They can be of the following types:

<table>
<thead>
<tr>
<th>Special object type</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>allow selecting subsets of entities of the loaded</td>
</tr>
</tbody>
</table>
A list of these objects defined in the current session can be printed in DRAW by command

\texttt{Draw> listitems}

In the following commands if several \texttt{<selection>} arguments are specified the results of each following selection are applied to the results of the previous one.

\texttt{Draw> givelist <selection}\_name\_1 [\texttt{<selection}\_name\_2]}

prints a list of loaded entities defined by selection argument.

\texttt{Draw> givecount <selection}\_name\_1 [\texttt{<selection}\_name\_2]}

prints a number of loaded entities defined by \texttt{selection} argument.

Three commands are used to calculate statistics on the entities in the model:

- \texttt{Draw> count} \texttt{<counter>} [\texttt{<selection>} ...] – prints only a number of entities per each type matching the criteria defined by arguments.
- \texttt{Draw> sumcount} \texttt{<counter>} [\texttt{<selection>} ...] – prints the total number of entities of all types matching the criteria defined by arguments and the largest number corresponding to one type.
- \texttt{Draw> listcount} \texttt{<counter>} [\texttt{<selection>} ...] – prints a list of entities per each type matching the criteria defined by arguments.

Optional \texttt{<selection>} argument, if specified, defines a subset of entities, which are to be taken into account. Argument \texttt{<counter>} should be one of the currently defined counters:

<table>
<thead>
<tr>
<th>Counter</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xst-types</td>
<td>Calculates how much entities of each OCCT type exist</td>
</tr>
<tr>
<td>iges-</td>
<td>Calculates how much entities of each IGES type and form</td>
</tr>
</tbody>
</table>
Types exist

| iges-levels | Calculates how much entities lie in different IGES levels |

The command:

```
Draw> listtypes <selection_name> ...
```

gives a list of entity types which were encountered in the last loaded file (with a number of IGES entities of each type). The list can be shown not for all entities but for a subset of them. This subset is defined by an optional selection argument.

Entities in the IGES file are numbered in the succeeding order. An entity can be identified either by its number (#) or by its label. Label is the letter ‘D’ followed by the index of the first line with the data for this entity in the Directory Entry section of the IGES file. The label can be calculated on the basis of the number as ‘D(2*# -1)’. For example, entity # 6 has label D11.

- `Draw> elab #` – provides a label for an entity with a known number;
- `Draw> enum #` – prints a number for an entity with the given label;
- `Draw> entity # <level_of_information>` – gives the content of an IGES entity;
- `Draw> estat #` – provides the list of entities referenced by a given entity and the list of entities referencing to it.

**Estimating the results of reading IGES**

All of the following commands are available only after the data are converted into OCCT shapes (i.e. after command `igesbrep`).

```
Draw> tpstat [*|?]<symbol> [<selection>]
```

provides all statistics on the last transfer, including the list of transferred entities with mapping from IGES to OCCT types, as well as fail and warning messages. The parameter `<symbol>` defines what information will be printed:

- `G` – General statistics (list of results and messages)
- C – Count of all warning and fail messages
- C – List of all warning and fail messages
- F – Count of all fail messages
- F – List of all fail messages
- N – List of all transferred roots
- S – The same, with types of source entity and result type
- B – The same, with messages
- T – Count of roots for geometrical types
- R – Count of roots for topological types
- I – The same, with a type of the source entity

The sign ‘*’ before the parameters n, s, b, t, r makes it work on all entities (not only on roots). The sign ‘?’ before n, s, b, t limits the scope of information to invalid entities.

Optional argument <selection> can limit the action of the command with a selected subset of entities. To get help, run this command without arguments.

For example, to get translation ratio on IGES faces, you can use.

```
Draw:> tpstat *1 iges-faces
```

The second version of the same command is TPSTAT (not capital spelling).

```
Draw:> TPSTAT <symbol>
```

Symbol can be of the following values:

- g – General statistics (list of results and messages)
- c – Count of all warning and fail messages
- C – List of all warning and fail messages
- r – Count of resulting OCCT shapes per each type
- s – Mapping of IGES roots and resulting OCCT shapes

Sometimes the trimming contours of IGES faces (i.e., entity 141 for 143, 142 for 144) can be lost during translation due to fails.

The number of lost trims and the corresponding IGES entities can be obtained by the command:
Draw> tplosttrim [<IGES_type>]

It outputs the rank and DE numbers of faces that lost their trims and their numbers for each type (143, 144, 510) and their total number. If a face lost several of its trims it is output only once.

Optional parameter <IGES_type> can be TrimmedSurface, BoundedSurface or Face to specify the only type of IGES faces.

For example, to get untrimmed 144 entities, use command

Draw> tplosttrim TrimmedSurface

To get the information on OCCT shape contents, use command

Draw> statshape <shape_name>

It outputs the number of each kind of shapes (vertex, edge, wire, etc.) in a shape and some geometrical data (number of C0 surfaces, curves, indirect surfaces, etc.).

Note. The number of faces is returned as a number of references. To obtain the number of single instances the standard command (from TTOPOLOGY executable) nbshapes can be used.

To analyze the internal validity of a shape, use command

Draw> checkbrep <shape_name> <expurged_shape_name>

It checks the geometry and topology of a shape for different cases of inconsistency, like self-intersecting wires or wrong orientation of trimming contours. If an error is found, it copies bad parts of the shape with the names "expurged_subshape_name _#" and generates an appropriate message. If possible, this command also tries to find IGES entities the OCCT shape was produced from.

<expurged_shape_name> will contain the original shape without invalid subshapes.

To get information on tolerances of subshapes, use command
Draw> tolerance <shape_name> [<min> [<max>]] [<symbol>]]

It outputs maximum, average and minimum values of tolerances for each kind of subshapes having tolerances or it can output tolerances of all subshapes of the whole shape.

When specifying min and max arguments this command outputs shapes with names <shape_name>... and their total number with tolerances in the range [min, max].

<Symbol> is used for specifying the kind of sub-shapes to analyze:

- v – for vertices,
- e – for edges,
- f – for faces,
- c – for shells and faces.
Writing an IGES file

Refer to **Setting the translation parameters** for a description of parameters used in reading an IGES file. The parameters are set by command `param`:

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>XSTEP.iges.header.author</td>
<td>String</td>
</tr>
<tr>
<td>Company</td>
<td>XSTEP.iges.header.company</td>
<td>String</td>
</tr>
<tr>
<td>Receiver</td>
<td>XSTEP.iges.header.receiver</td>
<td>String</td>
</tr>
<tr>
<td>Write mode for shapes</td>
<td>XSTEP.iges.writebrep.mode</td>
<td>0/Faces or 1/BRep</td>
</tr>
<tr>
<td>Measurement units</td>
<td>XSTEP.iges.unit</td>
<td>1-11 (or a string value)</td>
</tr>
</tbody>
</table>

Several shapes can be written in one file. To start writing a new file, enter command

```
Draw> newmodel
```

This command clears the `InterfaceModel` to make it empty.

```
Draw> brepiges <shape_name_1> [<filename.igs>]
```

Converts the specified shapes into IGES entities and puts them into the `InterfaceModel`.

```
Draw> writeall <filename.igs>
```

Allows writing the prepared model to a file with name `filename.igs`. 
Reading from and writing to IGES

Reading from IGES

Load an IGES file

Before performing any other operation, you must load an IGES file with:

```cpp
IGESCAFControl_Reader reader(XSDRAW::Session(), Standard_False);
IFSelect_ReturnStatus stat = reader.ReadFile("filename.igs");
```

Loading the file only memorizes, but does not translate the data.

Check the loaded IGES file

This step is not obligatory. See the description of Checking the IGES file above.

Set parameters for translation to XDE

See the description of Setting translation parameters above.

In addition, the following parameters can be set for XDE translation of attributes:

- For transferring colors:
  ```cpp
  reader.SetColorMode(mode);
  // mode can be Standard_True or Standard_False
  ```

- For transferring names:
  ```cpp
  reader.SetNameMode(mode);
  // mode can be Standard_True or Standard_False
  ```

Translate an IGES file to XDE
The following function performs a translation of the whole document:

```c++
Standard_Boolean ok = reader.Transfer(doc);
```

where `doc` is a variable which contains a handle to the output document and should have a type `Handle(TDocStd_Document)`.
Writing to IGES

The translation from XDE to IGES can be initialized as follows:

```plaintext
IGESCAFControl_Writer
  aWriter(XSDRAW::Session(),Standard_False);
```

Set parameters for translation from XDE to IGES

The following parameters can be set for translation of attributes to IGES:

- For transferring colors:
  ```plaintext
  aWriter.SetColorMode(mode);
  // mode can be Standard_True or Standard_False
  ```

- For transferring names:
  ```plaintext
  aWriter.SetNameMode(mode);
  // mode can be Standard_True or Standard_False
  ```

Translate an XDE document to IGES

You can perform the translation of a document by calling the function:

```plaintext
IFSelect_ReturnStatus aRetSt = aWriter.Transfer(doc);
```

where "doc" is a variable which contains a handle to the input document for transferring and should have a type `Handle(TDocStd_Document)`.

Write an IGES file

Write an IGES file with:

```plaintext
IFSelect_ReturnStatus statw =
  aWriter.WriteFile("filename.igs");
```

or

```plaintext
IFSelect_ReturnStatus statw = writer.WriteFile (S);
```
where $S$ is OStream.
STEP processor

Table of Contents

- Introduction
  - STEP Exchanges in Open Cascade technology
  - STEP Interface
- Reading STEP
  - Procedure
- Domain covered
  - Assemblies
  - Shape representations
  - Topological entities
  - Geometrical entities
- Description of the process
  - Loading the STEP file
  - Checking the STEP file
  - Setting the translation parameters
  - Performing the STEP file translation
  - Getting the translation results
Selecting STEP entities for translation

Mapping STEP entities to Open CASCADE Technology shapes

- Assembly structure representation entities
- Models
- Topological entities
- Geometrical entities

Tolerance management
- Values used for tolerances during reading STEP
- Initial setting of tolerances in translating objects
- Transfer process

Code architecture
Example

Writing STEP
- Procedure
- Domain covered
  - Writing geometry and topology
  - Writing assembly structures

Description of the process
  - Initializing the process
Setting the translation parameters
Performing the Open CASCADE Technology shape translation
Writing the STEP file
Mapping Open CASCADE Technology shapes to STEP entities
Assembly structures and product information
Topological shapes
Geometrical objects
Tolerance management
Code architecture
Graph of calls
Example
Physical STEP file reading and writing
Architecture of STEP Read and Write classes
General principles
Complex entities
Physical file reading
Loading a STEP file and syntactic analysis of its contents
Mapping STEP
entities to arrays of strings

- Creating empty Open CASCADE Technology objects that represent STEP entities
- Initializing Open CASCADE Technology objects
- Building a graph

- How to add a new entity in scope of the STEP processor
- Physical file writing
- How to add a new entity to write in the STEP file.

- Using DRAW

  - DRAW STEP Commands Overview
  - Setting the interface parameters
  - Reading a STEP file
  - Analyzing the transferred data
    - Checking file contents
    - Estimating the results of reading STEP
  - Writing a STEP file

- Reading from and writing to STEP

  - Reading from STEP
  - Attributes read from STEP
  - Writing to STEP
  - Attributes written to STEP
Introduction

STEP is more and more widely used to exchange data between various software, involved in CAD, PDM, Analysis, etc... STEP is far more than an "exchange standard" : it provides a technology and a set of methodologies to describe the data to exchange in a modular and upgradeable way. Regarding OCCT, this mostly applies to CAD data but it is not a limitation, other kinds of data for specific applications can be addressed too.

Open Cascade allows its users to employ STEP in the following domains:

- Exchange of data for technical applications, following the state-of-the-art definitions and rules;
- Extension of case coverage, according to specific needs or to the evolution of general business uses;
- Expertise in data architecture of an application, to get experience from STEP definitions and make easier the mapping to them, for a
better interoperability with outer world.

This manual is intended to provide technical documentation on the Open CASCADE Technology (OCCT) STEP processor and to help Open CASCADE Technology users with the use of the STEP processor (to read and write STEP files).

Only geometrical, topological STEP entities (shapes) and assembly structures are translated by the basic translator described in sections 2 to 6. Data that cannot be translated on this level are also loaded from a STEP file and can be translated later. XDE STEP translator (see section 7 Reading from and writing to XDE) translates names, colors, layers, validation properties and other data associated with shapes and assemblies into XDE document.

File translation is performed in the programming mode, via C++ calls.

Shape Healing toolkit provides tools to heal various problems, which may be encountered in translated shapes, and to make them valid in Open CASCADE. The Shape Healing is smoothly connected to STEP translator using the same API, only the names of API packages change.

For testing the STEP component in DRAW Test Harness, a set of commands for reading and writing STEP files and analysis of relevant data are provided by the TKXSDRAW plugin.

See also our E-learning & Training offerings.
STEP Exchanges in Open Cascade technology

Beyond the upper level API, which is fitted for an easy end-use, the STEP exchange functions enter in the general frame of Exchanges in Open Cascade, adapted for STEP:

- Specific packages for Data definition and checking;
- Physical Access supported by Drivers (Part 21 file access is embedded);
- Conversion to/from Open Cascade or applicative data supported by drivers (OCC-BREP and XDE ard basically provided);
- Tools for analysis, filtering, etc... including DRAW commands.

These modules share common architecture and capabilities with other exchange modules of Open Cascade, like Shape Healing. Also, built-in Viewer and Converter (as Plugin for Netscape, Internet Explorer ..), are based on the same technology.

In addition, Open Cascade provides tools to process models described using STEP: to reflect EXPRESS descriptions, to read, write and check data, to analyze the whole models ... Their key features are:

- Modularity by sets of data types, which can be hierarchized to reflect the original modularity describing the resources and application protocols;
- Implementation as C++ classes, providing comprehensive access to their members;
- Early binding is basically used, providing good performance, easy installation and use as well as the capability to support non-compiled descriptions.

This provides a natural way to deal with non-supported protocols when they share common definitions, as for geometry, which can then be exploited. The common frame, as the already supported data types, give a good foundation to go towards new uses of STEP, either on data definition (protocols from ISO or from industrial consortia) or on mapping with applicative data.
**STEP Interface**

The STEP interface reads STEP files produced in accordance with STEP Application Protocol 214 (Conformance Class 2 both CD and DIS versions of schema) and translates them to Open CASCADE Technology models. STEP Application Protocol 203 and some parts of AP242 are also supported.

The STEP interface also translates OCCT models to STEP files. STEP files that are produced by this interface conform to STEP AP 203 or AP 214 (Conformance Class 2, either CD or DIS version of the schema) depending on the user's option.

Basic interface reads and writes geometrical, topological STEP data and assembly structures.

The interface is able to translate one entity, a group of entities or a whole file.

Other kinds of data such as colors, validation properties, layers, GD&T, names and the structure of assemblies can be read or written with the help of XDE tools: `STEPCAFControl_Reader` and `STEPCAFControl_Writer`.

To choose a translation mode when exporting to a STEP format, use `STEPControl_STEPMModelType`.

There is a set of parameters that concern the translation and can be set before the beginning of the translation.

Please, note:

- a STEP model is a STEP file that has been loaded into memory;
- all references to shapes indicate OCCT shapes unless otherwise explicitly stated;
- a root entity is the highest level entity of any given type, i.e. an entity that is not referenced by any other one.
Reading STEP

Procedure

You can translate a STEP file into an OCCT shape in the following steps:

1. load the file,
2. check file consistency,
3. set the translation parameters,
4. perform the translation,
5. fetch the results.
Domain covered

Assemblies

The ProSTEP Round Table Agreement Log (version July 1998), item 21, defines two alternatives for the implementation of assembly structure representations: using mapped_item entities and using representation_relationship_with_transformation entities. Both these alternative representations are recognized and processed at reading. On writing, the second alternative is always employed.

Handling of assemblies is implemented in two separate levels: firstly STEP assembly structures are translated into OCCT shapes, and secondly the OCCT shape representing the assembly is converted into any data structure intended for representing assemblies (for example, OCAF).

The first part of this document describes the basic STEP translator implementing translation of the first level, i.e. translation to OCCT Shapes. On this level, the acyclic graph representing the assembly structure in a STEP file is mapped into the structure of nested TopoDS_Compounds in Open CASCADE Technology. The (sub)assemblies become (sub)compounds containing shapes which are the results of translating components of that (sub)assembly. The sharing of components of assemblies is preserved as Open CASCADE Technology sharing of subshapes in compounds.

The attributive information attached to assembly components in a STEP file (such as names and descriptions of products, colors, layers etc.) can be translated after the translation of the shape itself by parsing the STEP model (loaded in memory). Several tools from the package STEPConStruct provide functionalities to read styles (colors), validation properties, product information etc. Implementation of the second level of translation (conversion to XDE data structure) is provided by XDE STEP translator.

Shape representations

Length units, plane angle units and the uncertainty value are taken from
*shape_representation* entities. This data is used in the translation process.

The types of STEP representation entities that are recognized are:

- advanced_brep_shape_representation
- faceted_brep_shape_representation
- manifold_surface_shape_representation
- geometrically_bounded_wireframe_shape_representation
- geometrically_bounded_surface_shape_representation
- hybrid representations (shape_representation containing models of different type)

**Topological entities**

The types of STEP topological entities that can be translated are:

- vertices
- edges
- loops
- faces
- shells
- solids For further information see Mapping STEP entities to Open CASCADE Technology shapes.

**Geometrical entities**

The types of STEP geometrical entities that can be translated are:

- points
- vectors
- directions
- curves
- surfaces

For further information see 2.4 Mapping STEP entities to Open CASCADE Technology shapes.
Description of the process

Loading the STEP file

Before performing any other operation you have to load the file with:

```csharp
STEPControl_Reader reader;
IFSelect_ReturnStatus stat =
    reader.ReadFile(;filename.stp;);
```

Loading the file only memorizes the data, it does not translate it.

Checking the STEP file

This step is not obligatory. Check the loaded file with:

```csharp
reader.PrintCheckLoad(failsonly, mode);
```

Error messages are displayed if there are invalid or incomplete STEP entities, giving you the information on the cause of error.

If `failsonly` is true only fail messages are displayed. All messages are displayed if `failsonly` is false. Your analysis of the file can be either message-oriented or entity-oriented. Choose your preference with:

```csharp
IFSelect_PrintCount mode = IFSelect_xxx
```

Where `xxx` can be one of the following:

- `ItemsByEntity` – gives a sequential list of all messages per STEP entity,
- `CountByItem` – gives the number of STEP entities with their types per message
- `ListByItem` – gives the number of STEP entities with their types and rank numbers per message

Setting the translation parameters
The following parameters can be used to translate a STEP file into an OCCT shape.

If you give a value that is not within the range of possible values it will simply be ignored.

**read.precision.mode**

Defines which precision value will be used during translation (see section 2.5 below for details on precision and tolerances).

- *File (0)* – the precision value is set to length_measure in uncertainty_measure_with_unit from STEP file.
- *User (1)* – the precision value is that of the `read.precision.val` parameter.

Read this parameter with:

```cpp
Standard_Integer ic = Interface_Static::IVal("read.precision.mode");
```

Modify this parameter with:

```cpp
if(!Interface_Static::SetIVal("read.precision.mode",1 ))
  .. error ..
```

Default value is File (0).

**read.precision.val:**

User defined precision value. This parameter gives the precision for shape construction when the `read.precision.mode` parameter value is 1. By default it is 0.0001, but can be any real positive (non null) value.

This value is a basic value of tolerance in the processor. The value is in millimeters, independently of the length unit defined in the STEP file.

Read this parameter with:
Standard_Real rp =
  Interface_Static::RVal("read.precision.val");

Modify this parameter with:

    if(!Interface_Static::SetRVal("read.precision.val",0.01))
    .. error ..

By default this value is 0.0001.

The value given to this parameter is a basic value for ShapeHealing algorithms and the processor. It does its best to reach it. Under certain circumstances, the value you give may not be attached to all of the entities concerned at the end of processing. STEP-to-OpenCASCADE translation does not improve the quality of the geometry in the original STEP file. This means that the value you enter may be impossible to attach to all shapes with the given quality of the geometry in the STEP file.

**read.maxprecision.val**

Defines the maximum allowed tolerance (in mm) of the shape. It should be not less than the basic value of tolerance set in the processor (either the uncertainty from the file or `read.precision.val`). Actually, the maximum between `read.maxprecision.val` and the basis tolerance is used to define the maximum allowed tolerance.

Read this parameter with:

    Standard_Real rp =
      Interface_Static::RVal("read.maxprecision.val");

Modify this parameter with:

    if(!Interface_Static::SetRVal("read.maxprecision.val",0.1))
    .. error ..
Default value is 1. Note that maximum tolerance even explicitly defined by the user may be insufficient to ensure the validity of the shape (if real geometry is of bad quality). Therefore the user is provided with an additional parameter, which allows him to choose: either he prefers to ensure the shape validity or he rigidly sets the value of maximum tolerance. In the first case there is a possibility that the tolerance will not have any upper limit, in the second case the shape may be invalid.

**read.maxprecision.mode:**

Defines the mode of applying the maximum allowed tolerance. Its possible values are:

- 0 (Preferred) – maximum tolerance is used as a limit but sometimes it can be exceeded (currently, only for deviation of a 3D curve and pcurves of an edge, and vertices of such edge) to ensure the shape validity,
- 1 (Forced) – maximum tolerance is used as a rigid limit, i.e. no tolerance can exceed it and if it is the case, the tolerance is trimmed by the maximum tolerance.

Read this parameter with:

```plaintext
Standard_Integer ic = Interface_Static::IVal("read.maxprecision.mode") ;
```

Modify this parameter with:

```plaintext
if(!Interface_Static::SetIVal("read.maxprecision.mode ",1))
  .. error ..
```

Default value is 0 ("Preferred").

**read.stdsameparameter.mode**

defines the use of `BRepLib::SameParameter`. Its possible values are:
- 0 (Off) – BRepLib::SameParameter is not called,
- 1 (On) – BRepLib::SameParameter is called. The functionality of BRepLib::SameParameter is used through ShapeFix_Edge::SameParameter. It ensures that the resulting edge will have the lowest tolerance taking pcurves either unmodified from the STEP file or modified by BRepLib::SameParameter.

Read this parameter with:

```cpp
Standard_Integer mv = Interface_Static::IVal("read.stdsameparameter.mode");
```

Modify this parameter with:

```cpp
if (!Interface_Static::SetIVal("read.stdsameparameter.mode",1))
    .. error ..;
```

Default value is 0 (;Off;).

**read.surfacecurve.mode:**

a preference for the computation of curves in an entity which has both 2D and 3D representation. Each TopoDS_Edge in TopoDS_Face must have a 3D and 2D curve that references the surface.

If both 2D and 3D representation of the entity are present, the computation of these curves depends on the following values of parameter:

- **Default (0)**: no preference, both curves are taken (default value),
- **3DUse_Preferred (3)**: 3D curves are used to rebuild 2D ones.

Read this parameter with:

```cpp
Standard_Integer rp = Interface_Static::IVal("read.surfacecurve.mode");
```
Modify this parameter with:

```cpp
if(!Interface_Static::SetIVal("read.surfacecurve.mode ",3))
.. error ..
```

Default value is (0).

**read.encoderegularity.angle**

This parameter is used for call to `BRepLib::EncodeRegularity()` function which is called for the shape read from an IGES or a STEP file at the end of translation process. This function sets the regularity flag of the edge in the shell when this edge is shared by two faces. This flag shows the continuity these two faces are connected with at that edge. Read this parameter with:

```cpp
Standard_Real era =
    Interface_Static::RVal("read.encoderegularity.angle");
```

Modify this parameter with:

```cpp
if (!Interface_Static::SetRVal
         ("read.encoderegularity.angle",0.1))
.. error ..;
```

Default value is 0.01.

**step.angleunit.mode**

This parameter is obsolete (it was required in the past for STEP files with a badly encoded angle unit). It indicates what angle units should be used when a STEP file is read: the units from file (default), or forced RADIANS or DEGREES.

Default value is File

**read.step.resource.name and read.step.sequence**
These two parameters define the name of the resource file and the name of the sequence of operators (defined in that file) for Shape Processing, which is automatically performed by the STEP translator. Shape Processing is a user-configurable step, which is performed after translation and consists in applying a set of operators to a resulting shape. This is a very powerful tool allowing customizing the shape and adapting it to the needs of a receiving application. By default the sequence consists of a single operator ShapeFix – that is how Shape Healing is called from the STEP translator.

Please find an example of the resource file for STEP (which defines parameters corresponding to the sequence applied by default, i.e. if the resource file is not found) in the Open CASCADE Technology installation, by the path `CASROOT%/src/XSTEPResource/STEP`.

In order for the STEP translator to use that file, you have to define the `CSF_STEPDefaults` environment variable, which should point to the directory where the resource file resides. Note that if you change parameter `read.step.resource.name`, you will change the name of the resource file and the environment variable correspondingly.

Default values:

- `read.step.resource.name` – STEP,
- `read.step.sequence` – FromSTEP.

**xstep.cascade.unit**

This parameter defines units to which a shape should be converted when translated from IGES or STEP to CASCADE. Normally it is MM; only those applications that work internally in units other than MM should use this parameter.

Default value is MM.

**read.step.product.mode:**

Defines the approach used for selection of top-level STEP entities for translation, and for recognition of assembly structures.
• **1 (ON)** – *PRODUCT_DEFINITION* entities are taken as top-level ones; assembly structure is recognized by *NEXT_ASSEMBLY_USAGE_OCCURRENCE* entities. This is regular mode for reading valid STEP files conforming to AP 214, AP203 or AP 209.

• **0 (OFF)** – *SHAPE_DEFINITION_REPRESENTATION* entities are taken as top-level ones; assembly is recognized by *CONTEXT_DEPENDENT_SHAPE_REPRESENTATION* entities. This is compatibility mode, which can be used for reading legacy STEP files produced by older versions of STEP translators and having incorrect or incomplete product information.

Read this parameter with:

```cpp
Standard_Integer ic = 
    Interface_Static::IVal("read.step.product.mode") 
; 
```

Modify this parameter with:

```cpp
if(!Interface_Static::SetIVal("read.step.product.mode 
",1))
.. error ..
```

Default value is 1 (ON).

Note that the following parameters have effect only if *read.step.product.mode* is ON.

**read.step.product.context:**

When reading AP 209 STEP files, allows selecting either only `design` or `analysis`, or both types of products for translation

• **1 (all)** – translates all products;

• **2 (design)** – translates only products that have *PRODUCT_DEFINITION_CONTEXT* with field *life_cycle_stage* set to `design`;

• **3 (analysis)** – translates only products associated with *PRODUCT_DEFINITION_CONTEXT* entity whose field
**life_cycle_stage** set to `analysis`.

Note that in AP 203 and AP214 files all products should be marked as `design`, so if this mode is set to `analysis`, nothing will be read.

Read this parameter with:

```cpp
Standard_Integer ic = Interface_Static::IVal("read.step.product.context");
```

Modify this parameter with:

```cpp
if(!Interface_Static::SetIVal(;read.step.product.context;,1))
.. error ..
```

Default value is 1 (all).

**read.step.shape.repr:**

Specifies preferred type of representation of the shape of the product, in case if a STEP file contains more than one representation (i.e. multiple PRODUCT_DEFINITION_SHAPE entities) for a single product

- 1 (All) – Translate all representations (if more than one, put in compound).
- 2 (ABSR) - Prefer ADVANCED_BREP_SHAPE_REPRESENTATION
- 3 (MSSR) – Prefer MANIFOLD_SURFACE_SHAPE_REPRESENTATION
- 4 (GBSSR) – Prefer GEOMETRICALLY_BOUNDED_SURFACE_SHAPE_REPRESENTATION
- 5 (FBSR) – Prefer FACETTED_BREP_SHAPE_REPRESENTATION
- 6 (EBWSR) – Prefer EDGE_BASED_WIREFRAME_SHAPE_REPRESENTATION
- 7 (GBWSR) – Prefer GEOMETRICALLY_BOUNDED_WIREFRAME_SHAPE_REPRESENTATION

When this option is not equal to 1, for products with multiple representations the representation having a type closest to the selected
one in this list will be translated.

Read this parameter with:

```plaintext
Standard_Integer ic = Interface_Static::IVal("read.step.shape.repr");
```

Modify this parameter with:

```plaintext
if(!Interface_Static::SetIVal("read.step.shape.repr", 1))
  .. error ..
```

Default value is 1 (All).

**read.step.assembly.level:**

Specifies which data should be read for the products found in the STEP file:

- 1 (All) – Translate both the assembly structure and all associated shapes. If both shape and sub-assemblies are associated with the same product, all of them are read and put in a single compound. Note that this situation is confusing, as semantics of such configuration is not defined clearly by the STEP standard (whether this shape is an alternative representation of the assembly or is an addition to it), therefore warning will be issued in such case.
- 2 (assembly) – Translate the assembly structure and shapes associated with parts only (not with sub-assemblies).
- 3 (structure) – Translate only the assembly structure without shapes (a structure of empty compounds). This mode can be useful as an intermediate step in applications requiring specialized processing of assembly parts.
- 4 (shape) – Translate only shapes associated with the product, ignoring the assembly structure (if any). This can be useful to translate only a shape associated with specific product, as a complement to assembly mode.

Read this parameter with:
Standard_Integer ic =
    Interface_Static::IVal("read.step.assembly.level ");

Modify this parameter with:

if(!Interface_Static::SetIVal("read.step.assembly.level",1))
.. error ..

Default value is 1 (All).

**read.step.shape.relationship:**

Defines whether shapes associated with the main
SHAPE_DEFINITION_REPRESENTATION entity of the product via
SHAPE_REPRESENTATIONSHIP_RELATION should be translated. This
kind of association is used for the representation of hybrid models (i.e.
models whose shape is composed of different types of representations)
in AP 203 files since 1998, but it can be also used to associate auxiliary
data with the product. This parameter allows to avoid translation of such
auxiliary data.

- 1 (ON) – translate
- 0 (OFF) – do not translate

Read this parameter with:

Standard_Integer ic =
    Interface_Static::IVal("read.step.shape.relation ship");

Modify this parameter with:

if(!Interface_Static::SetIVal("read.step.shape.relati onship",1))
.. error ..

Default value is 1 (ON).
read.step.shape.aspect:

Defines whether shapes associated with the PRODUCT_DEFINITION_SHAPE entity of the product via SHAPE_ASPECT should be translated. This kind of association was used for the representation of hybrid models (i.e. models whose shape is composed of different types of representations) in AP 203 files before 1998, but it is also used to associate auxiliary information with the sub-shapes of the part. Though STEP translator tries to recognize such cases correctly, this parameter may be useful to avoid unconditionally translation of shapes associated via SHAPE_ASPECT entities.

- 1 (ON) – translate
- 0 (OFF) – do not translate

Read this parameter with:

```
Standard_Integer ic = 
    Interface_Static::IVal("read.step.shape.aspect")
```

Modify this parameter with:

```
if(!Interface_Static::SetIVal(;read.step.shape.aspect ;,1))
.. error ..
```

Default value is 1 (ON).

**Performing the STEP file translation**

Perform the translation according to what you want to translate. You can choose either root entities (all or selected by the number of root), or select any entity by its number in the STEP file. There is a limited set of types of entities that can be used as starting entities for translation. Only the following entities are recognized as transferable:

- product_definition
- next_assembly_usage_occurrence
- shape_definition_representation
- subtypes of shape_representation (only if referred representation is transferable)
- manifold_solid_brep
- brep_with_voids
- faceted_brep
- faceted_brep_and_brep_with_voids
- shell_based_surface_model
- geometric_set and geometric_curve_set
- mapped_item
- subtypes of face_surface (including advanced_face)
- subtypes of shape_representation_relationship
- context_dependent_shape_representation

The following methods are used for translation:

- `Standard_Boolean ok = reader.TransferRoot(rank)` – translates a root entity identified by its rank;
- `Standard_Boolean ok = reader.TransferOne(rank)` – translates an entity identified by its rank;
- `Standard_Integer num = reader.TransferList(list)` – translates a list of entities in one operation (this method returns the number of successful translations);

**Getting the translation results**

Each successful translation operation outputs one shape. A series of translations gives a set of shapes.

Each time you invoke `TransferOne()`, `TransferRoot()` or `TransferList()`, their results are accumulated and the counter of results increases. You can clear the results with:

```
reader.ClearShapes();
```

between two translation operations, if you do not, the results from the next translation will be added to the accumulation.

`TransferRoots()` operations automatically clear all existing results before
they start.

- `Standard_Integer num = reader.NbShapes()` – gets the number of shapes recorded in the result;
- `TopoDS_Shape shape = reader.Shape(rank)` – gets the result identified by its rank, where rank is an integer between 1 and `NbShapes`;
- `TopoDS_Shape shape = reader.Shape()` – gets the first result of translation;
- `TopoDS_Shape shape = reader.OneShape()` – gets all results in a single shape, which is:
  - a null shape if there are no results,
  - in case of a single result, a shape that is specific to that result,
  - a compound that lists the results if there are several results.

**Clearing the accumulation of results**

If several individual translations follow each other, the results give a list that can be purged with `reader.ClearShapes()`, which erases the existing results.

**Checking that translation was correctly performed**

Each time you invoke `Transfer` or `TransferRoots()`, you can display the related messages with the help of:

```
reader.PrintCheckTransfer(failsonly, mode);
```

This check concerns the last invocation of `Transfer` or `TransferRoots()` only.

**Selecting STEP entities for translation**

**Selection possibilities**

There are three selection possibilities. You can select:

- the whole file,
- a list of entities,
one entity.

The whole file

Transferring the whole file means transferring all root entities. The number of roots can be evaluated when the file is loaded:

```cpp
Standard_Integer NbRoots = reader.NbRootsForTransfer();
Standard_Integer num = reader.TransferRoots();
```

List of entities

A list of entities can be formed by invoking
`STEP214Control_Reader::GiveList` (this is a method of the parent class).

Here is a simple example of how a list is translated:

```cpp
Handle(TColStd_HSequenceOfTransient) list = reader.GiveList();
```

The result is a `TColStd_HSequenceOfTransient`. You can either translate a list entity by entity or all at once. An entity-by-entity operation lets you check each individual entity translated.

Translating a whole list in one operation

```cpp
Standard_Integer nbtrans = reader.TransferList(list);
```

`nbtrans` gives the number of items in the list that produced a shape.

Translating a list entity by entity:

```cpp
Standard_Integer i,nb = list->Length();
for (i = 1; i <= nb; i ++) {
    Handle(Standard_Transient) ent = list->Value(i);
    Standard_Boolean OK = reader.TransferEntity (ent);
}
Selections

There is a number of predefined operators that can be used. They are:

- \textit{step214-placed-items} – selects all mapped\_items or context\_depended\_shape\_representations.
- \textit{step214-shape-def-repr} – selects all shape\_definition\_representations.
- \textit{step214-shape-repr} – selects all shape\_representations.
- \textit{step214-type(<entity\_type>)} – selects all entities of a given type
- \textit{step214-faces} – selects all faces\_surface, advanced\_face entities and the surface entity or any sub type if these entities are not shared by any face entity or shared by geometric\_set entity.
- \textit{step214-derived(<entity\_type>)} – selects entities of a given type or any subtype.
- \textit{step214-GS-curves} – selects all curve entities or any subtype except the composite\_curve if these entities are shared by the geometric\_set entity.
- \textit{step214-assembly} – selects all mapped\_items or context\_depended\_shape\_representations involved into the assembly structure.
- \textit{xst-model-all} – selects all entities.
- \textit{xst-model-roots} – selects all roots.
- \textit{xst-shared + <selection>} – selects all entities shared by at least one entity selected by selection.
- \textit{xst-sharing + <selection>} – selects all entities sharing at least one entity selected by selection.
- \textit{xst-transferrable-all} – selects all transferable entities.
- \textit{xst-transferrable-roots} – selects all translatable roots. Cumulative lists can be used as well.

Single entities

You can select an entity either by its rank or by its handle (an entity's handle can be obtained by invoking the \texttt{StepData\_StepModel::Entity} function).

Selection by rank
Use method `StepData_StepModel::NextNumberForLabel` to find its rank with the following:

```cpp
Standard_CString label = `#...';
StepData_StepModel model = reader.StepModel();
rank = model->NextNumberForLabe(label, 0,
    Standard_False);
```

Translate an entity specified by its rank:

```cpp
Standard_Boolean ok = reader.Transfer(rank);
```

**Direct selection of an entity**

`ent` is the entity. The argument is a `Handle(Standard_Transient)`.

```cpp
Standard_Boolean ok = reader.TransferEntity (ent);
```
Mapping STEP entities to Open CASCADE Technology shapes

Tables given in this paragraph show the mapping of STEP entities to OCCT objects. Only topological and geometrical STEP entities and entities defining assembly structures are described in this paragraph. For a full list of STEP entities please refer to Appendix A.

Assembly structure representation entities

Not all entities defining the assembly structure in the STEP file are translated to OCCT shapes, but they are used to identify the relationships between assemblies and their components. Since the graph of `natural' dependencies of entities based on direct references between them does not include the references from assemblies to their components, these dependencies are introduced in addition to the former ones. This is made basing on the analysis of the following entities describing the structure of the assembly.

<table>
<thead>
<tr>
<th>STEP entity type</th>
<th>CASCADE shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>product_definition</td>
<td>A TopoDS_Comp for assemblies, CASCADE shape corresponding to the component of the assembly.</td>
</tr>
<tr>
<td>product_definition_shape</td>
<td>A TopoDS_Comp for assemblies,</td>
</tr>
<tr>
<td>shape_definition_representation</td>
<td>CASCADE shape representing the component for components</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>next_assembly_usage_occurence</td>
<td></td>
</tr>
<tr>
<td>mapped_item</td>
<td>TopoDS_Shape</td>
</tr>
<tr>
<td>context_dependent_shape_representation</td>
<td>TopoDS_Shape</td>
</tr>
<tr>
<td>shape_representation_relationship_with_transformation</td>
<td></td>
</tr>
<tr>
<td>item_defined_transformation</td>
<td></td>
</tr>
<tr>
<td>cartesian_transformation_operator</td>
<td></td>
</tr>
</tbody>
</table>

**Models**

<table>
<thead>
<tr>
<th>STEP entity type</th>
<th>CASCADE shape</th>
<th>Comments</th>
</tr>
</thead>
</table>
### Solid Models

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>brep_with_voids</td>
<td>TopoDS_Solid</td>
</tr>
<tr>
<td>faceted_brep</td>
<td>TopoDS_Solid</td>
</tr>
<tr>
<td>manifold_solid_brep</td>
<td>TopoDS_Solid</td>
</tr>
</tbody>
</table>

### Surface Models

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>shell_based_surface_model</td>
<td>TopoDS_Compound</td>
<td><em>shell_based_surface</em> is translated into one or more <em>TopoDS_Shell</em> grouped in a <em>TopoDS_Compound</em></td>
</tr>
<tr>
<td>geometric_set</td>
<td>TopoDS_Compound</td>
<td><em>TopoDS_Compound</em> contains only <em>TopoDS_Faces</em>, <em>TopoDS_Wires</em>, <em>TopoDS_Edges</em> and/or <em>TopoDS_Vertices</em>.</td>
</tr>
</tbody>
</table>

### Wireframe Models

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>geometric_curve_set</td>
<td>TopoDS_Compound</td>
<td><em>TopoDS_Compound</em> contains only <em>TopoDS_Wires</em>, <em>TopoDS_Edges</em> and/or <em>TopoDS_Vertices</em>.</td>
</tr>
</tbody>
</table>

### Topological entities

<table>
<thead>
<tr>
<th>Topology</th>
<th>STEP entity type</th>
<th>CASCADE shape</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>vertex_point</td>
<td>TopoDS_Vertex</td>
<td></td>
</tr>
<tr>
<td>Edges</td>
<td>oriented_edge</td>
<td>TopoDS_Edge</td>
<td></td>
</tr>
<tr>
<td></td>
<td>edge_curve</td>
<td>TopoDS_Edge</td>
<td></td>
</tr>
<tr>
<td>Loops</td>
<td>face_bound</td>
<td>TopoDS_Wire</td>
<td></td>
</tr>
<tr>
<td></td>
<td>face_outer_bound</td>
<td>TopoDS_Wire</td>
<td></td>
</tr>
<tr>
<td></td>
<td>edge_loop</td>
<td>TopoDS_Wire</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Each segment of poly_loop</em> is</td>
</tr>
</tbody>
</table>
poly_loop | TopoDS_Wire | translated into TopoDS_Edge with support of Geom_Line
vertex_loop | TopoDS_Wire | Resulting TopoDS_Wire contains only one degenerated TopoDS_Edge

| Faces | face_surface | TopoDS_Face |
|       | advanced_face | TopoDS_Face |

| Shells | connected_face_set | TopoDS_Shell |
|        | oriented_closed_shell | TopoDS_Shell |
|        | closed_shell | TopoDS_Shell |
|        | open_shell | TopoDS_Shell |

**Geometrical entities**

3D STEP entities are translated into geometrical objects from the Geom package while 2D entities are translated into objects from the Geom2d package.

<table>
<thead>
<tr>
<th>Geometry</th>
<th>STEP entity type</th>
<th>CASCADE object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>cartesian_point</td>
<td>Geom_CartesianPoint, Geom2d_CartesianPoint</td>
</tr>
<tr>
<td>Directions</td>
<td>direction</td>
<td>Geom_Direction, Geom2d_Direction</td>
</tr>
<tr>
<td>Vectors</td>
<td>vector</td>
<td>Geom_VectorWithMagnitude, Geom2d_VectorWithMagnitude</td>
</tr>
<tr>
<td>Placements</td>
<td>axis1_placement</td>
<td>Geom_Axis1Placement</td>
</tr>
<tr>
<td></td>
<td>axis2_placement_2d</td>
<td>Geom2d_AxisPlacement</td>
</tr>
<tr>
<td></td>
<td>axis2_placement_3d</td>
<td>Geom_Axis2Placement</td>
</tr>
<tr>
<td>Curves</td>
<td>circle</td>
<td>Geom_Circle, Geom2d_C, Geom2d_BsplineCurve</td>
</tr>
<tr>
<td>curve type</td>
<td>Geometric Types</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------</td>
<td></td>
</tr>
<tr>
<td>ellipse</td>
<td>Geom_Ellipse, Geom2d_Ellipse, Geom2d_BsplineCurve</td>
<td></td>
</tr>
<tr>
<td>hyperbola</td>
<td>Geom_Hyperbola, Geom2d_Hyperbola</td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>Geom_Line, Geom2d_Line</td>
<td></td>
</tr>
<tr>
<td>parabola</td>
<td>Geom_Parabola, Geom2d_Parabola</td>
<td></td>
</tr>
<tr>
<td>pcurve</td>
<td>Geom2d_Curve</td>
<td></td>
</tr>
<tr>
<td>curve_replica</td>
<td>Geom_Curve or Geom2d_Curve</td>
<td></td>
</tr>
<tr>
<td>offset_curve_3d</td>
<td>Geom_OffsetCurve</td>
<td></td>
</tr>
<tr>
<td>trimmed_curve</td>
<td>Geom_TrimmedCurve or Geom2d_BsplineCurve</td>
<td></td>
</tr>
<tr>
<td>b_spline_curve</td>
<td>Geom_BsplineCurve or Geom2d_BsplineCurve</td>
<td></td>
</tr>
<tr>
<td>b_spline_curve_with_knots</td>
<td>Geom_BsplineCurve or Geom2d_BsplineCurve</td>
<td></td>
</tr>
<tr>
<td>bezier_curve</td>
<td>Geom_BsplineCurve or Geom2d_BsplineCurve</td>
<td></td>
</tr>
<tr>
<td>rational_b_spline_curve</td>
<td>Geom_BsplineCurve or Geom2d_BsplineCurve</td>
<td></td>
</tr>
<tr>
<td>uniform_curve</td>
<td>Geom_BsplineCurve or Geom2d_BsplineCurve</td>
<td></td>
</tr>
<tr>
<td>quasi_uniform_curve</td>
<td>Geom_BsplineCurve or Geom2d_BsplineCurve</td>
<td></td>
</tr>
<tr>
<td>Surface Type</td>
<td>Geometric Representation</td>
<td></td>
</tr>
<tr>
<td>------------------------------------</td>
<td>--------------------------</td>
<td></td>
</tr>
<tr>
<td>surface_curve</td>
<td>TopoDS_Edge</td>
<td></td>
</tr>
<tr>
<td>seam_curve</td>
<td>TopoDS_Edge</td>
<td></td>
</tr>
<tr>
<td>composite_curve_segment</td>
<td>TopoDS_Edge</td>
<td></td>
</tr>
<tr>
<td>composite_curve</td>
<td>TopoDS_Wire</td>
<td></td>
</tr>
<tr>
<td>composite_curve_on_surface</td>
<td>TopoDS_Wire</td>
<td></td>
</tr>
<tr>
<td>boundary_curve</td>
<td>TopoDS_Wire</td>
<td></td>
</tr>
<tr>
<td>Surfaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b_spline_surface</td>
<td>Geom_BsplineSurface</td>
<td></td>
</tr>
<tr>
<td>b_spline_surface_with_knots</td>
<td>Geom_BsplineSurface</td>
<td></td>
</tr>
<tr>
<td>bezier_surface</td>
<td>Geom_BSplineSurface</td>
<td></td>
</tr>
<tr>
<td>conical_surface</td>
<td>Geom_ConicalSurface</td>
<td></td>
</tr>
<tr>
<td>cylindrical_surface</td>
<td>Geom_CylindricalSurface</td>
<td></td>
</tr>
<tr>
<td>offset_surface</td>
<td>Geom_OffsetSurface</td>
<td></td>
</tr>
<tr>
<td>surface_replica</td>
<td>Geom_Surface</td>
<td></td>
</tr>
<tr>
<td>plane</td>
<td>Geom_Plane</td>
<td></td>
</tr>
<tr>
<td>rational_b_spline_surface</td>
<td>Geom_BSplineSurface</td>
<td></td>
</tr>
<tr>
<td>rectangular_trimmed_surface</td>
<td>Geom_RectangularTrimmedSurface</td>
<td></td>
</tr>
<tr>
<td>spherical_surface</td>
<td>Geom_SphericalSurface</td>
<td></td>
</tr>
<tr>
<td>surface_of_linear_extrusion</td>
<td>Geom_SurfaceOfLinearExtrusion</td>
<td></td>
</tr>
<tr>
<td>surface_of_revolution</td>
<td>Geom_SurfaceOfRevolution</td>
<td></td>
</tr>
<tr>
<td>toroidal_surface</td>
<td>Geom_ToroidalSurface</td>
<td></td>
</tr>
<tr>
<td>degenerate_toroidal_surface</td>
<td>Geom_ToroidalSurface</td>
<td></td>
</tr>
<tr>
<td>uniform_surface</td>
<td>Geom_BSplineSurface</td>
<td></td>
</tr>
<tr>
<td>quasi_uniform_surface</td>
<td>Geom_BSplineSurface</td>
<td></td>
</tr>
<tr>
<td>rectangular_composite_surface</td>
<td>TopoDS_Compound</td>
<td></td>
</tr>
<tr>
<td>curve_bounded_surface</td>
<td>TopoDS_Face</td>
<td></td>
</tr>
</tbody>
</table>
Tolerance management

Values used for tolerances during reading STEP

During the STEP to OCCT translation several parameters are used as tolerances and precisions for different algorithms. Some of them are computed from other tolerances using specific functions.

3D (spatial) tolerance

- Package method `Precision::Confusion()` Value is 10^-7. It is used as the minimal distance between points, which are considered to be distinct.
- Uncertainty parameter is attached to each shape_representation entity in a STEP file and defined as `length_measure` in `uncertainty_measure_with_unit`. It is used as a fundamental value of precision during translation.
- User-defined variable `read.precision.val` is used instead of uncertainty from a STEP file when parameter `read.precision.mode` is 1 (User).

2D (parametric) tolerances

- Package method `Precision::PConfusion()` is a value of `0.01*Precision::Confusion()`. It is used to compare parametric bounds of curves.
- Methods `UResolution` and `VResolution` (tolerance3d) of the class `GeomAdaptor_Surface` or `BRepAdaptor_Surface` return tolerance in parametric space of a surface computed from 3d tolerance. When one tolerance value is to be used for both U and V parametric directions, the maximum or the minimum value of `UResolution` and `VResolution` is used.
- Methods `Resolution` (tolerance3d) of the class `GeomAdaptor_Curve` or `BRepAdaptor_Curve` return tolerance in parametric space of a curve computed from 3d tolerance.

Initial setting of tolerances in translating objects
In the STEP processor, the basic value of tolerance is set in method
`STEPControl_ActorRead::Transfer()` to either value of uncertainty in
shape_representation in STEP file (if parameter `read.precision.mode` is
0), or to a value of parameter `read.precision.val` (if `read.precision.mode` is
1 or if the uncertainty is not attached to the current entity in the STEP
file).

Translation starts from one entity translated as a root.
`STEPControl_ActorRead::Transfer()`, function which performs the
translation creates an object of the type `StepToTopoDS_Builder`, which is
intended to translate topology.

This object gets the initial tolerance value that is equal to
`read.precision.val` or the uncertainty from shape_representation. During
the translation of the entity, new objects of types
`StepToTopoDS_Translate...` are created for translating sub-entities. All of
them use the same tolerances as a `StepToTopoDS_Builder` object.

Transfer process

Evolution of shape tolerances during transfer

Let us follow the evolution of tolerances during the translation of STEP
entities into an OCCT shape.

If the starting STEP entity is a geometric_curve_set all the edges and
vertices are constructed with `Precision::Confusion()`.

If the starting STEP entity is not a geometric_curve_set the sub-shapes
of the resulting shape have the following tolerance:

- all the faces are constructed with `Precision::Confusion()`.
- edges are constructed with `Precision::Confusion()`. It can be modified
  later by:
  - `ShapeFix::SameParameter()` – the tolerance of edge shows real
development of the 3D curve and pcurves.
  - `ShapeFix_Wire::FixSelfIntersection()` if a pcurve of a self-intersecting
    edge is modified.
- vertices are constructed with `Precision::Confusion()`. It can be
  modified later by: `StepToTopoDS_TranslateEdge`
So, the final tolerance of sub-shapes shows the real local geometry of shapes (distance between vertices of adjacent edges, deviation of a 3D curve of an edge and its parametric curves and so on) and may be less or greater than the basic value of tolerance in the STEP processor.

Translating into Geometry

Geometrical entities are translated by classes StepToGeom_Make... Methods of these classes translate STEP geometrical entities into OCCT geometrical objects. Since these objects are not BRep objects, they do not have tolerances. Tolerance is used only as precision for detecting bad cases (such as points coincidence).

Translating into Topology

STEP topological entities are translated into OCCT shapes by use of classes from package StepToTopoDS.

Although in a STEP file the uncertainty value is assigned to shape_representation entities and this value is applied to all entities in this shape_representation, OCCT shapes are produced with different tolerances. As a rule, updating the tolerance is fulfilled according to the local geometry of shapes (distance between vertices of adjacent edges, deviation of edge's 3D curve and its parametric curves and so on) and may be either less or greater than the uncertainty value assigned to the entity.

The following default tolerances are used when creating shapes and how they are updated during translation.

- StepToTopoDS_TranslateVertex constructs TopoDS_Vertex from a STEP vertex_point entity with Precision::Confusion().
- StepToTopoDS_TranslateVertexLoop creates degenerated TopoDS_Edge in TopoDS_Wire with tolerance Precision::Confusion(). TopoDS_Vertex of a degenerated edge is constructed with the initial value of tolerance.
- StepToTopoDS_TranslateEdge constructs TopoDS_Edge only on the
basis of 3D curve with `Precision::Confusion()`. Tolerance of the vertices can be increased up to a distance between their positions and ends of 3D curve.

- `StepToTopoDS_TranslateEdgeLoop` constructs `TopoDS_Edges` in `TopoDS_Wire` with help of class `StepToTopoDS_TranslateEdge`. Pcurves from a STEP file are translated if they are present and `read.surfacecurve.mode` is 0. For each edge method `ShapeFix_Edge::FixSameParameter()` is called. If the resulting tolerance of the edge is greater than the maximum value between 1.0 and 2*Value of basis precision, then the pcurve is recomputed. The best of the original and the recomputed pcurve is put into `TopoDS_Edge`. The resulting tolerance of `TopoDS_Edge` is a maximal deviation of its 3D curve and its pcurve(s).

- `StepToTopoDS_TranslatePolyLoop` constructs `TopoDS_Edges` in `TopoDS_Wire` with help of class `StepToTopoDS_TranslateEdge`. Their tolerances are not modified inside this method.

- `StepToTopoDS_TranslateFace` constructs `TopoDS_Face` with the initial value of tolerance. `TopoDS_Wire` on `TopoDS_Face` is constructed with the help of classes `StepToTopoDS_TranslatePolyLoop`, `StepToTopoDS_TranslateEdgeLoop` or `StepToTopoDS_TranslateVertexLoop`.

- `StepToTopoDS_TranslateShell` calls `StepToTopoDS_TranslateFace::Init` for each face. This class does not modify the tolerance value.

- `StepToTopoDS_TranslateCompositeCurve` constructs `TopoDS_Edges` in `TopoDS_Wire` with help of class `BRepAPI_MakeEdge` and have a tolerance 10-7. Pcurves from a STEP file are translated if they are present and if `read.surfacecurve.mode` is not -3. The connection between segments of a composite curve (edges in the wire) is provided by calling method `ShapeFix_Wire::FixConnected()` with a precision equal to the initial value of tolerance.

- `StepToTopoDS_TranslateCurveBoundedSurface` constructs `TopoDS_Face` with tolerance `Precision::Confusion()`. `TopoDS_Wire` on `TopoDS_Face` is constructed with the help of class `StepToTopoDS_TranslateCompositeCurve`. Missing pcurves are computed using projection algorithm with the help of method `ShapeFix_Face::FixPcurves()`. For resulting face method `ShapeFix::SameParameter()` is called. It calls standard
**BRepLib::SameParameter** for each edge in each wire, which can either increase or decrease the tolerances of the edges and vertices. **SameParameter** writes the tolerance corresponding to the real deviation of pcurves from 3D curve which can be less or greater than the tolerance in a STEP file.

- **StepToTopoDS_Builder** a high level class. Its methods perform translation with the help of the classes listed above. If the value of read.maxprecision.mode is set to 1 then the tolerance of subshapes of the resulting shape is limited by 0 and read.maxprecision.val. Else this class does not change the tolerance value.

- **StepToTopoDS_MakeTransformed** performs a translation of mapped_item entity and indirectly uses class StepToTopoDS_Builder. The tolerance of the resulting shape is not modified inside this method.

### Healing of resulting shape in ShapeHealing component

**ShapeFix_Wire::FixSelfIntersection()**

This method is intended for detecting and fixing self-intersecting edges and intersections of adjacent edges in a wire. It fixes self-intersections by cutting edges at the intersection point and/or by increasing the tolerance of the vertex (so that the vertex comprises the point of intersection). There is a maximum tolerance that can be set by this method transmitted as a parameter, currently is read.maxprecision.value.

When a self-intersection of one edge is found, it is fixed by one of the two methods:

- tolerance of the vertex of that edge which is nearest to the point of self-intersection is increased so that it comprises both its own old position and the intersection point
- the self-intersecting loop on the pcurve is cut out and a new pcurve is constructed. This can increase the tolerance of the edge.

The method producing a smaller tolerance is selected.

When an intersection of two adjacent edges is detected, edges are cut at that point. Tolerance of the common vertex of these edges is increased in order to comprise both the intersection point and the old position.
This method can increase the tolerance of the vertex up to a value of `read.maxprecision.value`.

**ShapeFix_Wire::FixLacking()**

This method is intended to detect gaps between pcurves of adjacent edges (with the precision of surface UVResolution computed from tolerance of a corresponding vertex) and to fix these gaps either by increasing the tolerance of the vertex, or by inserting a new degenerated edge (straight in parametric space).

If it is possible to compensate a gap by increasing the tolerance of the vertex to a value of less than the initial value of tolerance, the tolerance of the vertex is increased. Else, if the vertex is placed in a degenerated point then a degenerated edge is inserted.

**ShapeFix_Wire::FixConnected()**

This method is intended to force two adjacent edges in the wire to share the same vertex. This method can increase the tolerance of the vertex. The maximal value of tolerance is `read.maxprecision.value`. 
Code architecture

The following diagram illustrates the structure of calls in reading STEP. The highlighted classes are intended to translate geometry.

The structure of calls in reading STEP
Example

```c++
#include <STEPControl_Reader.hxx>
#include <TopoDS_Shape.hxx>
#include <BRepTools.hxx>

Standard_Integer main()
{
    STEPControl_Reader reader;
    reader.ReadFile("MyFile.stp");

    // Loads file MyFile.stp
    Standard_Integer NbRoots =
        reader.NbRootsForTransfer();

    // gets the number of transferable roots
    cout;Number of roots in STEP file: ; NbRootsendl;

    Standard_Integer NbTrans = reader.TransferRoots();
    // translates all transferable roots, and returns
    // the number of  //successful translations
    cout;STEP roots transferred: ; NbTransendl;
    cout;Number of resulting shapes is:
        ;reader.NbShapes()endl;

    TopoDS_Shape result = reader.OneShape();
    // obtain the results of translation in one OCCT
    shape

    ...
}
```
Writing STEP

Procedure

You can translate OCCT shapes into STEP entities in the following steps: 1. initialize the process, 2. set the translation parameters, 3. perform the shape translation, 4. write the output file.

You can translate several shapes before writing a file. All these translations output a separate shape_representation entity in STEP file.

The user-defined option (parameter write.step.schema) is provided to define which version of schema (AP214 CD or DIS, or AP203) is used for the output STEP file.
Domain covered

Writing geometry and topology

There are two families of OCCT objects that can be translated:

- geometrical objects,
- topological shapes.

Writing assembly structures

The shapes organized in a structure of nested compounds can be translated either as simple compound shapes, or into the assembly structure, depending on the parameter `write.step.assembly`, which is described below.

The assembly structure placed in the produced STEP file corresponds to the structure described in the ProSTEP Agreement Log (item 21) as the second alternative (assembly structure through `representation_relationship / item_defined_transformation`). To represent an assembly it uses entities of the `representation_relationship_with_transformation` type. Transformation operators used for locating assembly components are represented by `item_defined_transformation` entities. If mode `write.step.assembly` is set to the values `ON` or `Auto` then an OCC shape consisting of nested compounds will be written as an assembly, otherwise it will be written as separate solids.

Please see also Mapping OCCT shapes to STEP entities.
Description of the process

Initializing the process

Before performing any other operation you have to create a writer object:

```cpp
STEPControl_Writer writer;
```

Setting the translation parameters

The following parameters are used for the OCCT-to-STEP translation.

write.precision.mode

writes the precision value.

- Least (-1) : the uncertainty value is set to the minimum tolerance of an OCCT shape
- Average (0) : the uncertainty value is set to the average tolerance of an OCCT shape.
- Greatest (1) : the uncertainty value is set to the maximum tolerance of an OCCT shape
- Session (2) : the uncertainty value is that of the write.precision.val parameter.

Read this parameter with:

```cpp
Standard_Integer ic = Interface_Static::IVal("write.precision.mode");
```

Modify this parameter with:

```cpp
if(!Interface_Static::SetIVal("write.precision.mode", 1))
  .. error ..
```

Default value is 0.

write.precision.val
a user-defined precision value. This parameter gives the uncertainty for
STEP entities constructed from OCCT shapes when the
write.precision.mode parameter value is 1.

- 0.0001: default
- any real positive (non null) value.

This value is stored in shape_representation in a STEP file as an
uncertainty.

Read this parameter with:

```cpp
Standard_Real rp = Interface_Static::RVal("write.precision.val");
```

Modify this parameter with:

```cpp
if(!Interface_Static::SetRVal("write.precision.val",0.01))
  .. error ..
```

Default value is 0.0001.

**write.step.assembly**

writing assembly mode.

- 0 (Off) : (default) writes STEP files without assemblies.
- 1 (On) : writes all shapes in the form of STEP assemblies.
- 2 (Auto) : writes shapes having a structure of (possibly nested)
  `TopoDS_Compounds` in the form of STEP assemblies, single shapes
  are written without assembly structures.

Read this parameter with:

```cpp
Standard_Integer rp = Interface_Static::IVal("write.step.assembly");
```

Modify this parameter with:
if(!Interface_Static::SetIVal("write.step.assembly",1))
.. error..

Default value is 0.

**write.step.schema**

defines the version of schema used for the output STEP file:

- 1 or **AP214CD** (default): AP214, CD version (dated 26 November 1996),
- 2 or **AP214DIS**: AP214, DIS version (dated 15 September 1998).
- 3 or **AP203**: AP203, possibly with modular extensions (depending on data written to a file).
- 4 or **AP214IS**: AP214, IS version (dated 2002)
- 5 or **AP242DIS**: AP242, DIS version.

Read this parameter with:

```cpp
TCollection_AsciiString schema = Interface_Static::CVal("write.step.schema");
```

Modify this parameter with:

```cpp
if(!Interface_Static::SetCVal("write.step.schema","DIS"))
.. error..
```

Default value is 1 (;CD;). For the parameter **write.step.schema** to take effect, method `STEPControl_Writer::Model(Standard_True)` should be called after changing this parameter (corresponding command in DRAW is `newmodel`).

**write.step.product.name**

Defines the text string that will be used for field `name` of PRODUCT entities written to the STEP file.
Default value: OCCT STEP translator (current OCCT version number).

**write.surfacecurve.mode**

This parameter indicates whether parametric curves (curves in parametric space of surface) should be written into the STEP file. This parameter can be set to Off in order to minimize the size of the resulting STEP file.

- Off (0) : writes STEP files without pcurves. This mode decreases the size of the resulting STEP file .
- On (1) : (default) writes pcurves to STEP file

Read this parameter with:

```cpp
Standard_Integer wp = Interface_Static::IVal("write.surfacecurve.mode");
```

Modify this parameter with:

```cpp
if(!Interface_Static::SetIVal("write.surfacecurve.mode",1))
.. error ..
```

Default value is On.

**write.step.unit**

Defines a unit in which the STEP file should be written. If set to unit other than MM, the model is converted to these units during the translation.

Default value is MM.

**write.step.resource.name and write.step.sequence**

These two parameters define the name of the resource file and the name of the sequence of operators (defined in that file) for Shape Processing, which is automatically performed by the STEP translator before
translating a shape to a STEP file. Shape Processing is a user-configurable step, which is performed before the translation and consists in applying a set of operators to a resulting shape. This is a very powerful tool allowing customizing the shape and adapting it to the needs of a receiving application. By default the sequence consists of two operators: SplitCommonVertex and DirectFaces, which convert some geometry and topological constructs valid in Open CASCADE Technology but not in STEP to equivalent definitions conforming to STEP format.

See description of parameter read.step.resource.name above for more details on using resource files.

Default values:

- read.step.resource.name – STEP,
- read.step.sequence – ToSTEP.

**write.step.vertex.mode**

This parameter indicates which of free vertices writing mode is switch on.

- 0 (One Compound) : (default) All free vertices are united into one compound and exported in one SHAPE DEFINITION REPRESENTATION (vertex name and style are lost).
- 1 (Single Vertex) : Each vertex exported in its own SHAPE DEFINITION REPRESENTATION (vertex name and style are not lost, but size of STEP file increases).

Read this parameter with:

```cpp
Standard_Integer ic = Interface_Static::IVal("write.step.vertex.mode");
```

Modify this parameter with:

```cpp
if(!Interface_Static::SetIVal("write.step.vertex.mode","1"))
  .. error ..
```
Default value is 0.

**Performing the Open CASCADE Technology shape translation**

An OCCT shape can be translated to STEP using one of the following models (shape_representations):

- manifold_solid_brep (advanced_brep_shape_representation)
- brep_with_voids (advanced_brep_shape_representation)
- faceted_brep (faceted_brep_shape_representation)
- shell_based_surface_model (manifold_surface_shape_representation)
- geometric_curve_set (geometrically_bounded_wireframe_shape_representation)

The enumeration `STEPControl_StepModelType` is intended to define a particular transferring model. The following values of enumeration are allowed:

- `STEPControl_AsIs` Translator selects the resulting representation automatically, according to the type of CASCADE shape to translate it in its highest possible model;
- `STEPControl_ManifoldSolidBrep` resulting entity is manifold_solid_brep or brep_with_voids
- `STEPControl_FacetedBrep` resulting entity is faceted_brep or faceted_brep_and_brep_with_voids Note that only planar-face shapes with linear edges can be written;
- `STEPControl_ShellBasedSurfaceModel` resulting entity is shell_based_surface_model;
- `STEPControl_GeometricCurveSet` resulting entity is geometric_curve_set;

The following list shows which shapes can be translated in which mode:

- `STEP214Control_AsIs` – any OCCT shape
- `STEP214Control_ManifoldSolidBrep` – TopoDS_Solid, TopoDS_Shell, TopoDS_Comound (if it contains TopoDS_Solids and TopoDS_Shells).
- `STEP214Control_FacetedBrep` – TopoDS_Solid or
TopoDS_Compound containing TopoDS_Solids if all its surfaces are Geom_Planes and all curves are Geom_Lines.

- STEP214Control_ShellBasedSurfaceModel – TopoDS_Solid, TopoDS_Shell, TopoDS_Face and TopoDS_Compound (if it contains all mentioned shapes)
- STEP214Control_GeometricCurveSet – any OCCT shape.

If TopoDS_Compound contains any other types besides the ones mentioned in the table, these sub-shapes will be ignored.

In case if an OCCT shape cannot be translated according to its mode the result of translation is void.

```plaintext
STEP214Control_StepModelTope mode = STEP214Control_ManifoldSolidBrep;
IFSelect_ReturnStatus stat = writer.Transfer(shape,mode);
```

**Writing the STEP file**

Write the STEP file with:

```plaintext
IFSelect_ReturnStatus stat = writer.Write("filename.stp");
```

to give the file name.
Mapping Open CASCADE Technology shapes to STEP entities

Only STEP entities that have a corresponding OCCT object and mapping of assembly structures are described in this paragraph. For a full list of STEP entities please refer to Appendix A.

Assembly structures and product information

The assembly structures are written to the STEP file if parameter write.step.assembly is 1 or 2. Each TopoDS_Compound is written as an assembly with subshapes of that compound being components of the assembly. The structure of nested compounds is translated to the structure of nested assemblies. Shared subshapes are translated into shared components of assemblies. Shapes that are not compounds are translated into subtypes of shape_representation according to their type (see the next subchapter for details).

A set of STEP entities describing general product information is written to the STEP file together with the entities describing the product geometry, topology and assembly structure. Most of these entities are attached to the entities being subtypes of shape_representation, but some of them are created only one per STEP file.

The table below describes STEP entities, which are created when the assembly structure and product information are written to the STEP file, and shows how many of these entities are created. Note that the appearance of some of these entities depends on the version of the schema (AP214, CD, DIS or IS, or AP203).

<table>
<thead>
<tr>
<th>CASCADE shape</th>
<th>STEP entity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>application_protocol_definition</td>
</tr>
<tr>
<td></td>
<td>application_context</td>
</tr>
<tr>
<td>TopoDS_Compound</td>
<td>shape_representation</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>TopoDS_Shape</td>
<td>subtypes of shape_representation</td>
</tr>
<tr>
<td></td>
<td>next_assembly_usage_occurrence</td>
</tr>
<tr>
<td></td>
<td>context_dependent_shape_representation</td>
</tr>
<tr>
<td></td>
<td>shape_representation_relationship_with_transformation</td>
</tr>
<tr>
<td></td>
<td>item_defined_transformation</td>
</tr>
<tr>
<td></td>
<td>shape_definition_representation</td>
</tr>
<tr>
<td></td>
<td>product_definition_shape</td>
</tr>
<tr>
<td></td>
<td>product_definition</td>
</tr>
<tr>
<td></td>
<td>product_definition_formation</td>
</tr>
<tr>
<td>Product</td>
<td>product_type (CD) or product_related_product_category (DIS,IS)</td>
</tr>
<tr>
<td></td>
<td>Mechanical_context (CD) or product_context (DIS,</td>
</tr>
<tr>
<td></td>
<td>product_definition_context</td>
</tr>
</tbody>
</table>

**Topological shapes**
<table>
<thead>
<tr>
<th>CASCADE shape</th>
<th>STEP entity</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopoDS_Compound</td>
<td>geometric_curve_set</td>
<td>If the write mode is <code>STEP214Control_GeometricCurveSet</code>, 3D curves of the edges in <code>TopoDS_Compound</code> are translated.</td>
</tr>
<tr>
<td></td>
<td>manifold_solid_brep</td>
<td>If the write mode is <code>STEP214Control_AsIs</code> and <code>TopoDS_Compound</code> is a <code>TopoDS_Solids</code>, all <code>TopoDS_Solids</code> are translated.</td>
</tr>
<tr>
<td></td>
<td>shell_based_surface_model</td>
<td>If the write mode is <code>STEP214Control_AsIs</code> and <code>TopoDS_Compound</code> is a <code>TopoDS_Solids</code>, <code>TopoDS_Shells</code>, <code>TopoDS_Faces</code> are translated.</td>
</tr>
<tr>
<td></td>
<td>geometric_curve_set</td>
<td>If the write mode is <code>STEP214Control_AsIs</code> and <code>TopoDS_Compound</code> is a <code>TopoDS_Solids</code>, <code>TopoDS_Wires</code>, <code>TopoDS_Vertices</code>. If the write mode is not <code>STEP214Control_AsIs</code> or <code>STEP214Control_ManifoldSolidBrep</code>, this table is not applicable.</td>
</tr>
<tr>
<td>TopoDS_Solid</td>
<td>manifold_solid_brep</td>
<td>If the write mode is <code>STEP214Control_AsIs</code> or <code>STEP214Control_ManifoldSolidBrep</code>, <code>TopoDS_Solids</code>, <code>TopoDS_Shells</code>, <code>TopoDS_Faces</code> are translated.</td>
</tr>
<tr>
<td></td>
<td>faceted_brep</td>
<td>If the write mode is <code>STEP214Control_FacetedBrep</code>, <code>TopoDS_Solids</code>, <code>TopoDS_Shells</code>, <code>TopoDS_Faces</code> are translated.</td>
</tr>
<tr>
<td></td>
<td>brep_with_voids</td>
<td>If the write mode is <code>STEP214Control_FacetedBrep</code>, <code>TopoDS_Solids</code>, <code>TopoDS_Shells</code>, <code>TopoDS_Faces</code> are translated.</td>
</tr>
<tr>
<td></td>
<td>shell_based_surface_model</td>
<td>If the write mode is <code>STEP214Control_ShellBasedSurfaceModel</code>, <code>TopoDS_Solids</code>, <code>TopoDS_Shells</code>, <code>TopoDS_Faces</code> are translated.</td>
</tr>
<tr>
<td></td>
<td>geometric_curve_set</td>
<td>If the write mode is <code>STEP214Control_GeometricCurveSet</code>, 3D curves of the edges in <code>TopoDS_Solid</code> are translated.</td>
</tr>
<tr>
<td>TopoDS_Shell in a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TopoDS_Solid</td>
<td>closed_shell</td>
<td>If TopoDS_Shell is closed shell.</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>TopoDS_Shell</td>
<td>manifold_solid_brep</td>
<td>If the write mode is STEP214Control_ManifoldSolidBrep.</td>
</tr>
<tr>
<td></td>
<td>shell_based_surface_model</td>
<td>If the write mode is STEP214Control_ShellBasedSurfaceModel.</td>
</tr>
<tr>
<td></td>
<td>geometric_curve_set</td>
<td>If the write mode is STEP214Control_GeometricCurveSet. 3D curves of the edg</td>
</tr>
<tr>
<td>TopoDS_Face</td>
<td>advanced_face</td>
<td>The resulting face_bound poly_loop if write mode is edge_loop if it is not.</td>
</tr>
<tr>
<td>TopoDS_Wire in a TopoDS_Face</td>
<td>face_bound</td>
<td>The resulting face_bound poly_loop if write mode is edge_loop if it is not.</td>
</tr>
<tr>
<td>TopoDS_Wire</td>
<td>geometric_curve_set</td>
<td>If the write mode is STEP214Control_GeometricCurveSet. 3D curves of the edg</td>
</tr>
<tr>
<td>TopoDS_Edge</td>
<td>oriented_edge</td>
<td></td>
</tr>
<tr>
<td>TopoDS_Vertex</td>
<td>vertex_point</td>
<td></td>
</tr>
</tbody>
</table>

**Geometrical objects**

<table>
<thead>
<tr>
<th>Geometry</th>
<th>CASCADE object</th>
<th>STEP entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>Geom_CartesianPoint, Geom2d_CartesianPoint</td>
<td>cartesian_point</td>
</tr>
<tr>
<td></td>
<td>TColgp_Array1OfPnt, TColgp_Array1OfPnt2d</td>
<td>polyline</td>
</tr>
<tr>
<td>Placements</td>
<td>Geom_Axis1Placement, Geom2d_AxisPlacement</td>
<td>axis1_placement</td>
</tr>
<tr>
<td></td>
<td>Geom_Axis2Placement</td>
<td>axis2_placement_3d</td>
</tr>
<tr>
<td>Directions</td>
<td>Geom_Direction, Geom2d_Direction</td>
<td>direction</td>
</tr>
<tr>
<td>Vectors</td>
<td>Geom_Vector, Geom2d_Vector</td>
<td>vector</td>
</tr>
<tr>
<td>Curves</td>
<td>Geom_Circle</td>
<td>circle</td>
</tr>
<tr>
<td></td>
<td>Geom2d_Circle</td>
<td>circle, rational_b_spline_curve</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Geom_Ellipse</td>
<td>Ellipse</td>
<td></td>
</tr>
<tr>
<td>Geom2d_Ellipse</td>
<td>Ellipse, rational_b_spline_curve</td>
<td></td>
</tr>
<tr>
<td>Geom_Hyperbola, Geom2d_Hyperbola</td>
<td>Hyperbola</td>
<td></td>
</tr>
<tr>
<td>Geom_Parabola, Geom2d_Parabola</td>
<td>Parabola</td>
<td></td>
</tr>
<tr>
<td>Geom_BSplineCurve</td>
<td>b_spline_curve_with_knots or rational_b_spline_curve</td>
<td></td>
</tr>
<tr>
<td>Geom2d_BSplineCurve</td>
<td>b_spline_curve_with_knots or rational_b_spline_curve</td>
<td></td>
</tr>
<tr>
<td>Geom_BezierCurve</td>
<td>b_spline_curve_with_knots</td>
<td></td>
</tr>
<tr>
<td>Geom_Line or Geom2d_Line</td>
<td>Line</td>
<td></td>
</tr>
<tr>
<td>Surfaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geom_Plane</td>
<td>Plane</td>
<td></td>
</tr>
<tr>
<td>Geom_OffsetSurface</td>
<td>offset_surface</td>
<td></td>
</tr>
<tr>
<td>Geom_ConicalSurface</td>
<td>conical_surface</td>
<td></td>
</tr>
<tr>
<td>Geom_CylindricalSurface</td>
<td>cylindrical_surface</td>
<td></td>
</tr>
<tr>
<td>Geom_OffsetSurface</td>
<td>offset_surface</td>
<td></td>
</tr>
<tr>
<td>Geom_RectangularTrimmedSurface</td>
<td>rectangular_trimmed_surface</td>
<td></td>
</tr>
<tr>
<td>Geom_SphericalSurface</td>
<td>spherical_surface</td>
<td></td>
</tr>
<tr>
<td>Geom_SurfaceOfLinear Extrusion</td>
<td>surface_of_linear_extrusion</td>
<td></td>
</tr>
<tr>
<td>Geom_SurfaceOf Revolution</td>
<td>surface_of_revolution</td>
<td></td>
</tr>
<tr>
<td>Geom_ToroidalSurface</td>
<td>toroidal_surface or degenerate_toroidal_surface</td>
<td></td>
</tr>
<tr>
<td>Geom_BezierSurface</td>
<td>b_spline_surface_with_knots</td>
<td></td>
</tr>
<tr>
<td>Geom_BsplineSurface</td>
<td>b_spline_surface_with_knots or rational_b_spline_curve</td>
<td></td>
</tr>
</tbody>
</table>
Tolerance management

There are four possible values for the uncertainty when writing a STEP file:

- user-defined value of the uncertainty
- minimal value of sub-shapes tolerances
- average value of sub-shapes tolerances
- maximal value of sub-shapes tolerances

The chosen value of the uncertainty is the final value that will be written into the STEP file. See parameter `write.precision.mode`. 
Code architecture

Graph of calls

The following diagram illustrates the structure of calls in writing STEP. The highlighted classes are intended to translate geometry.
Example

```cpp
#include <STEPControl.hxx>
#include <STEPControl_Writer.hxx>
#include <TopoDS_Shape.hxx>
#include <BRepTools.hxx>
#include <BRep_Builder.hxx>

Standard_Integer main()
{
  TopoDS_Solid source;
  ...

  STEPControl_Writer writer;
  writer.Transfer(source,
                  STEPControl_ManifoldSolidBrep);

  // Translates TopoDS_Shape into manifold_solid_brep entity
  writer.Write();Output.stp);
  // writes the resulting entity in the STEP file
}
```
Physical STEP file reading and writing

Architecture of STEP Read and Write classes

General principles

To perform data loading from a STEP file and to translate this data it is necessary to create correspondence between the EXPRESS schema and the structure of the classes. There are two possibilities to organize such correspondence: the so-called early binding and late binding.

- Late binding means that the processor works with a description of the schema. The processor builds a dictionary of entities and can recognize and read any entity that is described in the schema. To change the behavior and the scope of processor based on late binding it is enough to change the description of the schema. However, this binding has some disadvantages (for example low speed of reading process).
- In case of early binding, the structure of the classes is created beforehand with the help of a specific automatic tool or manually. If the processor finds an entity that is not found in this schema, it will simply be ignored. The processor calls constructors of appropriate classes and their read methods. To add a new type in the scope of the processor it is necessary to create a class corresponding to the new entity.

The STEP processor is based on early binding principles. It means that specific classes for each EXPRESS type have been created with the help of an automatic tool from the EXPRESS schema. There are two classes for each EXPRESS type. The first class (named the representing class) represents the STEP entity in memory. The second one (RW-class) is intended to perform the initialization of the representing class and to output data to an intermediate structure to be written in a STEP file.

Complex entities
EXPRESS schema allows multiple inheritance. Entities that are built on the basis of multiple inheritance are called complex entities. EXPRESS enables any type of complex entities that can be inherited from any EXPRESS type. In the manner of early binding it is not possible to create a C++ class for any possible complex type. Thus, only widespread complex entities have corresponding representing classes and RW-classes that are created manually beforehand.
Physical file reading

Physical file reading consists of the following steps: 1. Loading a STEP file and syntactic analysis of its contents 2. Mapping STEP entities to the array of strings 3. Creating empty OCCT objects representing STEP entities 4. Initializing OCCT objects 5. Building a references graph

Loading a STEP file and syntactic analysis of its contents

In the first phase, a STEP file is syntactically checked and loaded in memory as a sequence of strings.

Syntactic check is performed on the basis of rules defined in step.lex and step.yacc files. Files step.lex and step.yacc are located in the StepFile noCdlPack development unit. These files describe text encoding of STEP data structure (for additional information see ISO 10303 Part 21). The step.lex file describes the lexical structure of the STEP file. It describes identifiers, numbers, delimiters, etc. The step.yacc file describes the syntactic structure of the file, such as entities, parameters, and headers.

These files have been created only once and need to be updated only when norm ISO 10303-21 is changed.

Mapping STEP entities to arrays of strings

For each entity specified by its rank number the arrays storing its identifier, STEP type and parameters are filled.

Creating empty Open CASCADE Technology objects that represent STEP entities

For each STEP entity an empty OCCT object representing this entity is created. A map of correspondence between entity rank and OCCT object is created and filled out. If a STEP entity is not recognized by the STEP processor then the StepData_UndefinedEntity object is created.

Initializing Open CASCADE Technology objects
Each OCCT object (including StepData_UndefinedEntity) is initialized by its parameters with the help of the appropriate RW-class. If an entity has another entity as its parameter, the object that represents the latter entity will be initialized immediately. All initialized objects are put into a special map to avoid repeated initialization.

Building a graph

The final phase is building a graph of references between entities. For each entity its RW-class is used to find entities referenced by this entity. Back references are built on the basis of direct references. In addition to explicit references defined in the STEP entities some additional (implicit) references are created for entities representing assembly structures (links from assemblies to their components).
How to add a new entity in scope of the STEP processor

If it is necessary to read and translate a new entity by the STEP processor the Reader and Actor scope should be enhanced. Note that some actions to be made for adding a new type are different for simple and complex types. The following steps should be taken:

- Create a class representing a new entity. This can be `Stepxxx_NewEntity` class where `xxx` can be one of the following:
  - Basic
  - Geom
  - Shape
  - Visual
  - Repr
  - AP214
  - AP203
  - AP242

Each field of a STEP entity should be represented by a corresponding field of this class. The class should have methods for initializing, setting and obtaining fields and it should also have the default constructor.

- Create the `RWStepxxx_RWNewEntity` class with a default constructor and methods `ReadStep()`, `WriteStep()` and if the entity references other entities, then method `Share()`.
- Update file `StepAP214_Protocol.cxx`. In the constructor `StepAP214_Protocol::StepAP214_Protocol()` add the new type to the map of registered types and associate the unique integer identifier with this type.
- Update file `RWStepAP214_ReadWriteModule.cxx`. The changes should be the following:
  - For simple types:
    - Add a static object of class `TCollection_AsciiString` with name `Reco_NewEntity` and initialize it with a string containing the STEP type.
    - In constructor `WStepAP214_ReadWriteModule::RWStepAP214_WriteWrite()` add this object onto the list with the unique integer identifier
of the new entity type.

- In function `RWStepAP214_ReadWriteModule::StepType()` add a new C++ case operator for this identifier.
  - For complex types:
    - In the method `RWStepAP214_ReadWriteModule::CaseStep()` add a code for recognition the new entity type returning its unique integer identifier.
    - In the method `RWStepAP214_ReadWriteModule::IsComplex()` return True for this type.
    - In the method `RWStepAP214_ReadWriteModule::ComplexType()` fill the list of subtypes composing this complex type.
  - For both simple and complex types:
    - In function `RWStepAP214_ReadWriteModule::ReadStep()` add a new C++ case operator for the new identifier and call the `RWStepxxx_RWNewEntity` class, method `ReadStep` to initialize the new class.

- Update file `RWStepAP214_GeneralModule.cxx`. Add new C++ case operators to functions `NewVoid()` and `FillSharedCase()`, and in the method `CategoryNumber()` add a line defining a category of the new type.
- Enhance the `STEPControl_ActorRead` class (methods `Recognize()` and `Transfer()`), or class(es) translating some entities, to translate the new entity into an OCCT shape.
Physical file writing

Physical file writing consists of the following steps:

1. Building a references graph. Physical writing starts when STEP model, which was either loaded from a STEP file or created from OCCT shape with the help of translator, is available together with corresponding graph of references. During this step the graph of references can be recomputed.

2. Transferring data from a model to a sequence of strings. For each representing entity from the model a corresponding RW-class is called. RW-class writes data that is contained in the representing class into an intermediate data structure. The mentioned structure is a sequence of strings in memory.

3. Writing the sequence of strings into the file. The sequence of strings is written into the file. This is the last phase of physical STEP writing.
How to add a new entity to write in the STEP file.

If it is necessary to write and translate an OCCT shape into a new entity by the STEP processor the Writer and Actor scope should be enhanced.

For a description of steps, which should be taken for adding a new entity type to the STEP processor, see Physical file reading. Then, enhance the `STEPControl_ActorWrite` class i.e. methods `Recognize()` and `Transfer()`, or other classes from `TopoDSToStep`, to translate the OCCT shape into a new STEP entity.
Using DRAW

DRAW STEP Commands Overview

TKXSDRAW toolkit provides commands for testing XSTEP interfaces interactively in the DRAW environment. It provides an additional set of DRAW commands specific for data exchange tasks, which allows loading and writing data files and an analysis of the resulting data structures and shapes.

This section is divided into five parts. Two of them deal with reading and writing a STEP file and are specific for the STEP processor. The first and the forth parts describe some general tools for setting parameters and analyzing the data. Most of them are independent of the norm being tested. Additionally, a table of mentioned DRAW commands is provided.

In the description of commands, square brackets ([]) are used to indicate optional parameters. Parameters given in the angle brackets (<>), and sharps (#) are to be substituted by an appropriate value. When several exclusive variants are possible, a vertical dash (|) is used.
### Setting the interface parameters

A set of parameters for importing and exporting STEP data is defined in the XSTEP resource file. In XSDRAW, these parameters can be viewed or changed using the command

```plaintext
Draw:> param [<parameter_name> [value]]
```

Command `param` with no arguments gives a list of all parameters with their values. When the argument `parameter_name` is specified, information about this parameter is printed (current value and short description).

The third argument is used to set a new value of the given parameter. The result of the setting is printed immediately.

During all interface operations, the protocol of the process (fail and warning messages, mapping of loaded entities into OCCT shapes etc.) can be output to the trace file. Two parameters are defined in the DRAW session: trace level (integer value from 0 to 9, default is 0), and trace file (default is standard output).

Command `xtrace` is intended to view and change these parameters:

- `Draw:> xtrace` – prints current settings (e.g.: `Level=1 - Standard Output`);
- `Draw:> xtrace #` – sets trace level to the value #;
- `Draw:> xtrace tracefile.log` – sets the trace file as `tracefile.log`;
- `Draw:> xtrace.` – directs all messages to the standard output.
Reading a STEP file

For a description of parameters used in reading a STEP file refer to Setting the translation parameters section.

For reading a STEP file, the following parameters are defined (see above, the command *param*):

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision for input entities</td>
<td>read.precision.mode</td>
<td>0 or 1</td>
<td>If 0 (File), precision of the input STEP file will be used for the loaded shapes; If 1 (Session), the following parameter will be used as the precision value.</td>
</tr>
<tr>
<td></td>
<td>read.precision.val</td>
<td>real</td>
<td>Value of precision (used if the previous parameter is 1)</td>
</tr>
<tr>
<td>Surface curves</td>
<td>read.surfacecurve.mode</td>
<td>0 or 3</td>
<td>Defines a preferable way of representing surface curves (2d or 3d representation). If 0, no preference.</td>
</tr>
<tr>
<td>Maximal tolerance</td>
<td>read.maxprecision.mode</td>
<td>0 or 1</td>
<td>If 1, maximum tolerance is used as a rigid limit If 0, maximum tolerance is used as a limit but can be exceeded by some algorithms.</td>
</tr>
</tbody>
</table>
It is possible either only to load a STEP file into memory (i.e. fill the InterfaceModel with data from the file), or to read it (i.e. load and convert all entities to OCCT shapes). Loading is done by the command

Draw:> xload <file_name>

Once the file is loaded, it is possible to investigate the structure of the loaded data. To find out how you do it, look in the beginning of the analysis subsection. Reading a STEP file is done by the command

Draw:> stepread <file_name> <result_shape_name> [selection]

Here a dot can be used instead of a filename if the file is already loaded by xload or stepread. The optional selection (see below for a description of selections) specifies a set of entities to be translated. If an asterisk `*' is given, all transferable roots are translated. If a selection is not given, the user is prompted to define a scope of transfer interactively:

<table>
<thead>
<tr>
<th>N</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>End</td>
<td>Finish transfer and exit stepread</td>
</tr>
<tr>
<td>1</td>
<td>root with rank 1</td>
<td>Transfer first root</td>
</tr>
<tr>
<td>2</td>
<td>root by its rank</td>
<td>Transfer root specified by its rank</td>
</tr>
<tr>
<td>3</td>
<td>One entity</td>
<td>Transfer entity with a number provided by the user</td>
</tr>
<tr>
<td>4</td>
<td>Selection</td>
<td>Transfer only entities contained in selection</td>
</tr>
</tbody>
</table>

- root is an entity in the STEP file which is not referenced by another entities

Second parameter of the stepread command defines the name of the loaded shape.

During the STEP translation, a map of correspondence between STEP entities and OCCT shapes is created.

To get information on the result of translation of a given STEP entity use the command
Draw: tpent #*.

To create an OCCT shape, corresponding to a STEP entity, use the command

Draw: tpdraw #*.

To get the number of a STEP entity, corresponding to an OCCT shape, use the command

Draw: fromshape <shape_name>.

To clear the map of correspondences between STEP entities and OCCT shapes use the command

Draw: tpclear.
Analyzing the transferred data

The procedure of analysis of data import can be divided into two stages:

1. to check the file contents,
2. to estimate the translation results (conversion and validated ratios).

Checking file contents

General statistics on the loaded data can be obtained by using the command

```
Draw: data <symbol>
```

Information printed by this command depends on the symbol specified:

- $g$ – Prints the information contained in the header of the file;
- $c$ or $f$ – Prints messages generated during the loading of the STEP file (when the procedure of the integrity of the loaded data check is performed) and the resulting statistics ($f$ works only with fails while $c$ with both fail and warning messages);
- $t$ – The same as $c$ or $f$, with a list of failed or warned entities;
- $m$ or $l$ – The same as $t$ but also prints a status for each entity;
- $e$ – Lists all entities of the model with their numbers, types, validity status etc;
- $R$ – The same as $e$ but lists only root entities.

There is a set of special objects, which can be used to operate with a loaded model. They can be of the following types:

- Selection Filters – allow selecting subsets of entities of the loaded model;
- Counter – calculates some statistics on the model data.

A list of these objects defined in the current session can be printed in DRAW by command

```
Draw: listitems.
```
Command

**Draw:** > give list <selection_name>

prints a list of a subset of loaded entities defined by the `<selection>` argument:

- *xst-model-all* all entities of the model;
- *xst-model-roots* all roots;
- *xst-pointed* (Interactively) pointed entities (not used in DRAW);
- *xst-transferrable-all* all transferable (recognized) entities;

The command *listtypes* gives a list of entity types, which were encountered in the last loaded file (with a number of STEP entities of each type).

The list cannot be shown for all entities but for a subset of them. This subset is defined by an optional selection argument (for the list of possible values for STEP, see the table above).

Two commands are used to calculate statistics on the entities in the model:

**Draw:** > count <counter> [ <selection> ]

**Draw:** > list count <counter> [ <selection> ]

The former only prints a count of entities while the latter also gives a list of them.

The optional selection argument, if specified, defines a subset of entities, which are to be taken into account. The first argument should be one of the currently defined counters:

- *xst-types* – calculates how many entities of each OCCT type exist
- *step214-types* – calculates how many entities of each STEP type exist

Entities in the STEP file are numbered in the succeeding order. An entity can be identified either by its number or by its label. Label is the letter # followed by the rank.
• **Draw:** `elab #` outputs a label for an entity with a known number.
• **Draw:** `enum #` prints a number for the entity with a given label.
• **Draw:** `entity # <level_of_information>` outputs the contents of a STEP entity.
• **Draw:** `estat #` outputs the list of entities referenced by a given entity and the list of entities referencing to it.
• **Draw:** `dumpassembly` prints a STEP assembly as a tree.

Information about product names, `next_assembly_usage_occurrence`, `shape_definition_representation`, `context_dependent_shape_representation` or `mapped_item entities` that are involved into the assembly structure will be printed.

**Estimating the results of reading STEP**

All the following commands are available only after data is converted into OCCT shapes (i.e. after command `214read`).

Command **Draw:** `tpstat [*|?]<symbol> [<selection>]` is provided to get all statistics on the last transfer, including a list of transferred entities with mapping from STEP to OCCT types, as well as fail and warning messages. The parameter `<symbol>` defines what information will be printed:

- `g` – General statistics (a list of results and messages)
- `c` – Count of all warning and fail messages
- `C` – List of all warning and fail messages
- `f` – Count of all fail messages
- `F` – List of all fail messages
- `n` – List of all transferred roots
- `s` – The same, with types of source entity and the type of result
- `b` – The same, with messages
- `t` – Count of roots for geometrical types
- `r` – Count of roots for topological types
- `l` – The same, with the type of the source entity

The sign `*` before parameters `n, s, b, t, r` makes it work on all entities (not only on roots).

The sign `?` before `n, s, b, t` limits the scope of information to invalid entities.
Optional argument <selection> can limit the action of the command to the selection, not to all entities.

To get help, run this command without arguments.

The command Draw: tpstat *1 gives statistics on the result of translation of different types of entities (taking check messages into account) and calculates summary translation ratios.

To get information on OCCT shape contents use command Draw: statshape <shape_name>. It outputs the number of each kind of shapes (vertex, edge, wire, etc.) in the shape and some geometrical data (number of C0 surfaces, curves, indirect surfaces, etc.).

The number of faces is returned as a number of references. To obtain the number of single instances, the standard command (from TTOPOLOGY executable) nbshapes can be used.

To analyze the internal validity of the shape, use command Draw: checkbrep <shape_name> <expurged_shape_name>. It checks shape geometry and topology for different cases of inconsistency, like self-intersecting wires or wrong orientation of trimming contours. If an error is found, it copies bad parts of the shape with the names expurged_subshape_name _# and generates an appropriate message. If possible this command also tries to find STEP entities the OCCT shape was produced from.

<expurged_shape_name> will contain the original shape without invalid subshapes. To get information on tolerances of the shape use command Draw: tolerance <shape_name> [<min> [<max>] [<symbol>]]. It outputs maximum, average and minimum values of tolerances for each kind of subshapes having tolerances and for the whole shape in general.

When specifying min and max arguments this command saves shapes with tolerances in the range [min, max] with names shape_name_... and gives their total number.

<Symbol> is used for specifying the kind of sub-shapes to analyze:

- v – for vertices,
- e – for edges,
• $f$ – for faces,
• $c$ – for shells and faces.
Writing a STEP file

For writing shapes to a STEP file, the following parameters are defined (see above, the command *param*):

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncertainty for resulting</td>
<td>Write.precision.mode</td>
<td>-1, 0, 1 or 2</td>
<td>If -1 the uncertainty value is set to the minimal tolerance of CASCADE subshapes. If 0 the uncertainty value is set to the average tolerance of CASCADE subshapes. If 1 the uncertainty value is set to the maximal tolerance of CASCADE subshapes. If 2 the uncertainty value is set to write.precision.val</td>
</tr>
<tr>
<td>Value of uncertainty</td>
<td>Write.precision.val</td>
<td>real</td>
<td>Value of uncertainty (used if previous parameter is 2).</td>
</tr>
</tbody>
</table>

Several shapes can be written in one file. To start writing a new file, enter command `Draw:> newmodel`. Actually, command `newmodel` will clear the `InterfaceModel` to empty it, and the next command will convert the specified shape to STEP entities and add them to the `InterfaceModel`:

`Draw:> stepwrite <mode> <shape_name> [<file_name>]`

The following modes are available:

- a – "as is" – the mode is selected automatically depending on the
type & geometry of the shape;
- \( m \) – manifold\_solid\_brep or brep\_with_voids
- \( f \) – faceted\_brep
- \( w \) – geometric\_curve\_set
- \( s \) – shell\_based\_surface\_model

After a successful translation, if \textit{file\_name} parameter is not specified, the procedure asks you whether to write a STEP model in the file or not:

```
execution status : 1
Mode (0 end, 1 file) :
```

It is necessary to call command \textit{newmodel} to perform a new translation of the next OCCT shape.
Reading from and writing to STEP

The *STEPCAFControl* package (TKXDESTEP toolkit) provides tools to read and write STEP files (see XDE User's Guide).

In addition to the translation of shapes implemented in basic translator, it provides the following:

- STEP assemblies, read as OCCT compounds by basic translator, are translated to XDE assemblies;
- Names of products are translated and assigned to assembly components and instances in XDE;
- STEP external references are recognized and translated (if external documents are STEP files);
- Colors, layers, materials and validation properties assigned to parts or subparts are translated;
- STEP Geometric Dimensions and Tolerances are translated;
- STEP Saved Views are translated.
Reading from STEP

Load a STEP file

Before performing any other operation, you must load a STEP file with:

```cpp
STEPCAFControl_Reader reader(XSDRAW::Session(), Standard_False);
IFSelect_ReturnStatus stat = reader.ReadFile("filename.stp");
```

Loading the file only memorizes the data, it does not translate it.

Check the loaded STEP file

This step is not obligatory. See a description of this step in section Checking the STEP file.

Set parameters for translation to XDE

See a description of this step in section Setting the translation parameters.

In addition, the following parameters can be set for XDE translation of attributes:

- Parameter for transferring colors:
  ```cpp
  reader.SetColorMode(mode);
  // mode can be Standard_True or Standard_False
  ```

- Parameter for transferring names:
  ```cpp
  reader.SetNameMode(mode);
  // mode can be Standard_True or Standard_False
  ```

Translate a STEP file to XDE

The following function performs a translation of the whole document:
Standard_Boolean ok = reader.Transfer(doc);

where doc is a variable which contains a handle to the output document and should have a type Handle(TDocStd_Document).
Attributes read from STEP

Colors

Colors are implemented in accordance with Recommended practices for model styling and organization sections 4 and 5.

The following attributes are imported from STEP file:

- colors linked to assemblies, solids, shells, faces/surfaces, wireframes, edges/curves and vertices/points;
- information about invisibility.

The following attributes are mentioned in the Recommended Practices, but not handled by OCCT:

- styling different sides of surfaces with different colors;
- transparency and reflectance for surfaces;
- curve styles;
- point markers.

Layers

Layers are implemented in accordance with Recommended practices for model styling and organization section 6. All layers are imported, but invisibility styles are skipped.

Materials

Materials are implemented in accordance with Recommended practices for material identification and density section 4. OCCT translator processes materials attached to solids in shape representations. The name, description and density (name and value) are imported for each material.

Validation properties

Validation properties are implemented in accordance with Recommended
practices for geometric and assembly validation properties section 4 for AP214. OCCT processes several types of geometric validation properties for solids, shells and geometric sets:

- area;
- volume;
- centroid.

**Geometric dimensions and tolerances**

General types of STEP entities imported by OCCT are listed in the table below:

<table>
<thead>
<tr>
<th>STEP entity</th>
<th>OCCT attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensional_Size</td>
<td>XCAFDoc_Dimension</td>
</tr>
<tr>
<td>Dimensional_Location</td>
<td>XCAFDoc_Dimension</td>
</tr>
<tr>
<td>Dimensional_Size_With_Path</td>
<td>XCAFDoc_Dimension</td>
</tr>
<tr>
<td>Dimensional_Location_With_Path</td>
<td>XCAFDoc_Dimension</td>
</tr>
<tr>
<td>Angular_Size_Wrap_With_Path</td>
<td>XCAFDoc_Dimension</td>
</tr>
<tr>
<td>Angular_Location</td>
<td>XCAFDoc_Dimension</td>
</tr>
<tr>
<td>Geometric_Tolerance and subtypes</td>
<td>XCAFDoc_GeometricTolerance</td>
</tr>
<tr>
<td>Datum</td>
<td>XCAFDoc_Datum</td>
</tr>
<tr>
<td>Datum_Feature</td>
<td>XCAFDoc_Datum</td>
</tr>
<tr>
<td>Datum_Target</td>
<td>XCAFDoc_Datum</td>
</tr>
</tbody>
</table>

Processing of GD&T is realized in accordance with Recommended practices for the Representation and Presentation of Product Manufacturing for AP242. The general restriction is that OCCT STEP Reader imports GD&T assigned only to shapes (faces, edges, vertices, etc) or to shape groups from general shape model i.e. any constructive geometries are not translated as referenced shapes.

**Dimensions**

Dimensions are implemented according to section 5 of the latter document. Additionally to the reference shapes, the Reader imports from STEP file some auxiliary geometry for dimensional line building:
connection points and line orientation, if exist.

The following values and modifiers described in sections 5.2 and 5.3 can be imported from STEP file:

- qualifiers (minimum, maximum and average);
- plus/minus bounds;
- value range;
- class of tolerance;
- text notes, attached to dimension value;
- dimension modifiers type 2 (Table 8);
- number of decimal places.

**Datums**

Datums are implemented in accordance with sections 6.5 and 6.6.1-6.6.2. Each datum can have one or several datum features (shapes from the model, to which the datum is linked) and datum targets (auxiliary geometry: point, line, rectangle, circle or area).

**Tolerances**

Tolerances are implemented in accordance with sections 6.7-6.9 with several restrictions.

Types of imported tolerances:

- simple tolerances (see Table 10);
- tolerance with modifiers (section 6.9.3);
- tolerance with maximum value (section 6.9.5);
- tolerance with datums (section 6.9.7 (simple datums and datum with modifiers) and 6.9.8 (common datums));
- superposition of the mentioned types.

Not all tolerance zones can be imported by OCCT STEP Reader, only the Tolerance Zones with associated symbols from Table 11, Projected tolerance zone (section 6.9.2.2) and Runout zone definition.

**Presentations**
Each semantic representation of GD&T (Dimension, Tolerance, Datum Feature or Datum Target) can have a presentation; its processing by OCCT is implemented in accordance with sections 7.3, 8 and 9.1-9.2. Presentations have several types:

- **Graphic Presentation** (polylines or tessellated wireframes) - partially implemented in OCCT;
- **Minimal Presentation** (position and orientation) - implemented in OCCT as a part of Graphic presentation;
- **Character-based Presentation** (3D Text with information about fonts, curve styles etc.) - not handled by OCCT.

Note, that separate Minimal presentation and Character-based Presentation are not described in any Recommended Practices, so there is no agreement about how such information should be saved in STEP file.

OCCT STEP Reader imports only Annotation Planes, outline/stroked Polylines and Tessellated wireframes, i.e. all styling information (color, curve style, etc.) and filled characters are missed.

OCCT STEP Reader also handles Annotations, linked directly to shapes (section 9.3.1), processing of these presentations is subject to the same restrictions as the processing of presentations, linked to GD&T semantic.

**Geometric dimensions and tolerances AP214**

Simple types of GD&T (Dimensions, Tolerances and Datums without presentations or any types of modifiers) are also handled in AP214. However, according to the Recommended Practices for the Representation and Presentation of Product Manufacturing, this implementation is obsolete.

**Saved views**

Saved views are implemented in accordance with Recommended practices for the Representation and Presentation of Product Manufacturing section 9.4.1-9.4.4. For each Saved View OCCT STEP Reader will retrieve the following attributes:
- set of displayed shape representations;
- set of displayed PMI presentations;
- projection point;
- view direction;
- up direction of view window;
- horizontal size of view window;
- vertical size of view window;
- zoom factor;
- clipping planes (single plane of combination of planes);
- front and back plane clipping.
Writing to STEP

The translation from XDE to STEP can be initialized as follows:

```cpp
STEPCAFControl_Writer
aWriter(XSDRAW::Session(),Standard_False);
```

Set parameters for translation from XDE to STEP

The following parameters can be set for a translation of attributes to STEP:

- For transferring colors:
  ```cpp
  aWriter.SetColorMode(mode);
  // mode can be Standard_True or Standard_False
  ```

- For transferring names:
  ```cpp
  aWriter.SetNameMode(mode);
  // mode can be Standard_True or Standard_False
  ```

Translate an XDE document to STEP

You can perform the translation of document by calling the function:

```cpp
IFSelect_ReturnStatus aRetSt = aWriter.Transfer(doc);
```

where `doc` is a variable, which contains a handle to the input document for transferring and should have a type `Handle(TDocStd_Document)`.

Write a STEP file

Write a STEP file with:

```cpp
IFSelect_ReturnStatus statw =
  aWriter.writeFile("filename.stp");
```

or
IFSelect_ReturnStatus statw = writer.WriteFile (S);

where S is OStream.
Attributes written to STEP

Colors

The following attributes are exported to STEP file:

- colors linked to assemblies, solids, shells, faces/surfaces, wireframes, edges/curves;
- information about visibility.

Restrictions:

- colors and visibility information for points is not exported by default, it is necessary to use `write.step.vertex.mode` parameter;
- all colors are always applied to both sides of surfaces;
- all curves are exported with 'continuous' curve style.

Layers

All layers are exported, but invisibility styles can be connected only to shapes.

Materials

For solids with materials, the material is exported to STEP file (name, description and density (name and value)).

Validation properties

Geometric validation properties, such as volume, area and centroid, which are attached to shape, are exported to STEP file.

Geometric dimensions and tolerances

All entities, which can be imported from STEP, can be exported too. Please see the same item in section Reading from STEP to find more information.
Note: OCCT use AP214 by default, so for GD&T exporting AP242 should be set manually:

```cpp
Interface_Static::SetCVal("write.step.schema", "AP242DIS");
```

or

```cpp
Interface_Static::SetIVal("write.step.schema", 5));
```

### Saved views

Saved Views are not exported by OCCT.
Level Shapes

- Setting a given Shape at a given Label
- Getting a Shape from a Label
- Getting a Label from a Shape
- Other Queries on a Label
- Instances and References for Components

- Editing Shapes
- Management of Sub-Shapes
- Properties
  - Name
  - Centroid
  - Area
  - Volume
- Colors and Layers
  - Initialization
  - Adding a Color
  - Queries on Colors
  - Editing Colors
- Custom notes
  - Initialization
  - Creating Notes
  - Editing a Note
  - Adding Notes
  - Finding Notes
  - Removing Notes
  - Deleting Notes
- Reading and Writing STEP or IGES
  - Reading a STEP file
- Writing a STEP file
- Reading an IGES File
- Writing an IGES File
- Using an XDE Document
- XDE Data inside an Application Document
Introduction

This manual explains how to use the Extended Data Exchange (XDE). It provides basic documentation on setting up and using XDE. For advanced information on XDE and its applications, see our E-learning & Training offerings.

The Extended Data Exchange (XDE) module allows extending the scope of exchange by translating additional data attached to geometric BREP data, thereby improving the interoperability with external software.

Data types such as colors, layers, assembly descriptions and validation properties (i.e. center of gravity, etc.) are supported. These data are stored together with shapes in an XCAF document. It is also possible to add a new types of data taking the existing tools as prototypes.

Finally, the XDE provides reader and writer tools for reading and writing the data supported by XCAF to and from IGES and STEP files.

Shape imported using XDE
The XDE component requires *Shape Healing* toolkit for operation.
Basic terms

For better understanding of XDE, certain key terms are defined:

- **Shape** – a standalone shape, which does not belong to the assembly structure.
- **Instance** – a replication of another shape with a location that can be the same location or a different one.
- **Assembly** – a construction that is either a root or a sub-assembly.
XDE Organization

The basis of XDE, called XCAF, is a framework based on OCAF (Open CASCADE Technology Application Framework) and is intended to be used with assemblies and with various kinds of attached data (attributes). Attributes can be Individual attributes for a shape, specifying some characteristics of a shape, or they can be Grouping attributes, specifying that a shape belongs to a given group whose definition is specified apart from the shapes.

XDE works in an OCAF document with a specific organization defined in a dedicated XCAF module. This organization is used by various functions of XDE to exchange standardized data other than shapes and geometry.

The Assembly Structure and attributes assigned to shapes are stored in the OCAF tree. It is possible to obtain TopoDS representation for each level of the assembly in the form of TopoDS_Compound or TopoDS_Shape using the API.

Basic elements used by XDE are introduced in the XCAF sub-module by the package XCAFDoc. These elements consist in descriptions of commonly used data structures (apart from the shapes themselves) in normalized data exchanges. They are not attached to specific applications and do not bring specific semantics, but are structured according to the use and needs of data exchanges. The Document used by XDE usually starts as a TDocStd_Document.
Assemblies

XDE supports assemblies by separating shape definitions and their locations. Shapes are simple OCAF objects without a location definition. An assembly consists of several components. Each of these components references one and the same specified shape with different locations. All this provides an increased flexibility in working on multi-level assemblies.

For example, a mechanical assembly can be defined as follows:
XDE defines the specific organization of the assembly content. Shapes are stored on sub-labels of label 0:1:1. There can be one or more roots (called free shapes) whether they are true trees or simple shapes. A shape can be considered to be an Assembly (such as AS1 under 0:1:1:1 in Figure1) if it is defined with Components (sub-shapes, located or not).

*XCAFDoc_ShapeTool* is a tool that allows managing the Shape section of the XCAF document. This tool is implemented as an attribute and located at the root label of the shape section.
Validation Properties

Validation properties are geometric characteristics of Shapes (volume, centroid, surface area) written to STEP files by the sending system. These characteristics are read by the receiving system to validate the quality of the translation. This is done by comparing the values computed by the original system with the same values computed by the receiving system on the resulting model.

Advanced Data Exchange supports both reading and writing of validation properties, and provides a tool to check them.

<table>
<thead>
<tr>
<th>Area</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAINBODY_1</td>
<td>(27.041399390344537)</td>
</tr>
<tr>
<td>HEAD_1</td>
<td>121.00206954580614</td>
</tr>
<tr>
<td>TAIL_1</td>
<td>19.168</td>
</tr>
<tr>
<td>FOOT_1</td>
<td>4.3648e+06</td>
</tr>
<tr>
<td>FOOT_2</td>
<td>168003</td>
</tr>
<tr>
<td>MAINBODY</td>
<td>23.887726172662763</td>
</tr>
<tr>
<td>MAIN BODY BACK_1</td>
<td>3.76129e+06</td>
</tr>
<tr>
<td>MAIN BODY BACK_1</td>
<td>-42.5</td>
</tr>
</tbody>
</table>

Validation Property Descriptions

Check logs contain deviations of computed values from the values stored in a STEP file. A typical example appears as follows:
<table>
<thead>
<tr>
<th>Label</th>
<th>defect</th>
<th>defect</th>
<th>dX</th>
<th>dY</th>
<th>DZ</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1:1:1</td>
<td>312.6 (0%)</td>
<td>-181.7 (0%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>&quot;S1&quot;</td>
</tr>
<tr>
<td>0:1:1:2</td>
<td>-4.6 (0%)</td>
<td>-191.2 (0%)</td>
<td>-0.0</td>
<td>0.0</td>
<td>-0.0</td>
<td>&quot;MAINBODY&quot;</td>
</tr>
<tr>
<td>0:1:1:3</td>
<td>-2.3 (0%)</td>
<td>-52.5 (0%)</td>
<td>-0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>&quot;MAIN_BODY_BACK&quot;</td>
</tr>
<tr>
<td>0:1:1:4</td>
<td>-2.3 (0%)</td>
<td>-51.6 (0%)</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.0</td>
<td>&quot;MAIN_BODY_FRONT&quot;</td>
</tr>
<tr>
<td>0:1:1:5</td>
<td>2.0 (0%)</td>
<td>10.0 (0%)</td>
<td>-0.0</td>
<td>0.0</td>
<td>-0.0</td>
<td>&quot;HEAD&quot;</td>
</tr>
<tr>
<td>0:1:1:6</td>
<td>0.4 (0%)</td>
<td>0.0 (0%)</td>
<td>0.0</td>
<td>-0.0</td>
<td>-0.0</td>
<td>&quot;HEAD_FRONT&quot;</td>
</tr>
<tr>
<td>0:1:1:7</td>
<td>0.4 (0%)</td>
<td>0.0 (0%)</td>
<td>0.0</td>
<td>-0.0</td>
<td>-0.0</td>
<td>&quot;HEAD_BACK&quot;</td>
</tr>
<tr>
<td>0:1:1:8</td>
<td>-320.6 (0%)</td>
<td>10.9 (0%)</td>
<td>-0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>&quot;TAIL&quot;</td>
</tr>
<tr>
<td>0:1:1:9</td>
<td>0.0 (0%)</td>
<td>0.0 (0%)</td>
<td>-0.0</td>
<td>-0.0</td>
<td>0.0</td>
<td>&quot;TAIL_MIDDLE&quot;</td>
</tr>
<tr>
<td>0:1:1:10</td>
<td>-186.2 (0%)</td>
<td>4.8 (0%)</td>
<td>-0.0</td>
<td>0.0</td>
<td>-0.0</td>
<td>&quot;TAIL_TURBINE&quot;</td>
</tr>
<tr>
<td>0:1:1:11</td>
<td>0.3 (0%)</td>
<td>-0.0 (0%)</td>
<td>-0.0</td>
<td>-0.0</td>
<td>0.0</td>
<td>&quot;FOOT&quot;</td>
</tr>
<tr>
<td>0:1:1:12</td>
<td>0.0 (0%)</td>
<td>-0.0 (0%)</td>
<td>0.0</td>
<td>-0.0</td>
<td>-0.0</td>
<td>&quot;FOOT_FRONT&quot;</td>
</tr>
<tr>
<td>0:1:1:13</td>
<td>0.0 (0%)</td>
<td>0.0 (0%)</td>
<td>-0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>&quot;FOOT_BACK&quot;</td>
</tr>
</tbody>
</table>

In our example, it can be seen that no errors were detected for either area, volume or positioning data.
Names

XDE supports reading and writing the names of shapes to and from IGES and STEP file formats. This functionality can be switched off if you do not need this type of data, thereby reducing the size of the document.
Colors and Layers

XDE can read and write colors and layers assigned to shapes or their subparts (down to the level of faces and edges) to and from both IGES and STEP formats. Three types of colors are defined in the enumeration XCAFDoc_ColorType:

- generic color (XCAFDoc_ColorGen)
- surface color (XCAFDoc_ColorSurf)
- curve color (XCAFDoc_ColorCurv)
**Custom notes**

Custom notes is a kind of application specific data attached to assembly items, their attributes and sub-shapes. Basically, there are simple textual comments, binary data and other application specific data. Each note is provided with a timestamp and the user created it.

Notes API provides the following functionality:

- Returns total number of notes and annotated items
- Returns labels for all notes and annotated items
- Creates notes:
  - Comment note from a text string
  - Binary data note from a file or byte array
- Checks if an assembly item is annotated
- Finds a label for the annotated item
- Returns all note labels for the annotated item
- Add a note to item(s):
  - Assembly item
  - Assembly item attribute
  - Assembly item subshape index
- Remove note(s) from an annotated assembly item; orphan note(s) might be deleted optionally (items without linked notes will be deleted automatically)
- Delete note(s) and removes them from annotated items
- Get / delete orphan notes
Working with XDE

Getting started

As explained in the last chapter, XDE uses TDocStd_Documents as a starting point. The general purpose of XDE is:

- Checking if an existing document is fit for XDE;
- Getting an application and initialized document;
- Initializing a document to fit it for XDE;
- Adding, setting and finding data;
- Querying and managing shapes;
- Attaching properties to shapes.

The Document used by XDE usually starts as a TDocStd_Document.

Environment variables

To use XDE you have to set the environment variables properly. Make sure that two important environment variables are set as follows:

- CSF_PluginDefaults points to sources of %CASROOT%/src/XCAFResources ($CASROOT/src/XCAFResources).
- CSF_XCAFDefaults points to sources of %CASROOT%/src/XCAFResources ($CASROOT/src/XCAFResources).

General Check

Before working with shapes, properties, and other types of information, the global organization of an XDE Document can be queried or completed to determine if an existing Document is actually structured for use with XDE.

To find out if an existing TDocStd_Document is suitable for XDE, use:
Handle(TDocStd_Document) doc...
if ( XCAFDoc_DocumentTool::IsXCAFDocument (doc) ) {
  .. yes .. }

If the Document is suitable for XDE, you can perform operations and queries explained in this guide. However, if a Document is not fully structured for XDE, it must be initialized.

**Get an Application or an Initialized Document**

If you want to retrieve an existing application or an existing document (known to be correctly structured for XDE), use:

```cpp
Handle(TDocStd_Document) aDoc;
Handle(XCAFApp_Application) anApp =
    XCAFApp_Application::GetApplication();
anApp->NewDocument( ;MDTV-XCAF; , aDoc);
```
Shapes and Assemblies

Initialize an XDE Document (Shapes)

An XDE Document begins with a `TDocStd_Document`. Assuming you have a `TDocStd_Document` already created, you can ensure that it is correctly structured for XDE by initializing the XDE structure as follows:

```plaintext
Handle(TDocStd_Document) doc...
Handle(XCAFDoc_ShapeTool) myAssembly = XCAFDoc_DocumentTool::ShapeTool(Doc->Main());
TDF_Label aLabel = myAssembly->NewShape()
```

**Note** that the method `XCAFDoc_DocumentTool::ShapeTool` returns the `XCAFDoc_ShapeTool`. The first time this method is used, it creates the `XCAFDoc_ShapeTool`. In our example, a handle is used for the `TDocStd_Document`.

Get a Node considered as an Assembly

To get a node considered as an Assembly from an XDE structure, you can use the Label of the node. Assuming that you have a properly initialized `TDocStd_Document`, use:

```plaintext
Handle(TDocStd_Document) doc...
Handle(XCAFDoc_ShapeTool) myAssembly = XCAFDoc_DocumentTool::ShapeTool(aLabel);
```

In the previous example, you can also get the Main Item of an XDE document, which records the root shape representation (as a Compound if it is an Assembly) by using `ShapeTool(Doc->Main())` instead of `ShapeTool(aLabel)`.

You can then query or edit this Assembly node, the Main Item or another one (`myAssembly` in our examples).

**Note** that for the examples in the rest of this guide, `myAssembly` is always presumed to be accessed this way, so this information will not be
repeated.

**Updating the Assemblies after Filling or Editing**

Some actions in this chapter affect the content of the document, considered as an Assembly. As a result, you will sometimes need to update various representations (including the compounds).

To update the representations, use:

```cpp
myAssembly->UpdateAssemblies();
```

This call performs a top-down update of the Assembly compounds stored in the document.

**Note** that you have to run this method manually to actualize your Assemblies after any low-level modifications on shapes.

**Adding or Setting Top Level Shapes**

Shapes can be added as top-level shapes. Top level means that they can be added to an upper level assembly or added on their own at the highest level as a component or referred by a located instance. Therefore two types of top-level shapes can be added:

- shapes with upper level references
- free shapes (that correspond to roots) without any upper reference

**Note** that several top-level shapes can be added to the same component.

A shape to be added can be defined as a compound (if required), with the following interpretations:

- If the Shape is a compound, according to the user choice, it may or may not be interpreted as representing an Assembly. If it is an Assembly, each of its sub-shapes defines a sub-label.
- If the Shape is not a compound, it is taken as a whole, without breaking it down.

To break down a Compound in the assembly structure, use:
Standard_Boolean makeAssembly;
// True to interpret a Compound as an Assembly,
// False to take it as a whole
aLabel = myAssembly->AddShape(aShape, makeAssembly);

Each node of the assembly therefore refers to its sub-shapes.

Concerning located instances of sub-shapes, the corresponding shapes, (without location) appear at distinct sub-labels. They are referred to by a shape instance, which associates a location.

**Setting a given Shape at a given Label**

A top-level shape can be changed. In this example, no interpretation of compound is performed:

```cpp
Standard_CString LabelString ...;
// identifies the Label (form ;0:i:j...;)
TDF_Label aLabel...;
// A label must be present
myAssembly->SetShape(aLabel, aShape);
```

**Getting a Shape from a Label**

To get a shape from its Label from the top-level, use:

```cpp
TDF_Label aLabel...
// A label must be present
if (aLabel.IsNull()) {
  // no such label : abandon
}
TopoDS_Shape aShape;
aShape = myAssembly->GetShape(aLabel);
if (aShape.IsNull()) {
  // this label is not for a Shape
}
Note that if the label corresponds to an assembly, the result is a compound.

**Getting a Label from a Shape**

To get a Label, which is attached to a Shape from the top-level, use:

```cpp
Standard_Boolean findInstance = Standard_False;
// (this is default value)
alabel = myAssembly->FindShape(aShape
    [,findInstance]);
if (aLabel.IsNull()) {
    // no label found for this shape
}
```

If `findInstance` is True, a search is made for the shape with the same location. If it is False (default value), a search is made among original, non-located shapes.

**Other Queries on a Label**

Various other queries can be made from a Label within the Main Item of XDE:

**Main Shapes**

To determine if a Shape is recorded (or not), use:

```cpp
if ( myAssembly->IsShape(aLabel) ) { .. yes .. }
```

To determine if the shape is top-level, i.e. was added by the `AddShape` method, use:

```cpp
if ( myAssembly->IsTopLevel(aLabel) ) { .. yes .. }
```

To get a list of top-level shapes added by the `AddShape` method, use:

```cpp
TDF_LabelSequence frshapes;
myAssembly->GetShapes(frshapes);
```
To get all free shapes at once if the list above has only one item, use:

```cpp
TopoDS_Shape result = myAssembly->GetShape(frshapes.Value(1));
```

If there is more than one item, you must create and fill a compound, use:

```cpp
TopoDS_Compound C;
BRep_Builder B;
B.MakeCompound(C);
for(Standard_Integer i=1; i=frshapes.Length(); i++) {
   TopoDS_Shape S = myAssembly->GetShape(frshapes.Value(i));
   B.Add(C,S);
}
```

In our example, the result is the compound C. To determine if a shape is a free shape (no reference or super-assembly), use:

```cpp
if ( myAssembly->IsFree(aLabel) ) { .. yes .. }
```

To get a list of Free Shapes (roots), use:

```cpp
TDF_LabelSequence frshapes;
myAssembly->GetFreeShapes(frshapes);
```

To get the shapes, which use a given shape as a component, use:

```cpp
TDF_LabelSequence users;
Standard_Integer nbusers = myAssembly->GetUsers(aLabel,users);
```

The count of users is contained with `nbusers`. It contains 0 if there are no users.

**Assembly and Components**

To determine if a label is attached to the main part or to a sub-part (component), use:
if (myAssembly->IsComponent(aLabel)) { .. yes .. }

To determine whether a label is a node of a (sub-) assembly or a simple shape, use:

if ( myAssembly->IsAssembly(aLabel) ) { .. yes .. }

If the label is a node of a (sub-) assembly, you can get the count of components, use:

Standard_Boolean subchilds = Standard_False;
   //default
Standard_Integer nbc = myAssembly->NbComponents(aLabel [,subchilds]);

If subchilds is True, commands also consider sub-levels. By default, only level one is checked.

To get component Labels themselves, use:

Standard_Boolean subchilds = Standard_False;
   //default
TDF_LabelSequence comps;
Standard_Boolean isassembly = myAssembly->GetComponents(aLabel,comps[,subchilds]);

**Instances and References for Components**

To determine if a label is a simple shape, use:

if ( myAssembly->IsSimpleShape(aLabel) ) { .. yes .. }

To determine if a label is a located reference to another one, use:

if ( myAssembly->IsReference(aLabel) ) { .. yes .. }

If the label is a located reference, you can get the location, use:
TopLoc_Location loc = myAssembly->GetLocation(aLabel);

To get the label of a referenced original shape (also tests if it is a reference), use:

Standard_Boolean isref = myAssembly->GetReferredShape(aLabel, refLabel);

**Note** *isref* returns False if *aLabel* is not for a reference.
**Editing Shapes**

In addition to the previously described *AddShape* and *SetShape*, several shape edits are possible.

To remove a Shape, and all its sub-labels, use:

```plaintext
Standard_Boolean remsh = myAssembly->RemoveShape(aLabel);
// remsh is returned True if done
```

This operation will fail if the shape is neither free nor top level.

To add a Component to the Assembly, from a new shape, use:

```plaintext
Standard_Boolean expand = Standard_False; //default
TDF_Label aLabel = myAssembly->AddComponent(aShape [,expand]);
```

If *expand* is True and *aShape* is a Compound, *aShape* is broken down to produce sub-components, one for each of its sub-shapes.

To add a component to the assembly, from a previously recorded shape (the new component is defined by the label of the reference shape, and its location), use:

```plaintext
TDF_Label refLabel ...; // the label of reference shape
TopLoc_Location loc ...; // the desired location
TDF_Label aLabel = myAssembly->AddComponent (refLabel, loc);
```

To remove a component from the assembly, use:

```plaintext
myAssembly->RemoveComponent (aLabel);
```
Management of Sub-Shapes

In addition to components of a (sub-)assembly, it is possible to have individual identification of some sub-shapes inside any shape. Therefore, you can attach specific attributes such as Colors. Some additional actions can be performed on sub-shapes that are neither top-level, nor components: To add a sub-shape to a given Label, use:

\[
\text{TDF\_Label subLabel} = \text{myAssembly}\rightarrow\text{AddSubShape (aLabel, subShape)};
\]

To find the Label attached to a given sub-shape, use:

\[
\text{TDF\_Label subLabel; // new label to be computed if ( myAssembly}\rightarrow\text{FindSubShape (aLabel, subShape, subLabel)) \{ .. yes .. \}}
\]

If the sub-shape is found (yes), \text{subLabel} is filled by the correct value.

To find the top-level simple shape (not a compound whether free or not), which contains a given sub-shape, use:

\[
\text{TDF\_Label mainLabel} = \text{myAssembly}\rightarrow\text{FindMainShape(subShape)};
\]

\textbf{Note} that there should be only one shape for a valid model. In any case, the search stops on the first one found.

To get the sub-shapes of a shape, which are recorded under a label, use:

\[
\text{TDF\_LabelSequence subs; Standard\_Boolean hassubs} = \text{myAssembly}\rightarrow\text{GetSubShapes(aLabel,subs)};
\]
Properties

Some properties can be attached directly to shapes. These properties are:

- Name (standard definition from OCAF)
- Centroid (for validation of transfer)
- Volume (for validation of transfer)
- Area (for validation of transfer)

Some other properties can also be attached, and are also managed by distinct tools for Colors and Layers. Colors and Layers are managed as an alternative way of organizing data (by providing a way of identifying groups of shapes). Colors are put into a table of colors while shapes refer to this table.

There are two ways of attaching a color to a shape:

- By attaching an item from the table.
- Adding the color directly. When the color is added directly, a search is performed in the table of contents to determine if it contains the requested color. Once this search and initialize operation is done, the first way of attaching a color to a shape is used.

Name

Name is implemented and used as a \( TDataStd\_Name \), which can be attached to any label. Before proceeding, consider that:

- In IGES, every entity can have a name with an optional numeric part called a Subscript Label. For example, \( MYCURVE \) is a name, and \( MYCURVE(60) \) is a name with a Subscript Label.
- In STEP, there are two levels: Part Names and Entity Names:
  - Part Names are attached to ;main shapes; such as parts and assemblies. These Part Names are specifically supported by XDE.
  - Entity Names can be attached to every Geometric Entity. This option is rarely used, as it tends to overload the exploitation of the data structure. Only some specific cases justify using this option: for example, when the sending system can really ensure the stability of an entity name after each STEP writing. If such stability is ensured, you can use this option to send an Identifier for external applications using a database. Note that both IGES...
or STEP files handle names as pure ASCII strings.

These considerations are not specific to XDE. What is specific to data exchange is the way names are attached to entities.

To get the name attached to a label (as a reminder using OCAF), use:

```cpp
Handle(TDataStd_Name) N;
if ( !aLabel.FindAttribute(TDataStd_Name::GetID(), N))
    { // no name is attached
    }
TCollection_ExtendedString name = N->Get();
```

Don't forget to consider Extended String as ASCII, for the exchange file.

To set a name to a label (as a reminder using OCAF), use:

```cpp
TCollection_ExtendedString aName ...
// contains the desired name for this Label (ASCII)
TDataStd_Name::Set (aLabel, aName);
```

**Centroid**

A Centroid is defined by a Point to fix its position. It is handled as a property, item of the class `XCAFDoc_Centroid`, sub-class of `TDF_Attribute`. However, global methods give access to the position itself.

This notion has been introduced in STEP, together with that of Volume, and Area, as defining the Validation Properties: this feature allows exchanging the geometries and some basic attached values, in order to perform a synthetic checking on how they are maintained after reading and converting the exchange file. This exchange depends on reliable exchanges of Geometry and Topology. Otherwise, these values can be considered irrelevant.

A centroid can be determined at any level of an assembly, thereby allowing a check of both individual simple shapes and their combinations including locations.
To get a Centroid attached to a Shape, use:

```cpp
gp_Pnt pos;
Handle(XCAFDoc_Centroid) C;
aLabel.FindAttribute(XCAFDoc_Centroid::GetID(), C);
if (!C.IsNull()) pos = C->Get();
```

To set a Centroid to a Shape, use:

```cpp
gp_Pnt pos (X,Y,Z);
// the position previously computed for the centroid
XCAFDoc_Centroid::Set (aLabel, pos);
```

**Area**

An Area is defined by a Real, it corresponds to the computed Area of a Shape, provided that it contains surfaces. It is handled as a property, item of the class `XCAFDoc_Area`, sub-class of `TDF_Attribute`. This notion has been introduced in STEP but it is usually disregarded for a Solid, as Volume is used instead. In addition, it is attached to simple shapes, not to assemblies.

To get an area attached to a Shape, use:

```cpp
Standard_Real area;
Handle(XCAFDoc_Area) A;
L.FindAttribute(XCAFDoc_Area::GetID(), A);
if (!A.IsNull()) area = A->Get();
```

To set an area value to a Shape, use:

```cpp
Standard_Real area ...;
// value previously computed for the area
XCAFDoc_Area::Set(aLabel, area);
```

**Volume**

A Volume is defined by a Real and corresponds to the computed volume of a Shape, provided that it contains solids. It is handled as a property, an
item of the class \textit{XCAFDoc\_Volume}, sub-class of \textit{TDF\_Attribute}. This notion has been introduced in STEP. It may be attached to simple shapes or their assemblies for computing cumulated volumes and centers of gravity.

To get a Volume attached to a Shape, use:

\begin{verbatim}
Standard_Real volume;
Handle(XCAFDoc_Volume) V;
L.FindAttribute ( XCAFDoc_Volume::GetID(), V );
if ( !V.IsNull() ) volume = V->Get();
\end{verbatim}

To set a volume value to a Shape, use:

\begin{verbatim}
Standard_Real volume ...;
// value previously computed for the volume
XCAFDoc_Volume::Set ( aLabel, volume );
\end{verbatim}
Colors and Layers

XDE can read and write colors and layers assigned to shapes or their subparts (down to level of faces and edges) to and from both IGES and STEP formats.

In an XDE document, colors are managed by the class \textit{XCAFDoc\_ColorTool}. This is done with the same principles as for ShapeTool with Shapes, and with the same capability of having a tool on the Main Label, or on any sub-label. The Property itself is defined as an \textit{XCAFDoc\_Color}, sub-class of \textit{TDF\_Attribute}.

Colors are stored in a child of the starting document label: it is the second level (0.1.2), while Shapes are at the first level. Each color then corresponds to a dedicated label, the property itself is a \textit{Quantity\_Color}, which has a name and value for Red, Green, Blue. A Color may be attached to Surfaces (flat colors) or to Curves (wireframe colors), or to both. A Color may be attached to a sub-shape. In such a case, the sub-
shape (and its own sub-shapes) takes its own Color as a priority.

Layers are handled using the same principles as Colors. In all operations described below you can simply replace Color with Layer when dealing with Layers. Layers are supported by the class XCAFDoc_LayerTool.

The class of the property is XCAFDoc_Layer, sub-class of TDF_Attribute while its definition is a TCollection_ExtendedString. Integers are generally used when dealing with Layers. The general cases are:

- IGES has LevelList as a list of Layer Numbers (not often used)
- STEP identifies a Layer (not by a Number, but by a String), to be more general.

Colors and Shapes are related to by Tree Nodes.

These definitions are common to various exchange formats, at least for STEP and IGES.

**Initialization**

To query, edit, or initialize a Document to handle Colors of XCAF, use:

```cpp
Handle(XCAFDoc_ColorTool) myColors = XCAFDoc_DocumentTool::ColorTool(Doc->Main());
```

This call can be used at any time. The first time it is used, a relevant structure is added to the document. This definition is used for all the following color calls and will not be repeated for these.

**Adding a Color**

There are two ways to add a color. You can:

- add a new Color defined as Quantity_Color and then directly set it to a Shape (anonymous Color)
- define a new Property Color, add it to the list of Colors, and then set it to various shapes. When the Color is added by its value Quantity_Color, it is added only if it has not yet been recorded (same RGB values) in the Document.
To set a Color to a Shape using a label, use:

```plaintext
Quantity_Color Col (red,green,blue);
XCAFDoc_ColorType ctype ..;
// can take one of these values:
// XCAFDoc_ColorGen : all types of geometries
// XCAFDoc_ColorSurf : surfaces only
// XCAFDoc_ColorCurv : curves only
myColors->SetColor (aLabel, Col, ctype);
```

Alternately, the Shape can be designated directly, without using its label, use:

```plaintext
myColors->SetColor (aShape, Col, ctype);
// Creating and Adding a Color, explicitly
Quantity_Color Col (red,green,blue);
TDF_Label ColLabel = myColors->AddColor (Col);
```

**Note** that this Color can then be named, allowing later retrieval by its Name instead of its Value.

To set a Color, identified by its Label and already recorded, to a Shape, use:

```plaintext
XCAFDoc_ColorType ctype ..; // see above
if (myColors->SetColors (aLabel, ColLabel, ctype))
   {.. it is done .. }
```

In this example, `aLabel` can be replaced by `aShape` directly.

**Queries on Colors**

Various queries can be performed on colors. However, only specific queries are included in this section, not general queries using names.

To determine if a Color is attached to a Shape, for a given color type (ctype), use:

```plaintext
if (myColors->IsSet (aLabel, ctype)) {
```
In this example, \textit{aLabel} can be replaced by \textit{aShape} directly.

To get the Color attached to a Shape (for any color type), use:

\begin{verbatim}
Quantity_Color col;
// will receive the recorded value (if there is some)
if ( !myColors->GetColor(aLabel, col) ) {
    // sorry, no color ..
}
\end{verbatim}

Color name can also be queried from \textit{col.StringName} or \textit{col.Name}. In this example, \textit{aLabel} can be replaced by \textit{aShape} directly.

To get the Color attached to a Shape, with a specific color type, use:

\begin{verbatim}
XCAFDoc_ColorType ctype ..;
Quantity_Color col;
// will receive the recorded value (if there is some)
if ( !myColors->GetColor(aLabel, ctype, col) ) {
    // sorry, no color ..
}
\end{verbatim}

To get all the Colors recorded in the Document, use:

\begin{verbatim}
Quantity_Color col; // to receive the values
TDF_LabelSequence ColLabels;
myColors->GetColors(ColLabels);
Standard_Integer i, nbc = ColLabels.Length();
for (i = 1; i = nbc; i ++) {
    aLabel = Labels.Value(i);
    if ( !myColors->GetColor(aLabel, col) ) continue;
    // col receives the color n° i ..
}
\end{verbatim}

To find a Color from its Value, use:
Editing Colors

Besides adding colors, the following attribute edits can be made:

To unset a Color on a Shape, use:

```cpp
XCAFDoc_ColorType ctype ...;
// desired type (XCAFDoc_ColorGen for all )
myColors->UnSetColor (aLabel, ctype);
```

To remove a Color and all the references to it (so that the related shapes will become colorless), use:

```cpp
myColors->RemoveColor(ColLabel);
```
Custom notes

In an XDE document, custom notes are managed by the class `XCAFDoc_NotesTool`. This is done with the same principles as for ShapeTool with Shapes, and with the same capability of having a tool on the Main Label, or on any sub-label. The Property itself is defined as sub-classes of an `XCAFDoc_Note` abstract class, which is a sub-class of `TDF_Attribute` one.

Custom notes are stored in a child of the `XCAFDoc_NotesTool` label: it is at label 0.1.9.1. Each note then corresponds to a dedicated label. A note may be attached to a document item identified by a label, a sub-shape identified by integer index or an attribute identified by GUID. Annotations are stored in a child of the `XCAFDoc_NotesTool` label: it is at label 0.1.9.2. Notes binding is done through `XCAFDoc_GraphNode` attribute.

Structure of notes part of XCAF document

Initialization
To query, edit, or initialize a Document to handle custom notes of XCAF, use:

```cpp
Handle(XCAFDoc_NotesTool) myNotes = XCAFDoc_DocumentTool::NotesTool(Doc->Main());
```

This call can be used at any time. The first time it is used, a relevant structure is added to the document. This definition is used for all the following notes calls and will not be repeated for these.

**Creating Notes**

Before annotating a Document item a note must be created using one of the following methods of `XCAFDoc_NotesTool` class:

- `CreateComment` : creates a note with a textual comment
- `CreateBinData` : creates a note with arbitrary binary data, e.g. contents of a file

Both methods return an instance of `XCAFDoc_Note` class.

```cpp
Handle(XCAFDoc_NotesTool) myNotes = ...
Handle(XCAFDoc_Note) myNote = myNotes-
    >CreateComment("User", "Timestamp", "Hello, World!");
```

This code adds a child label to label 0.1.9.1 with `XCAFDoc_NoteComment` attribute.

**Editing a Note**

An instance of `XCAFDoc_Note` class can be used for note editing. One may change common note data.

```cpp
myNote->Set("New User", "New Timestamp");
```

To change specific data one need to down cast `myNote` handle to the appropriate sub-class:

```cpp
Handle(XCAFDoc_NoteComment) myCommentNote =
```
Adding Notes

Once a note has been created it can be bound to a Document item using the following \texttt{XCAFDoc_NotesTool} methods:

- \texttt{AddNote} : binds a note to a label
- \texttt{AddNoteToAttr} : binds a note to a label's attribute
- \texttt{AddNoteToSubshape} : binds a note to a sub-shape

All methods return a pointer to \texttt{XCAFDoc_AssemblyItemRef} attribute identifying the annotated item.

```cpp
Handle(XCAFDoc_NotesTool) myNotes = ...;
Handle(XCAFDoc_Note) myNote = ...;
TDF_Label theLabel; ...
Handle(XCAFDoc_AssemblyItemRef) myRef = myNotes->AddNote(myNote->Label(), theLabel);
Standard_GUID theAttrGUID; ...
Handle(XCAFDoc_AssemblyItemRef) myRefAttr = myNotes->AddNoteToAttr(myNote->Label(), theAttrGUID);
Standard_Integer theSubshape = 1;
Handle(XCAFDoc_AssemblyItemRef) myRefSubshape = myNotes->AddNoteToSubshape(myNote->Label(), theSubshape);
```

This code adds three child labels to label 0.1.9.2 with \texttt{XCAFDoc_AssemblyItemRef} attribute with \texttt{XCAFDoc_GraphNode} attributes added to this and note labels.

Finding Notes

To find annotation labels under label 0.1.9.2 use the following \texttt{XCAFDoc_NotesTool} methods:
- **FindAnnotatedItem**: returns an annotation label for a label
- **FindAnnotatedItemAttr**: returns an annotation label for a label's attribute
- **FindAnnotatedItemSubshape**: returns an annotation label for a sub-shape

```plaintext
Handle(XCAFDoc_NotesTool) myNotes = ...  
TDF_Label theLabel; ...
TDF_Label myLabel = myNotes->FindAnnotatedItem(theLabel);
Standard_GUID theAttrGUID; ...
TDF_Label myLabelAttr = myNotes->FindAnnotatedItemAttr(theLabel, theAttrGUID);
Standard_Integer theSubshape = 1;
TDF_Label myLabelSubshape = myNotes->FindAnnotatedItemSubshape(theLabel, theSubshape);
```

Null label will be returned if there is no corresponding annotation.

To get all notes of the Document item use the following `XCAFDoc_NotesTool` methods:

- **GetNotes**: outputs a sequence of note labels bound to a label
- **GetAttrNotes**: outputs a sequence of note labels bound to a label's attribute
- **GetAttrSubshape**: outputs a sequence of note labels bound to a sub-shape

All these methods return the number of notes.

```plaintext
Handle(XCAFDoc_NotesTool) myNotes = ...  
TDF_Label theLabel; ...
TDF_LabelSequence theNotes;  
myNotes->GetNotes(theLabel, theNotes);
Standard_GUID theAttrGUID; ...
TDF_LabelSequence theNotesAttr;  
myNotes->GetAttrNotes(theLabel, theAttrGUID, theNotesAttr);
```
Removing Notes

To remove a note use one of the following XCAFDoc_NotesTool methods:

- RemoveNote : unbinds a note from a label
- RemoveAttrNote : unbinds a note from a label's attribute
- RemoveSubshapeNote : unbinds a note from a sub-shape

```
Handle(XCAFDoc_Note) myNote = ...  
TDF_Label theLabel; ...  
myNotes->RemoveNote(myNote->Label(), theLabel);  
Standard_GUID theAttrGUID; ...  
myRefAttr = myNotes->RemoveAttrNote(myNote->Label(), theAttrGUID);  
Standard_Integer theSubshape = 1;  
myNotes->RemoveSubshapeNote(myNote->Label(), theSubshape);  
```

A note will not be deleted automatically. Counterpart methods to remove all notes are available too.

Deleting Notes

To delete note(s) use the following XCAFDoc_NotesTool methods:

- DeleteNote : deletes a single note
- DeleteNotes : deletes a sequence of notes
- DeleteAllNotes : deletes all Document notes
- DeleteOrphanNotes : deletes notes not bound to Document items

All these methods excepting the last one break all links with Document items as well.
Reading and Writing STEP or IGES

Note that saving and restoring the document itself are standard OCAF operations. As the various previously described definitions enter into this frame, they will not be explained any further. The same can be said for Viewing: presentations can be defined from Shapes and Colors.

There are several important points to consider:

- Previously defined Readers and Writers for dealing with Shapes only, whether Standard or Advanced, remain unchanged in their form and in their dependencies. In addition, functions other than mapping are also unchanged.
- XDE provides mapping with data other than Shapes. Names, Colors, Layers, Validation Properties (Centroid, Volume, Area), and Assembly Structure are hierarchic with rigid motion.
- XDE mapping is relevant for use within the Advanced level of Data Exchanges, rather than Standard ones, because a higher level of information is better suited to a higher quality of shapes. In addition, this allows to avoid the multiplicity of combinations between various options. Note that this choice is not one of architecture but of practical usage and packaging.
- Reader and Writer classes for XDE are generally used like those for Shapes. However, their use is adapted to manage a Document rather than a Shape.

The packages to manage this are IGESCAFControl for IGES, and STEPCAFControl for STEP.

Reading a STEP file

To read a STEP file by itself, use:

```c
STEPCAFControl_Reader reader;
IFSelect_ReturnStatus readstat =
    reader.ReadFile(filename);
// The various ways of reading a file are available here too:
```
In addition, the reader provides methods that are applicable to document transfers and for directly querying of the data produced.

**Writing a STEP file**

To write a STEP file by itself, use:

```cpp
STEPControl_StepModelType mode = STEPControl_AsIs;
// Asis is the recommended value, others are available
// Firstly, perform the conversion to STEP entities
STEPCAFControl_Writer writer;
// (the user can work with an already prepared
   WorkSession or create a //new one)
Standard_Boolean scratch = Standard_False;
STEPCAFControl_Writer writer ( WS, scratch );
// Translating document (conversion) to STEP
if ( ! writer.Transfer ( Doc, mode ) ) {
    cout;The document cannot be translated or gives no result;endl;
    // abandon ..
}
```
// Writing the File
IFSelect_ReturnStatus stat = writer.Write(file-name);

**Reading an IGES File**

Use the same procedure as for a STEP file but with IGESCAFControl instead of STEPCAFControl.

**Writing an IGES File**

Use the same procedure as for a STEP file but with IGESCAFControl instead of STEPCAFControl.
Using an XDE Document

There are several ways of exploiting XDE data from an application, you can:

1. Get the data relevant for the application by mapping XDE/Appli, then discard the XDE data once it has been used.
2. Create a reference from the Application Document to the XDE Document, to have its data available as external data.
3. Embed XDE data inside the Application Document (see the following section for details).
4. Directly exploit XDE data such as when using file checkers.

XDE Data inside an Application Document

To have XCAF data elsewhere than under label 0.1, you use the DocLabel of XDE. The method DocLabel from XCAFDoc_DocumentTool determines the relevant Label for XCAF. However, note that the default is 0.1.

In addition, as XDE data is defined and managed in a modular way, you can consider exclusively Assembly Structure, only Colors, and so on.

As XDE provides an extension of the data structure, for relevant data in standardized exchanges, note the following:

- This data structure is fitted for data exchange, rather than for use by the final application.
- The provided definitions are general, for common use and therefore do not bring strongly specific semantics.

As a result, if an application works on Assemblies, on Colors or Layers, on Validation Properties (as defined in STEP), it can rely on all or a part of the XDE definitions, and include them in its own data structure.

In addition, if an application has a data structure far from these notions, it can get data (such as Colors and Names on Shapes) according to its needs, but without having to consider the whole.
Introduction

Purpose of OCAF

Architecture

Overview

Application

Document

Attribute

Reference-key model

The Data Framework

Data Structure

Examples of a Data Structure

Tag

Creating child labels using random delivery of tags

Creation of a child label by user delivery from a tag

Label

Label creation

Creating child labels

Retrieving child labels

Retrieving the father label

Attribute
Retrieving an attribute from a label
Identifying an attribute using a GUID
Attaching an attribute to a label
Testing the attachment to a label
Removing an attribute from a label
Specific attribute creation

Compound documents
Transaction mechanism
Standard Document Services
Overview

The Application
Creating an application
Creating a new document
Retrieving the application to which the document belongs

The Document
Accessing the main label of the framework
Retrieving the document from a label in its framework
Defining storage format
Defining storage format by resource files
Saving a document
Opening the document from a file
Cutting, copying and pasting inside a document

External Links
Copying the document

OCAF Shape Attributes
Overview
Shape attributes in data framework.
Registering shapes and their evolution
Using naming resources
Reading the contents of a named shape attribute
Topological naming
Algorithm history
Loading history in data framework
Selection / recomputation mechanism
Exploring shape evolution
Example of topological naming usage
### Standard Attributes

**Overview**

**Services common to all attributes**
- Accessing GUIDs
- Conventional Interface of Standard Attributes

**The choice between standard and custom attributes**
- Comparison and analysis of approaches
- Conclusion

**Visualization Attributes**

**Overview**

**Services provided**
- Defining an interactive viewer attribute
- Defining a presentation attribute
  - Creating your own driver
  - Using a container for drivers

**Function Services**

- Finding functions, their owners and roots
- Storing and accessing information about function status
- Propagating modifications

**Example of Function Mechanism Usage**

**Introduction**
Step 1: Data Tree
Step 2: Interfaces
  - Creation of the nail
  - Computation
  - Visualization
  - Removal of the nail
Step 3: Functions
  - Example 1: iteration and execution of functions.
  - Example 2: Cylinder function driver

XML Support
  - Document Drivers
  - Attribute Drivers
  - XML Document Structure
  - XML Schema

GLOSSARY

Samples
  - Getting Started
  - Implementation of Attribute Transformation in a HXX file
  - Implementation of Attribute Transformation in a CPP file
  - Implementation of typical actions with standard OCAF attributes.
Introduction

This manual explains how to use the Open CASCADE Application Framework (OCAF). It provides basic documentation on using OCAF. For advanced information on OCAF and its applications, see our E-learning & Training offerings.
Purpose of OCAF

OCAF (the Open CASCADE Application Framework) is an easy-to-use platform for rapidly developing sophisticated domain-specific design applications. A typical application developed using OCAF deals with two or three-dimensional (2D or 3D) geometric modeling in trade-specific Computer Aided Design (CAD) systems, manufacturing or analysis applications, simulation applications or illustration tools.

Developing a design application requires addressing many technical aspects. In particular, given the functional specification of your application, you must at least:

- Design the architecture of the application—definition of the software components and the way they cooperate;
- Define the data model able to support the functionality required — a design application operates on data maintained during the whole end-user working session;
- Structure the software in order to:
  - synchronize the display with the data — commands modifying objects must update the views;
  - support generalized undo-redo commands — this feature has to be taken into account very early in the design process;
- Implement the function for saving the data — if the application has a long life cycle, the compatibility of data between versions of the application has to be addressed;
- Build the application user interface.

Architectural guidance and ready-to-use solutions provided by OCAF offer you the following benefits:

- You can concentrate on the functionality specific for your application;
- The underlying mechanisms required to support the application are already provided;
- The application can be rapidly be prototyped thanks to the coupling the other Open CASCADE Technology modules;
- The final application can be developed by industrializing the prototype — you don't need to restart the development from scratch.
- The Open Source nature of the platform guarantees the long-term
usefulness of your development.

OCAF is much more than just one toolkit among many in the CAS.CADE Object Libraries. Since it can handle any data and algorithms in these libraries – be it modeling algorithms, topology or geometry – OCAF is their logical supplement.

The table below contrasts the design of a modeling application using object libraries alone and using OCAF.

**Table 1: Services provided by OCAF**

<table>
<thead>
<tr>
<th>Development tasks</th>
<th>Comments</th>
<th>Without OCAF</th>
<th>With OCAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation of geometry</td>
<td>Algorithm Calling the modeling libraries</td>
<td>To be created by the user</td>
<td>To be created by the user</td>
</tr>
<tr>
<td>Data organization</td>
<td>Including specific attributes and modeling process</td>
<td>To be created by the user</td>
<td>Simplified</td>
</tr>
<tr>
<td>Saving data in a file</td>
<td>Notion of document</td>
<td>To be created by the user</td>
<td>Provided</td>
</tr>
<tr>
<td>Document-view management</td>
<td></td>
<td>To be created by the user</td>
<td>Provided</td>
</tr>
<tr>
<td>Application infrastructure</td>
<td>New, Open, Close, Save and Save As File menus</td>
<td>To be created by the user</td>
<td>Provided</td>
</tr>
<tr>
<td>Undo-Redo</td>
<td>Robust, multi-level</td>
<td>To be created by the user</td>
<td>Provided</td>
</tr>
<tr>
<td>Application-specific dialog boxes</td>
<td></td>
<td>To be created by the user</td>
<td>To be created by the user</td>
</tr>
</tbody>
</table>

OCAF uses other modules of Open CASCADE Technology — the Shape is implemented with the geometry supported by the Modeling Data
module and the viewer is the one provided with the Visualization module. Modeling functions can be implemented using the Modeling Algorithms module.

The relationship between OCAF and the Open CASCADE Technology (OCCT) Object Libraries can be seen in the image below.

![OCCT Architecture](image)

**OCCT Architecture**

In the image, the OCAF (Open CASCADE Application Framework) is shown with black rectangles and OCCT Object Libraries required by OCAF are shown with white rectangles.

The subsequent chapters of this document explain the concepts and show how to use the services of OCAF.
Architecture Overview

OCAF provides you with an object-oriented Application-Document-Attribute model consisting of C++ class libraries.

The Application-Document-Attribute model

Application

The Application is an abstract class in charge of handling documents during the working session, namely:

- Creating new documents;
- Saving documents and opening them;
- Initializing document views.

Document

The document, implemented by the concrete class Document, is the container for the application data. Documents offer access to the data framework and serve the following purposes:

- Manage the notification of changes
- Update external links
- Manage the saving and restoring of data
- Store the names of software extensions.
- Manage command transactions
- Manage Undo and Redo options.
Each document is saved in a single flat ASCII file defined by its format and extension (a ready-to-use format is provided with OCAF).

Apart from their role as a container of application data, documents can refer to each other; Document A, for example, can refer to a specific label in Document B. This functionality is made possible by means of the reference key.

Attribute

Application data is described by Attributes, which are instances of classes derived from the Attribute abstract class, organized according to the OCAF Data Framework.

The OCAF Data Framework references aggregations of attributes using persistent identifiers in a single hierarchy. A wide range of attributes come with OCAF, including:

- **Standard attributes** allow operating with simple common data in the data framework (for example: integer, real, string, array kinds of data), realize auxiliary functions (for example: tag sources attribute for the children of the label counter), create dependencies (for example: reference, tree node)....;
- **Shape attributes** contain the geometry of the whole model or its elements including reference to the shapes and tracking of shape evolution;
- Other geometric attributes such as Datums (points, axis and plane) and Constraints (tangent-to, at-a-given-distance, from-a-given-angle, concentric, etc.)
- User attributes, that is, attributes typed by the application
- **Visualization attributes** allow placing viewer information to the data framework, visual representation of objects and other auxiliary visual information, which is needed for graphical data representation.
- **Function services** — the purpose of these attributes is to rebuild objects after they have been modified (parameterization of models). While the document manages the notification of changes, a function manages propagation of these changes. The function mechanism provides links between functions and calls to various algorithms.

In addition, application-specific data can be added by defining new attribute classes; naturally, this changes the standard file format. The
only functions that have to be implemented are:

- Copying the attribute
- Converting it from and persistent data storage
Reference-key model

In most existing geometric modeling systems, the data are topology driven. They usually use a boundary representation (BRep), where geometric models are defined by a collection of faces, edges and vertices, to which application data are attached. Examples of data include:

- a color;
- a material;
- information that a particular edge is blended.

When the geometric model is parameterized, that is, when you can change the value of parameters used to build the model (the radius of a blend, the thickness of a rib, etc.), the geometry is highly subject to change. In order to maintain the attachment of application data, the geometry must be distinguished from other data.

In OCAF, the data are reference-key driven. It is a uniform model in which reference-keys are the persistent identification of data. All accessible data, including the geometry, are implemented as attributes attached to reference-keys. The geometry becomes the value of the Shape attribute, just as a number is the value of the Integer and Real attributes and a string that of the Name attribute.

On a single reference-key, many attributes can be aggregated; the application can ask at runtime which attributes are available. For example, to associate a texture to a face in a geometric model, both the face and the texture are attached to the same reference-key.
Topology driven versus reference-key driven approaches

Reference-keys can be created in two ways:

- At programming time, by the application
- At runtime, by the end-user of the application (providing that you include this capability in the application)

As an application developer, you generate reference-keys in order to give semantics to the data. For example, a function building a prism may create three reference-keys: one for the base of the prism, a second for the lateral faces and a third for the top face. This makes up a semantic built-in the application's prism feature. On the other hand, in a command allowing the end-user to set a texture to a face he/she selects, you must create a reference-key to the selected face if it has not previously been referenced in any feature (as in the case of one of the lateral faces of the prism).

When you create a reference-key to selected topological elements (faces, edges or vertices), OCAF attaches to the reference-key information defining the selected topology — the Naming attribute. For example, it may be the faces to which a selected edge is common to. This information, as well as information about the evolution of the topology at each modeling step (the modified, updated and deleted faces), is used by the naming algorithm to maintain the topology attached to the reference-key. As such, on a parametrized model, after modifying the value of a parameter, the reference-keys still address the appropriate faces, even if their geometry has changed. Consequently, you change the size of the cube shown in the figure above, the user texture stay attached to the right face.
Note As Topological naming is based on the reference-key and attributes such as Naming (selection information) and Shape (topology evolution information), OCAF is not coupled to the underlying modeling libraries. The only modeling services required by OCAF are the following:

- Each algorithm must provide information about the evolution of the topology (the list of faces modified, updated and deleted by the algorithm)
- Exploration of the geometric model must be available (a 3D model is made of faces bounded by close wires, themselves composed by a sequence of edges connected by their vertices)

Currently, OCAF uses the Open CASCADE Technology modeling libraries.

To design an OCAF-based data model, the application developer is encouraged to aggregate ready-to-use attributes instead of defining new attributes by inheriting from an abstract root class. There are two major advantages in using aggregation rather than inheritance:

- As you don’t implement data by defining new classes, the format of saved data provided with OCAF doesn’t change; so you don’t have to write the Save and Open functions
- The application can query the data at runtime if a particular attribute is available

Summary

- OCAF is based on a uniform reference-key model in which:
  - Reference-keys provide persistent identification of data;
  - Data, including geometry, are implemented as attributes attached to reference-keys;
  - Topological naming maintains the selected geometry attached to reference-keys in parametrized models;
- In many applications, the data format provided with OCAF doesn’t need to be extended;
- OCAF is not coupled to the underlying modeling libraries.
The Data Framework

Data Structure

The OCAF Data Framework is the Open CASCADE Technology realization of the reference-key model in a tree structure. It offers a single environment where data from different application components can be handled. This allows exchanging and modifying data simply, consistently, with a maximum level of information and stable semantics.

The building blocks of this approach are:

- The tag
- The label
- The attribute

As it has been mentioned earlier, the first label in a framework is the root label of the tree. Each label has a tag expressed as an integer value, and a label is uniquely defined by an entry expressed as a list of tags from the root, 0:1:2:1, for example.

Each label can have a list of attributes, which contain data, and several attributes can be attached to a label. Each attribute is identified by a GUID, and although a label may have several attributes attached to it, it must not have more than one attribute of a single GUID.

The sub-labels of a label are called its children. Conversely, each label, which is not the root, has a father. Brother labels cannot share the same tag.

The most important property is that a label's entry is its persistent address in the data framework.
In this image the circles contain tags of the corresponding labels. The lists of tags are located under the circles. The root label always has a zero tag.

The children of a root label are middle-level labels with tags 1 and 3. These labels are brothers.

List of tags of the right-bottom label is "0:3:4": this label has tag 4, its father (with entry "0:3") has tag 3, father of father has tag 0 (the root label always has "0" entry).
Examples of a Data Structure

Let's have a look at the example:

In the image the application for designing coffee machines first allocates a label for the machine unit. It then adds sub-labels for the main features (glass coffee pot, water receptacle and filter) which it refines as needed (handle and reservoir of the coffee pot and spout of the reservoir).

You now attach technical data describing the handle — its geometry and color — and the reservoir — its geometry and material. Later on, you can modify the handle's geometry without changing its color — both remain attached to the same label.
The data structure of the coffee machine

The nesting of labels is key to OCAF. This allows a label to have its own structure with its local addressing scheme which can be reused in a more complex structure. Take, for example, the coffee machine. Given that the coffee pot's handle has a label of tag [1], the entry for the handle in the context of the coffee pot only (without the machine unit) is [0:1:1]. If you now model a coffee machine with two coffee pots, one at the label [1], the second at the label [4] in the machine unit, the handle of the first pot would have the entry [0:1:1:1] whereas the handle of the second pot would be [0:1:4:1]. This way, we avoid any confusion between coffee pot handles.

Another example is the application for designing table lamps. The first label is allocated to the lamp unit.
The root label cannot have brother labels. Consequently, various lamps in the framework allocation correspond to the sub-labels of the root label. This allows avoiding any confusion between table lamps in the data framework. Different lamp parts have different material, color and other attributes, so a child label of the lamp with the specified tags is allocated for each sub-unit of the lamp:

- a lamp-shade label with tag 1
- a bulb label with tag 2
- a stem label with tag 3

Label tags are chosen at will. They are only identifiers of the lamp parts. Now you can refine all units: by setting geometry, color, material and other information about the lamp or its parts to the specified label. This information is placed into special attributes of the label: the pure label contains no data – it is only a key to access data.

Remember that tags are private addresses without any meaning outside the data framework. It would, for instance, be an error to use part names as tags. These might change or be removed from production in next versions of the application, whereas the exact form of that part might be reused in your design, the part name could be integrated into the framework as an attribute.
So, after the user changes the lamp design, only corresponding attributes are changed, but the label structure is maintained. The lamp shape must be recreated by new attribute values and attributes of the lamp shape must refer to a new shape.

The previous figure shows the table-lamps document structure: each child of the root label contains a lamp shape attribute and refers to the sub-labels, which contain some design information about corresponding sub-units.

The data framework structure allows to create more complex structures: each lamp label sub-label may have children labels with more detailed information about parts of the table lamp and its components.

Note that the root label can have attributes too, usually global attributes: the name of the document, for example.
Tag

A tag is an integer, which identifies a label in two ways:

- Relative identification
- Absolute identification.

In relative identification, a label's tag has a meaning relative to the father label only. For a specific label, you might, for example, have four child labels identified by the tags 2, 7, 18, 100. In using relative identification, you ensure that you have a safe scope for setting attributes.

In absolute identification, a label's place in the data framework is specified unambiguously by a colon-separated list of tags of all the labels from the one in question to the root of the data framework. This list is called an entry. `TDF_Tool::TagList` allows retrieving the entry for a specific label.

In both relative and absolute identification, it is important to remember that the value of an integer has no intrinsic semantics whatsoever. In other words, the natural sequence that integers suggest, i.e. 0, 1, 2, 3, 4 ... – has no importance here. The integer value of a tag is simply a key.

The tag can be created in two ways:

- Random delivery
- User-defined delivery

As the names suggest, in random delivery, the tag value is generated by the system in a random manner. In user-defined delivery, you assign it by passing the tag as an argument to a method.

Creating child labels using random delivery of tags

To append and return a new child label, you use `TDF_TagSource::NewChild`. In the example below, the argument `level2`, which is passed to `NewChild`, is a `TDF_Label`.

```cpp
TDF_Label child1 = TDF_TagSource::NewChild (level2);```
Creation of a child label by user delivery from a tag

The other way to create a child label from a tag is by user delivery. In other words, you specify the tag, which you want your child label to have.

To retrieve a child label from a tag which you have specified yourself, you need to use `TDF_Label::FindChild` and `TDF_Label::Tag` as in the example below. Here, the integer 3 designates the tag of the label you are interested in, and the Boolean false is the value for the argument `create`. When this argument is set to `false`, no new child label is created.

```cpp
TDF_Label achild = root.FindChild(3, Standard_False);
if (!achild.IsNull()) {
    Standard_Integer tag = achild.Tag();
}
```
Label

The tag gives a persistent address to a label. The label – the semantics of the tag – is a place in the data framework where attributes, which contain data, are attached. The data framework is, in fact, a tree of labels with a root as the ultimate father label.

Label can not be deleted from the data framework, so, the structure of the data framework that has been created can not be removed while the document is opened. Hence any kind of reference to an existing label will be actual while an application is working with the document.

Label creation

Labels can be created on any labels, compared with brother labels and retrieved. You can also find their depth in the data framework (depth of the root label is 0, depth of child labels of the root is 1 and so on), whether they have children or not, relative placement of labels, data framework of this label. The class TDF_Label offers the above services.

Creating child labels

To create a new child label in the data framework using explicit delivery of tags, use TDF_Label::FindChild.

```cpp
//creating a label with tag 10 at Root
TDF_Label lab1 = aDF->Root().FindChild(10);

//creating labels 7 and 2 on label 10
TDF_Label lab2 = lab1.FindChild(7);
TDF_Label lab3 = lab1.FindChild(2);
```

You could also use the same syntax but add the Boolean true as a value of the argument create. This ensures that a new child label will be created if none is found. Note that in the previous syntax, this was also the case since create is true by default.
TDF_Label level1 = root.FindChild(3,Standard_True);
TDF_Label level2 = level1.FindChild(1,Standard_True);

Retrieving child labels

You can retrieve child labels of your current label by iteration on the first level in the scope of this label.

TDF_Label current;
//
for (TDF_ChildIterator it1 (current,Standard_False);
    it1.More(); it1.Next()) {
    achild = it1.Value();
    //
    // do something on a child (level 1)
    //
}

You can also retrieve all child labels in every descendant generation of your current label by iteration on all levels in the scope of this label.

for (TDF_ChildIterator itall (current,Standard_True);
    itall.More(); itall.Next()) {
    achild = itall.Value();
    //
    // do something on a child (all levels)
    //
}

Using TDF_Tool::Entry with TDF_ChildIterator you can retrieve the entries of your current label's child labels as well.

void DumpChildren(const TDF_Label& aLabel)
{
    TDF_ChildIterator it;
    TCollection_AsciiString es;
    for (it.Initialize(aLabel,Standard_True); it.More();
         it.Next()){

Retrieving the father label

Retrieving the father label of a current label.

```
TDF_Label father = aclild.Father();
isroot = father.IsRoot();
```
**Attribute**

The label itself contains no data. All data of any type whatsoever – application or non-application – is contained in attributes. These are attached to labels, and there are different types for different types of data. OCAF provides many ready-to-use standard attributes such as integer, real, constraint, axis and plane. There are also attributes for topological naming, functions and visualization. Each type of attribute is identified by a GUID.

The advantage of OCAF is that all of the above attribute types are handled in the same way. Whatever the attribute type is, you can create new instances of them, retrieve them, attach them to and remove them from labels, "forget" and "remember" the attributes of a particular label.

**Retrieving an attribute from a label**

To retrieve an attribute from a label, you use `TDF_Label::FindAttribute`. In the example below, the GUID for integer attributes, and `INT`, a handle to an attribute are passed as arguments to `FindAttribute` for the current label.

```cpp
if(current.FindAttribute(TDataStd_Integer::GetID(),INT))
{
    // the attribute is found
}
else
{
    // the attribute is not found
}
```

**Identifying an attribute using a GUID**

You can create a new instance of an attribute and retrieve its GUID. In the example below, a new integer attribute is created, and its GUID is passed to the variable `guid` by the method ID inherited from
**TDF_Attribute.**

Handle(TDataStd_Integer) INT = new TDataStd_Integer();
Standard_GUID guid = INT->ID();

**Attaching an attribute to a label**

To attach an attribute to a label, you use `TDF_Label::Add`. Repetition of this syntax raises an error message because there is already an attribute with the same GUID attached to the current label.

`TDF_Attribute::Label` for `INT` then returns the label `attach` to which `INT` is attached.

```cpp
current.Add (INT); // INT is now attached to current
current.Add (INT); // causes failure
TDF_Label attach = INT->Label();
```

**Testing the attachment to a label**

You can test whether an attribute is attached to a label or not by using `TDF_Attribute::IsA` with the GUID of the attribute as an argument. In the example below, you test whether the current label has an integer attribute, and then, if that is so, how many attributes are attached to it. `TDataStd_Integer::GetID` provides the GUID argument needed by the method `IsAttribute`.

`TDF_Attribute::HasAttribute` tests whether there is an attached attribute, and `TDF_Tool::NbAttributes` returns the number of attributes attached to the label in question, e.g. `current`.

```cpp
// Testing of attribute attachment
//
if (current.IsA(TDataStd_Integer::GetID())) { // the label has an Integer attribute attached
}
if (current.HasAttribute()) { // the label has at least one attribute attached
```
Removing an attribute from a label

To remove an attribute from a label, you use `TDF_Label::Forget` with the GUID of the deleted attribute. To remove all attributes of a label, `TDF_Label::ForgetAll`.

```cpp
current.Forget(TDataStd_Integer::GetID()); // integer attribute is now not attached to current label
current.ForgetAll(); // current has now 0 attributes attached
```

Specific attribute creation

If the set of existing and ready to use attributes implementing standard data types does not cover the needs of a specific data presentation task, the user can build his own data type and the corresponding new specific attribute implementing this new data type.

There are two ways to implement a new data type: create a new attribute (standard approach), or use the notion of User Attribute by means of a combination of standard attributes (alternative way).

In order to create a new attribute in the standard way, create a class inherited from `TDF_Attribute` and implement all purely virtual and necessary virtual methods:

- **ID()** – returns a unique GUID of a given attribute
- **Restore(attribute)** – sets fields of this attribute equal to the fields of a given attribute of the same type
- **Paste(attribute, relocation_table)** – sets fields of a given attribute equal to the field values of this attribute; if the attribute has references to some objects of the data framework and relocation_table has this element, then the given attribute must also refer to this object.
- **NewEmpty()** – returns a new attribute of this class with empty fields
- **Dump(stream)** – outputs information about a given attribute to a given stream debug (usually outputs an attribute of type string only)

Methods **NewEmpty**, **Restore** and **Paste** are used for the common transactions mechanism (Undo/Redo commands). If you don’t need this attribute to react to undo/redo commands, you can write only stubs of these methods, else you must call the Backup method of the **TDF_Attribute** class every time attribute fields are changed.

To enable possibility to save / restore the new attribute in XML format, do the following:

1. Create a new package with the nameXml[package name] (for example XmlMyAttributePackage) containing class XmlMyAttributePackage_MyAttributeDriver. The new class inherits XmlMDF_ADriver class and contains the translation functionality: from transient to persistent and vice versa (see the realization of the standard attributes in the packages XmlMDataStd, for example). Add package method AddDrivers which adds your class to a driver table (see below).
2. Create a new package (or do it in the current one) with two package methods:
   - **Factory**, which loads the document storage and retrieval drivers; and
   - **AttributeDrivers**, which calls the methods AddDrivers for all packages responsible for persistence of the document.
3. Create a plug-in implemented as an executable (see example XmlPlugin). It calls a macro PLUGIN with the package name where you implemented the method Factory.

To enable possibility to save / restore the new attribute in binary format, do the following:

1. Create a new package with nameBin[package name] (for example BinMyAttributePackage) containing a class BinMyAttributePackage_MyAttributeDriver. The new class inherits BinMDF_ADriver class and contains the translation functionality: from transient to persistent and vice versa (see the realization of the standard attributes in the packages BinMDataStd, for example). Add package method AddDrivers, which adds your class to a driver table.
2. Create a new package (or do it in the current one) with two package methods:
   - Factory, which loads the document storage and retrieval drivers; and
   - AttributeDrivers, which calls the methods AddDrivers for all packages responsible for persistence of the document.
3. Create a plug-in implemented as an executable (see example BinPlugin). It calls a macro PLUGIN with the package name where you implemented the method Factory. See Saving the document and Opening the document from a file for the description of document save/open mechanisms.

If you decided to use the alternative way (create a new attribute by means of UAttribute and a combination of other standard attributes), do the following:

1. Set a TDataStd_UAttribute with a unique GUID attached to a label. This attribute defines the semantics of the data type (identifies the data type).
2. Create child labels and allocate all necessary data through standard attributes at the child labels.
3. Define an interface class for access to the data of the child labels.

Choosing the alternative way of implementation of new data types allows to forget about creating persistence classes for your new data type. Standard persistence classes will be used instead. Besides, this way allows separating the data and the methods for access to the data (interfaces). It can be used for rapid development in all cases when requirements to application performance are not very high.

Let’s study the implementation of the same data type in both ways by the example of transformation represented by gp_Trsf class. The class gp_Trsf defines the transformation according to the type (gp_TrsfForm) and a set of parameters of the particular type of transformation (two points or a vector for translation, an axis and an angle for rotation, and so on).

1. The first way: creation of a new attribute. The implementation of the transformation by creation of a new attribute is represented in the Samples.
2. The second way: creation of a new data type by means of
combination of standard attributes. Depending on the type of transformation it may be kept in data framework by different standard attributes. For example, a translation is defined by two points. Therefore the data tree for translation looks like this:

- Type of transformation \((gp\_Translation)\) as \(TDataStd\_Integer\);
- First point as \(TDataStd\_RealArray\) (three values: \(X1, Y1\) and \(Z1\));
- Second point as \(TDataStd\_RealArray\) (three values: \(X2, Y2\) and \(Z2\)).

Data tree for translation

If the type of transformation is changed to rotation, the data tree looks like this:

- Type of transformation \((gp\_Rotation)\) as \(TDataStd\_Integer\);
- Point of axis of rotation as \(TDataStd\_RealArray\) (three values: \(X, Y\) and \(Z\));
- Axis of rotation as \(TDataStd\_RealArray\) (three values: \(DX, DY\) and \(DZ\));
- Angle of rotation as \(TDataStd\_Real\).
The attribute \textit{TDataStd_UAttribute} with the chosen unique GUID identifies the data type. The interface class initialized by the label of this attribute allows access to the data container (type of transformation and the data of transformation according to the type).
Compound documents

As the identification of data is persistent, one document can reference data contained in another document, the referencing and referenced documents being saved in two separate files.

Let’s look at the coffee machine application again. The coffee pot can be placed in one document. The coffee machine document then includes an occurrence — a positioned copy — of the coffee pot. This occurrence is defined by an XLink attribute (the external Link) which references the coffee pot of the first document (the XLink contains the relative path of the coffee pot document and the entry of the coffee pot data [0:1]).
The coffee machine compound document

In this context, the end-user of the coffee machine application can open the coffee pot document, modify the geometry of, for example, the reservoir, and overwrite the document without worrying about the impact of the modification in the coffee machine document. To deal with this situation, OCAF provides a service which allows the application to check whether a document is up-to-date. This service is based on a modification counter included in each document: when an external link is created, a copy of the referenced document counter is associated to the XLink in the referencing document. Providing that each modification of the referenced document increments its own counter, we can detect that the referencing document has to be updated by comparing the two counters (an update function importing the data referenced by an XLink into the referencing document is also provided).
Transaction mechanism

The Data Framework also provides a transaction mechanism inspired from database management systems: the data are modified within a transaction which is terminated either by a Commit if the modifications are validated or by an Abort if the modifications are abandoned — the data are then restored to the state it was in prior to the transaction. This mechanism is extremely useful for:

- Securing editing operations (if an error occurs, the transaction is abandoned and the structure retains its integrity)
- Simplifying the implementation of the **Cancel** function (when the end-user begins a command, the application may launch a transaction and operate directly in the data structure; abandoning the action causes the transaction to Abort)
- Executing **Undo** (at commit time, the modifications are recorded in order to be able to restore the data to their previous state)

The transaction mechanism simply manages a backup copy of attributes. During a transaction, attributes are copied before their first modification. If the transaction is validated, the copy is destroyed. If the transaction is abandoned, the attribute is restored to its initial value (when attributes are added or deleted, the operation is simply reversed).

Transactions are document-centered, that is, the application starts a transaction on a document. So, modifying a referenced document and updating one of its referencing documents requires two transactions, even if both operations are done in the same working session.
Standard Document Services

Overview

Standard documents offer ready-to-use documents containing a TDF-based data framework. Each document can contain only one framework.

The documents themselves are contained in the instantiation of a class \texttt{TDocStd\_Application} (or its descendant). This application manages the creation, storage and retrieval of documents.

You can implement undo and redo in your document, and refer from the data framework of one document to that of another one. This is done by means of external link attributes, which store the path and the entry of external links.

To sum up, standard documents alone provide access to the data framework. They also allow you to:

- Update external links
- Manage the saving and opening of data
- Manage the undo/redo functionality.
The Application

As a container for your data framework, you need a document, and your document must be contained in your application. This application will be a class \textit{TDocStd\_Application} or a class inheriting from it.

Creating an application

To create an application, use the following syntax.

\begin{verbatim}
Handle(TDocStd\_Application) app = new TDocStd\_Application();
\end{verbatim}

Creating a new document

To the application which you declared in the previous example (4.2.1), you must add the document \textit{doc} as an argument of \textit{TDocStd\_Application::NewDocument}.

\begin{verbatim}
Handle(TDocStd\_Document) doc;
app->NewDocument("NewDocumentFormat", doc);
\end{verbatim}

Here "NewDocumentFormat" is identifier of the format of your document. OCCT defines several standard formats, distinguishing by a set of supported OCAF attributes, and method of encoding (e.g. binary data or XML), described below. If your application defines specific OCAF attributes, you need to define your own format for it.

Retrieving the application to which the document belongs

To retrieve the application containing your document, you use the syntax below.

\begin{verbatim}
app = Handle(TDocStd\_Application)::DownCast (doc->Application());
\end{verbatim}
The Document

The document contains your data framework, and allows you to retrieve this framework, recover its main label, save it in a file, and open or close this file.

Accessing the main label of the framework

To access the main label in the data framework, you use `TDocStd_Document::Main` as in the example below. The main label is the first child of the root label in the data framework, and has the entry 0:1.

```cpp
TDF_Label label = doc->Main();
```

Retrieving the document from a label in its framework

To retrieve the document from a label in its data framework, you use `TDocStd_Document::Get` as in the example below. The argument `label` passed to this method is an instantiation of `TDF_Label`.

```cpp
doc = TDocStd_Document::Get(label);
```

Defining storage format

OCAF uses a customizable mechanism for storage of the documents. In order to use OCAF persistence to save and read your documents to / from the file, you need to define one or several formats in your application.

For that, use method `TDocStd_Application::DefineFormat()`, for instance:

```cpp
app->DefineFormat("NewDocumentFormat", "New format for OCAF documents", "ndf",
                 new
                 NewDocumentFormat_RetrievalDriver(),
                 new
                 NewDocumentFormat_StorageDriver());
```
This example defines format "NewDocumentFormat" with a default file extension "ndf", and instantiates drivers for reading and storing documents from and to that format. Either of the drivers can be null, in this case the corresponding action will not be supported for that format.

OCAF provides several standard formats, each covering some set of OCAF attributes:

<table>
<thead>
<tr>
<th>Format</th>
<th>Persistent toolkit</th>
<th>OCAF attributes covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy formats (read only)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCC-StdLite</td>
<td>TKStdL</td>
<td>TKLCAF</td>
</tr>
<tr>
<td>MDTV-Standard</td>
<td>TKStd</td>
<td>TKLCAF + TKCAF</td>
</tr>
<tr>
<td>Binary formats</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BinLOcaf</td>
<td>TKBinL</td>
<td>TKLCAF</td>
</tr>
<tr>
<td>BinOcaf</td>
<td>TKBin</td>
<td>TKLCAF + TKCAF</td>
</tr>
<tr>
<td>BinXCAF</td>
<td>TKBinXCAF</td>
<td>TKLCAF + TKCAF + TKXCAF</td>
</tr>
<tr>
<td>TObjBin</td>
<td>TKBinTObj</td>
<td>TKLCAF + TKTObj</td>
</tr>
<tr>
<td>XML formats</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XmlLOcaf</td>
<td>TKXmlL</td>
<td>TKLCAF</td>
</tr>
<tr>
<td>XmlOcaf</td>
<td>TKXml</td>
<td>TKLCAF + TKCAF</td>
</tr>
<tr>
<td>XmlXCAF</td>
<td>TKXmlXCAF</td>
<td>TKLCAF + TKCAF + TKXCAF</td>
</tr>
<tr>
<td>TObjXml</td>
<td>TKXmlTObj</td>
<td>TKLCAF + TKTObj</td>
</tr>
</tbody>
</table>

For convenience, these toolkits provide static methods `DefineFormat()` accepting handle to application. These methods allow defining corresponding formats easily, e.g.:

```cpp
BinDrivers::DefineFormat (app); // define format "BinOcaf"
```

Use these toolkits as an example for implementation of persistence drivers for custom attributes, or new persistence formats.

The application can define several storage formats. On save, the format specified in the document (see `TDocStd_Document::StorageFormat()`) will be used (save will fail if that format is not defined in the application).
On reading, the format identifier stored in the file is used and recorded in the document.

**Defining storage format by resource files**

The alternative method to define formats is via usage of resource files. This method was used in earlier versions of OCCT and is considered as deprecated since version 7.1.0. This method allows loading persistence drivers on demand, using plugin mechanism.

To use this method, create your own application class inheriting from `TDocStd_Application`, and override method `ResourcesName()`. That method should return a string with a name of resource file, e.g. "NewDocumentFormat", which will contain a description of the format.

Then create that resource file and define the parameters of your format:

```plaintext
NewDocumentFormat.Description: New Document Format Version 1.0
NewDocumentFormat.FileExtension: ndf
NewDocumentFormat.StoragePlugin: bb5aa176-c65c-4c84-862e-6b7c1fe16921
NewDocumentFormat.RetrievalPlugin: 76fb4c04-ea9a-46aa-88a2-25f6a228d902
```

The GUIDs should be unique and correspond to the GUIDs supported by relevant plugin. You can use an existing plugins (see the table above) or create your own.

Finally, make a copy of the resource file "Plugin" from `$CASROOT/src/StdResource` and, if necessary, add the definition of your plugin in it, for instance:

```plaintext
bb5aa176-c65c-4c84-862e-6b7c1fe16921.Location: TKNewFormat
76fb4c04-ea9a-46aa-88a2-25f6a228d902.Location: TKNewFormat
```

In order to have these resource files loaded during the program
execution, it is necessary to set two environment variables: 
\textit{CSF\_PluginDefaults} and \textit{CSF\_NewFormatDefaults}. For example, set the 
files in the directory \textit{MyApplicationPath/MyResources}:

\begin{verbatim}
setenv CSF_PluginDefaults MyApplicationPath/MyResources
setenv CSF_NewFormatDefaults MyApplicationPath/MyResources
\end{verbatim}

\section*{Saving a document}

To save the document, make sure that its parameter \textit{StorageFormat()} 
corresponds to one of the formats defined in the application, and use 
method \textit{TDocStd\_Application::SaveAs}, for instance:

\begin{verbatim}
app->SaveAs(doc, "\textpath{tmp/example.caf}");
\end{verbatim}

\section*{Opening the document from a file}

To open the document from a file where it has been previously saved, 
you can use \textit{TDocStd\_Application::Open} as in the example below. The 
arguments are the path of the file and the document saved in this file.

\begin{verbatim}
app->Open("\textpath{tmp/example.caf}", doc);
\end{verbatim}

\section*{Cutting, copying and pasting inside a document}

To cut, copy and paste inside a document, use the class 
\textit{TDF\_CopyLabel}.

In fact, you must define a \textit{Label}, which contains the temporary value of a 
cut or copy operation (say, in \textit{Lab\_Clipboard}). You must also define two 
other labels:

- The data container (e.g. \textit{Lab\_source})
- The destination of the copy (e.g. \textit{Lab\_Target})

\begin{verbatim}
Copy = copy (Lab_Source => Lab_Clipboard)
Cut = copy + Lab_Source.ForgetAll() // command clear
\end{verbatim}
Paste = copy (Lab_Clipboard => Lab_target)

So we need a tool to copy all (or a part) of the content of a label and its sub-label, to another place defined by a label.

TDF_CopyLabel aCopy;
TDF_IDFilter aFilter (Standard_False);

//Don't copy TDataStd_TreeNode attribute

   aFilter.Ignore(TDataStd_TreeNode::GetDefaultTreeID());
   aCopy.Load(aSource, aTarget);
   aCopy.UseFilter(aFilter);  aCopy.Perform();

// copy the data structure to clipboard

return aCopy.IsDone();  

The filter is used to forbid copying a specified type of attribute.

You can also have a look at the class TDF_Closure, which can be useful to determine the dependencies of the part you want to cut from the document.
External Links

External links refer from one document to another. They allow you to update the copy of data framework later on.

Note that documents can be copied with or without a possibility of updating an external link.

Copying the document

With the possibility of updating it later

To copy a document with a possibility of updating it later, you use `TDocStd_XLinkTool::CopyWithLink`.

Handle(TDocStd_Document) doc1;
Handle(TDocStd_Document) doc2;

TDF_Label source = doc1->GetData()->Root();
TDF_Label target = doc2->GetData()->Root();
TDocStd_XLinkTool XLinkTool;
XLinkTool.CopyWithLink(target, source);

Now the target document has a copy of the source document. The copy also has a link in order to update the content of the copy if the original changes.

In the example below, something has changed in the source document. As a result, you need to update the copy in the target document. This copy is passed to TDocStd_XLinkTool::UpdateLink as the argument target.

XLinkTool.UpdateLink(target);

**Without any link between the copy and the original**

You can also create a copy of the document with no link between the original and the copy. The syntax to use this option is TDocStd_XLinkTool::Copy. The copied document is again represented by the argument target, and the original – by source.

XLinkTool.Copy(target, source);
OCAF Shape Attributes

Overview

A topological attribute can be seen as a hook into the topological structure. It is possible to attach data to define references to it.

OCAF shape attributes are used for topology objects and their evolution access. All topological objects are stored in one TNaming_UsedShapes attribute at the root label of the data framework. This attribute contains a map with all topological shapes used in a given document.

The user can add the TNaming_NamedShape attribute to other labels. This attribute contains references (hooks) to shapes from the TNaming_UsedShapes attribute and an evolution of these shapes. The TNaming_NamedShape attribute contains a set of pairs of hooks: to the Old shape and to a New shape (see the following figure). It allows not only to get the topological shapes by the labels, but also to trace the evolution of the shapes and to correctly update dependent shapes by the changed one.

If a shape is newly created, then the old shape of a corresponding named shape is an empty shape. If a shape is deleted, then the new shape in this named shape is empty.
Shape attributes in data framework.

Different algorithms may dispose sub-shapes of the result shape at the individual labels depending on whether it is necessary to do so:

- If a sub-shape must have some extra attributes (material of each face or color of each edge). In this case a specific sub-shape is placed to a separate label (usually to a sub-label of the result shape label) with all attributes of this sub-shape.
- If the topological naming algorithm is needed, a necessary and sufficient set of sub-shapes is placed to child labels of the result shape label. As usual, for a basic solid and closed shells, all faces of the shape are disposed.

TNaming_NamedShape may contain a few pairs of hooks with the same evolution. In this case the topology shape, which belongs to the named shape is a compound of new shapes.

Consider the following example. Two boxes (solids) are fused into one solid (the result one). Initially each box was placed to the result label as a named shape, which has evolution PRIMITIVE and refers to the corresponding shape of the TNaming_UsedShapes map. The box result label has a material attribute and six child labels containing named shapes of Box faces.

![Resulting box](image_url)

After the fuse operation a modified result is placed to a separate label as
a named shape, which refers to the old shape (one of the boxes) and to the new shape resulting from the fuse operation, and has evolution MODIFY (see the following figure).

Named shapes, which contain information about modified faces, belong to the fuse result sub-labels:

- sub-label with tag 1 – modified faces from box 1,
- sub-label with tag 2 – modified faces from box 2.
This is necessary and sufficient information for the functionality of the right naming mechanism: any sub-shape of the result can be identified unambiguously by name type and set of labels, which contain named shapes:

- face F1' as a modification of face F11
- face F1" as generation of face F12
- edges as an intersection of two contiguous faces
- vertices as an intersection of three contiguous faces

After any modification of source boxes the application must automatically rebuild the naming entities: recompute the named shapes of the boxes (solids and faces) and fuse the resulting named shapes (solids and faces) that reference to the new named shapes.
Registering shapes and their evolution

When using TNaming_NamedShape to create attributes, the following fields of an attribute are filled:

- A list of shapes called the "old" and the "new" shapes. A new shape is recomputed as the value of the named shape. The meaning of this pair depends on the type of evolution.
- The type of evolution, which is a term of the TNaming_Evolution enumeration used for the selected shapes that are placed to the separate label:
  - PRIMITIVE – newly created topology, with no previous history;
  - GENERATED – as usual, this evolution of a named shape means, that the new shape is created from a low-level old shape (a prism face from an edge, for example);
  - MODIFY – the new shape is a modified old shape;
  - DELETE – the new shape is empty; the named shape with this evolution just indicates that the old shape topology is deleted from the model;
  - SELECTED – a named shape with this evolution has no effect on the history of the topology.

Only pairs of shapes with equal evolution can be stored in one named shape.
Using naming resources

The class *TNaming_Builder* allows creating a named shape attribute. It has a label of a future attribute as an argument of the constructor. Respective methods are used for the evolution and setting of shape pairs. If for the same TNaming_Builder object a lot of pairs of shapes with the same evolution are given, then these pairs would be placed in the resulting named shape. After the creation of a new object of the TNaming_Builder class, an empty named shape is created at the given label.

```cpp
// a new empty named shape is created at "label"
TNaming_Builder builder(label);
// set a pair of shapes with evolution GENERATED
builder.Generated(oldshape1,newshape1);
// set another pair of shapes with the same evolution
builder.Generated(oldshape2,newshape2);
// get the result - TNaming_NamedShape attribute
Handle(TNaming_NamedShape) ns = builder.NamedShape();
```
Reading the contents of a named shape attribute

You can use the method `TNaming_NamedShape::Evolution()` to get the evolution of this named shape and the method `TNaming_NamedShape::Get()` to get a compound of new shapes of all pairs of this named shape.

More detailed information about the contents of the named shape or about the modification history of a topology can be obtained with the following:

- **TNaming_Tool** provides a common high-level functionality for access to the named shapes contents:
  - The method `GetShape(Handle(TNaming_NamedShape))` returns a compound of new shapes of the given named shape;
  - The method `CurrentShape(Handle(TNaming_NamedShape))` returns a compound of the shapes, which are latest versions of the shapes from the given named shape;
  - The method `NamedShape(TopoDS_Shape,TDF_Label)` returns a named shape, which contains a given shape as a new shape. A given label is any label from the data framework – it just gives access to it.
- **TNaming_Iterator** gives access to the named shape and hooks pairs.

```cpp
// create an iterator for a named shape
TNaming_Iterator iter(namedshape);
// iterate while some pairs are not iterated
while(iter.More()) {
    // get the new shape from the current pair
    TopoDS_Shape newshape = iter.NewShape();
    // get the old shape from the current pair
    TopoDS_Shape oldshape = iter.OldShape();
    // do something...

    // go to the next pair
    iter.Next();
```
Topological naming

The Topological Naming mechanism is based on 3 components:

- History of the used modeling operation algorithm;
- Registering of the built result in Data Framework (i.e. loading the necessary elements of the extracted history in OCAF document);
- Selection / Recomputation of a "selected" sub-shape of the algorithm result.

To get the expected result the work of the three components should be synchronized and the rules of each component should be respected.

Algorithm history

The "correct" history of a used modeling operation serves the basis of naming mechanism. It should be provided by the algorithm supporting the operation. The history content depends on the type of the topological result. The purpose of the history is to provide all entities for consistent and correct work of the Selection / Recomputation mechanism. The table below presents expected types of entities depending on the result type.

<table>
<thead>
<tr>
<th>Result type</th>
<th>Type of sub-shapes to be returned by history of algorithm</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solid or closed shell</td>
<td>Faces</td>
<td>All faces</td>
</tr>
<tr>
<td>Open shell or single face</td>
<td>Faces and edges of opened boundaries only</td>
<td>All faces plus all edges of opened boundaries</td>
</tr>
<tr>
<td>Closed wire</td>
<td>Edges</td>
<td>All edges</td>
</tr>
<tr>
<td>Opened wire</td>
<td>Edges and ending vertexes</td>
<td>All edges plus ending vertexes of the wire</td>
</tr>
<tr>
<td>Edge</td>
<td>Vertexes</td>
<td>Two vertexes are expected</td>
</tr>
</tbody>
</table>
The history should return (and track) only elementary types of sub-shapes, i.e. Faces, Edges and Vertexes, while other so-called aggregation types: Compounds, Shells, Wires, are calculated by Selection mechanism automatically.

There are some simple exceptions for several cases. For example, if the Result contains a seam edge – in conical, cylindrical or spherical surfaces – this seam edge should be tracked by the history and in addition should be defined before the types. All degenerated entities should be filtered and excluded from consideration.

**Loading history in data framework**

All elements returned by the used algorithm according to the aforementioned rules should be put in the Data Framework (or OCAF document in other words) consequently in linear order under the so-called **Result Label**.

The "Result Label" is `TDF_label` used to keep the algorithm result `Shape` from `TopoDS` in `NamedShape` attribute. During loading sub-shapes of the result in Data Framework should be used the rules of chapter **Registering shapes and their evolution**. These rules are also applicable for loading the main shape, i.e. the resulting shape produced by the modeling algorithm.

**Selection / re-computation mechanism**

When the Data Framework is filled with all impacted entities (including the data structures resulting from the current modeling operation and the data structures resulting from the previous modeling operations, on which the current operation depends) any sub-shape of the current result can be **selected**, i.e. the corresponding new naming data structures, which support this functionality, can be produced and kept in the Data Framework.
One of the user interfaces for topological naming is the class \textit{TNaming\_Selector}. It implements the above mentioned sub-shape "selection" functionality as an additional one. I.e. it can be used for:

- Storing the selected shape on a label – its \textbf{Selection};
- Accessing the named shape – check the kept value of the shape
- Update of this naming – recomputation of an earlier selected shape.

The selector places a new named shape with evolution \textbf{SELECTED} to the given label. The selector creates a \textbf{name} of the selected shape, which is a unique description (data structure) of how to find the selected topology using as resources:

- the given context shape, i.e. the main shape kept on \textbf{Result Label}, which contains a selected sub-shape,
- its evolution and
- naming structure.

After any modification of a context shape and update of the corresponding naming structure, it is necessary to call method \textit{TNaming\_Selector::Solve}. If the naming structure, i.e. the above mentioned \textbf{name}, is correct, the selector automatically updates the selected sub-shape in the corresponding named shape, else it fails.
Exploring shape evolution

The class **TNaming_Tool** provides a toolkit to read current data contained in the attribute.

If you need to create a topological attribute for existing data, use the method **NamedShape**.

```cpp
class MyPkg_MyClass
{
public: Standard_Boolean SameEdge (const Handle(CafTest_Line)& L1, const Handle(CafTest_Line)& L2);
};

Standard_Boolean CafTest_MyClass::SameEdge (const Handle(CafTest_Line)& L1, const Handle(CafTest_Line)& L2)
{
    Handle(TNaming_NamedShape) NS1 = L1->NamedShape();
    Handle(TNaming_NamedShape) NS2 = L2->NamedShape();
    return BRepTools::Compare(NS1,NS2);
}
```
Example of topological naming usage

Topological naming is a mechanism of Open CASCADE aimed to keep reference to the selected shape. If, for example, we select a vertex of a solid shape and “ask” the topological naming to keep reference to this vertex, it will refer to the vertex whatever happens with the shape (translations, scaling, fusion with another shape, etc.).

Let us consider an example: imagine a wooden plate. The job is to drive several nails in it:

There may be several nails with different size and position. A Hammer should push each Nail exactly in the center point of the top surface. For this the user does the following:

- Makes several Nails of different height and diameter (according to the need),
- Chooses (selects) the upper surface of each Nail for the Hammer.

The job is done. The application should do the rest – the Hammer calculates a center point for each selected surface of the Nail and “strikes” each Nail driving it into the wooden plate.

What happens if the user changes the position of some Nails? How will the Hammer know about it? It keeps reference to the surface of each Nail. However, if a Nail is relocated, the Hammer should know the new
position of the selected surface. Otherwise, it will “strike” at the old position (keep the fingers away!)…

Topological naming mechanism should help the Hammer to obtain the relocated surfaces. The Hammer “asks” the mechanism to “resolve” the selected shapes by calling method `TNaming_Selection::Solve()` and the mechanism “returns” the modified surfaces located at the new position by calling `TNaming_Selector::NamedShape()`.

The topological naming is represented as a “black box” in the example above. Now it is time to make the box a little more “transparent”.

The application contains 3 functions:

- **Nail** – produces a shape representing a nail,
- **Translator** – translates a shape along the wooden plate,
- **Hammer** – drives the nail in the wooden plate.

Each function gives the topological naming some hints how to “re-solve” the selected sub-shapes:

- The Nail constructs a solid shape and puts each face of the shape into sub-labels:

```
  Nail
  └── Face 1 (top face)
    └── Face 2
      └── ...
```

**Distribution of faces through sub-labels of the Nail**

- The **Translator** moves a shape and registers modification for each face: it puts a pair: “old” shape – “new” shape at a sub-label of each moving Nail. The “old” shape represents a face of the Nail at the initial position. The “new” shape – is the same face, but at a new position:
Registration of relocation of faces of a Nail

How does it work?

- The Hammer selects a face of a Nail calling `TNaming_Selector::Select()`. This call makes a unique name for the selected shape. In our example, it will be a direct reference to the label of the top face of the Nail (Face 1).
- When the user moves a Nail along the wooden plate, the Translator registers this modification by putting the pairs: “old” face of the Nail – new face of the Nail into its sub-labels.
- When the Hammer calls `TNaming::Solve()`, the topological naming “looks” at the unique name of the selected shape and tries to re-solve it:
  - It finds the 1st appearance of the selected shape in the data tree – it is a label under the Nail function `Face 1`.
  - It follows the evolution of this face. In our case, there is only one evolution – the translation: `Face 1` (top face) – `Face 1'` (relocated top face). So, the last evolution is the relocated top face.
- Calling the method `TNaming_Selector::NamedShape()` the Hammer obtains the last evolution of the selected face – the relocated top face.

The job is done.

P.S. Let us say a few words about a little more complicated case – selection of a wire of the top face. Its topological name is an “intersection” of two faces. We remember that the Nail puts only faces under its label. So, the selected wire will represent an “intersection” of the top face and
the conic face keeping the “head” of the nail. Another example is a selected vertex. Its unique name may be represented as an “intersection” of three or even more faces (depends on the shape).
Standard Attributes

Overview

Standard attributes are ready-to-use attributes, which allow creating and modifying attributes for many basic data types. They are available in the packages *TDDataStd, TDataXtd* and *TDF*. Each attribute belongs to one of four types:

- Geometric attributes;
- General attributes;
- Relationship attributes;
- Auxiliary attributes.

Geometric attributes

- **Axis** – simply identifies, that the concerned *TNaming_NamedShape* attribute with an axis shape inside belongs to the same label;
- **Constraint** – contains information about a constraint between geometries: used geometry attributes, type, value (if exists), plane (if exists), "is reversed", "is inverted" and "is verified" flags;
- **Geometry** – simply identifies, that the concerned *TNaming_NamedShape* attribute with a specified-type geometry belongs to the same label;
- **Plane** – simply identifies, that the concerned *TNaming_NamedShape* attribute with a plane shape inside belongs to the same label;
- **Point** – simply identifies, that the concerned *TNaming_NamedShape* attribute with a point shape inside belongs to the same label;
- **Shape** – simply identifies, that the concerned *TNaming_NamedShape* attribute belongs to the same label;
- **PatternStd** – identifies one of five available pattern models (linear, circular, rectangular, circular rectangular and mirror);
- **Position** – identifies the position in 3d global space.

General attributes
- **AsciiString** – contains AsciiString value;
- **BooleanArray** – contains an array of Boolean;
- **BooleanList** – contains a list of Boolean;
- **ByteArray** – contains an array of Byte (unsigned char) values;
- **Comment** – contains a string – the comment for a given label (or attribute);
- **Expression** – contains an expression string and a list of used variables attributes;
- **ExtStringArray** – contains an array of *ExtendedString* values;
- **ExtStringList** – contains a list of *ExtendedString* values;
- **Integer** – contains an integer value;
- **IntegerArray** – contains an array of integer values;
- **IntegerList** – contains a list of integer values;
- **IntPackedMap** – contains a packed map of integers;
- **Name** – contains a string – the name of a given label (or attribute);
- **NamedData** – may contain up to 6 of the following named data sets (vocabularies): *DataMapOfStringInteger*, *DataMapOfStringReal*, *DataMapOfStringString*, *DataMapOfStringByte*, *DataMapOfStringHArray1OfInteger* or *DataMapOfStringHArray1OfReal*;
- **NoteBook** – contains a *NoteBook* object attribute;
- **Real** – contains a real value;
- **RealArray** – contains an array of real values;
- **RealList** – contains a list of real values;
- **Relation** – contains a relation string and a list of used variables attributes;
- **Tick** – defines a boolean attribute;
- **Variable** – simply identifies, that a variable belongs to this label; contains the flag *is constraint* and a string of used units ("mm", "m"...);
- **UAttribute** – attribute with a user-defined GUID. As a rule, this attribute is used as a marker, which is independent of attributes at the same label (note, that attributes with the same GUIDs can not belong to the same label).

**Relationship attributes**

- **Reference** – contains reference to the label of its own data framework;
- **ReferenceArray** – contains an array of references;
- **ReferenceList** – contains a list of references;
- **TreeNode** – this attribute allows to create an internal tree in the data framework; this tree consists of nodes with the specified tree ID; each node contains references to the father, previous brother, next brother, first child nodes and tree ID.

**Auxiliary attributes**

- **Directory** – high-level tool attribute for sub-labels management;
- **TagSource** – this attribute is used for creation of new children: it stores the tag of the last-created child of the label and gives access to the new child label creation functionality.

All attributes inherit class *TDF_Attribute*, so, each attribute has its own GUID and standard methods for attribute creation, manipulation, getting access to the data framework.
Services common to all attributes

Accessing GUIDs

To access the GUID of an attribute, you can use two methods:

- **Method GetID** is the static method of a class. It returns the GUID of any attribute, which is an object of a specified class (for example, `TDataStd_Integer` returns the GUID of an integer attribute). Only two classes from the list of standard attributes do not support these methods: `TDataStd_TreeNode` and `TDataStd_Uattribute`, because the GUIDs of these attributes are variable.
- **Method ID** is the method of an object of an attribute class. It returns the GUID of this attribute. Absolutely all attributes have this method: only by this identifier you can discern the type of an attribute.

To find an attribute attached to a specific label, you use the GUID of the attribute type you are looking for. This information can be found using the method `GetID` and the method `Find` for the label as follows:

```plaintext
Standard_GUID anID = MyAttributeClass::GetID();
Standard_Boolean HasAttribute = aLabel.Find(anID, anAttribute);
```

Conventional Interface of Standard Attributes

It is usual to create standard named methods for the attributes:

- **Method Set(label, [value])** is the static method, which allows to add an attribute to a given label. If an attribute is characterized by one value this method may set it.
- **Method Get()** returns the value of an attribute if it is characterized by one value.
- **Method Dump(Standard_OStream)** outputs debug information about a given attribute to a given stream.
The choice between standard and custom attributes

When you start to design an application based on OCAF, usually it is necessary to choose, which attribute will be used for allocation of data in the OCAF document: standard or newly-created?

It is possible to describe any model by means of standard OCAF attributes. However, it is still a question if this description will be efficient in terms of memory and speed, and, at the same time, convenient to use.

This depends on a particular model.

OCAF imposes the restriction that only one attribute type may be allocated to one label. It is necessary to take into account the design of the application data tree. For example, if a label should possess several double values, it is necessary to distribute them through several child sub-labels or use an array of double values.

Let us consider several boundary implementations of the same model in OCAF tree and analyze the advantages and disadvantages of each approach.

Comparison and analysis of approaches

Below are described two different model implementations: one is based on standard OCAF attributes and the other is based on the creation of a new attribute possessing all data of the model.

A load is distributed through the shape. The measurements are taken at particular points defined by (x, y and z) co-ordinates. The load is represented as a projection onto X, Y and Z axes of the local co-ordinate system at each point of measurement. A matrix of transformation is needed to convert the local co-ordinate system to the global one, but this is optional.

So, we have 15 double values at each point of measurement. If the number of such points is 100 000, for example, it means that we have to store 1 500 000 double values in the OCAF document.
The first approach consists in using standard OCAF attributes. Besides, there are several variants of how the standard attributes may be used:

- Allocation of all 1 500 000 double values as one array of double values attached to one label;
- Allocation of values of one measure of load (15 values) as one array of double values and attachment of one point of measure to one label;
- Allocation of each point of measure as an array of 3 double values attached to one label, the projection of load onto the local coordinate system axes as another array of 3 double values attached to a sub-label, and the matrix of projection (9 values) as the third array also attached to a sub-label.

Certainly, other variants are also possible.

**TDataStd_RealArray [1..1 500 000]**

**Allocation of all data as one array of double values**

The first approach to allocation of all data represented as one array of double values saves initial memory and is easy to implement. But access to the data is difficult because the values are stored in a flat array. It will be necessary to implement a class with several methods giving access to particular fields like the measurement points, loads and so on.

If the values may be edited in the application, it means that the whole array will be backed-up on each edition. The memory usage will increase very fast! So, this approach may be considered only in case of non-editable data.

Let's consider the allocation of data of each measurement point per label (the second case). In this case we create 100 000 labels – one label for each measurement point and attach an array of double values to these labels:
Allocation of data of each measurement point as arrays of double values

Now edition of data is safer as far as memory usage is concerned. Change of value for one measurement point (any value: point coordinates, load, and so on) backs-up only one small array of double values. But this structure (tree) requires more memory space (additional labels and attributes).

Besides, access to the values is still difficult and it is necessary to have a class with methods of access to the array fields.

The third case of allocation of data through OCAF tree is represented below:

Allocation of data into separate arrays of double values

In this case sub-labels are involved and we can easily access the values of each measurement point, load or matrix. We don't need an interface class with methods of access to the data (if it exists, it would help to use the data structure, but this is optional).

On the one hand, this approach requires more memory for allocation of the attributes (arrays of double values). On the other hand, it saves memory during the edition of data by backing-up only the small array containing the modified data. So, if the data is fully modifiable, this approach is more preferable.

Before making a conclusion, let's consider the same model implemented
through a newly created OCAF attribute.

For example, we might allocate all data belonging to one measurement point as one OCAF attribute. In this case we implement the third variant of using the standard attributes (see picture 3), but we use less memory (because we use only one attribute instead of three):

![Diagram of LoadMeasure with attributes x, y, z, Lx, Ly, Lz, A11, A12, ...](image)

**Allocation of data into newly created OCAF attribute**

The second variant of using standard OCAF attributes still has drawbacks: when data is edited, OCAF backs-up all values of the measurement point.

Let’s imagine that we have some non-editable data. It would be better for us to allocate this data separately from editable data. Back-up will not affect non-editable data and memory will not increase so much during data edition.

**Conclusion**

When deciding which variant of data model implementation to choose, it is necessary to take into account the application response time, memory allocation and memory usage in transactions.

Most of the models may be implemented using only standard OCAF attributes. Some other models need special treatment and require implementation of new OCAF attributes.
Visualization Attributes

Overview

Standard visualization attributes implement the Application Interactive Services (see Visualization User's Guide). in the context of Open CASCADE Technology Application Framework. Standard visualization attributes are AISViewer and Presentation and belong to the TPrsStd package.
Services provided

Defining an interactive viewer attribute

The class `TPrsStd_AISViewer` allows you to define an interactive viewer attribute. There may be only one such attribute per one data framework and it is always placed to the root label. So, it could be set or found by any label ("access label") of the data framework. Nevertheless the default architecture can be easily extended and the user can manage several Viewers per one framework by himself.

To initialize the AIS viewer as in the example below, use method `Find`.

```cpp
// "access" is any label of the data framework
Handle(TPrsStd_AISViewer) viewer =
    TPrsStd_AISViewer::Find(access)
```
Defining a presentation attribute

The class `TPrsStd_AISPresentation` allows you to define the visual presentation of document labels contents. In addition to various visual fields (color, material, transparency, `isDisplayed`, etc.), this attribute contains its driver GUID. This GUID defines the functionality, which will update the presentation every time when needed.

Creating your own driver

The abstract class `TPrsStd_Driver` allows you to define your own driver classes. Simply redefine the `Update` method in your new class, which will rebuild the presentation.

If your driver is placed to the driver table with the unique driver GUID, then every time the viewer updates presentations with a GUID identical to your driver's GUID, the `Update` method of your driver for these presentations must be called:

As usual, the GUID of a driver and the GUID of a displayed attribute are the same.

Using a container for drivers

You frequently need a container for different presentation drivers. The class `TPrsStd_DriverTable` provides this service. You can add a driver to the table, see if one is successfully added, and fill it with standard drivers.

To fill a driver table with standard drivers, first initialize the AIS viewer as
in the example above, and then pass the return value of the method
InitStandardDrivers to the driver table returned by the method Get. Then
attach a TNaming_NamedShape to a label and set the named shape in
the presentation attribute using the method Set. Then attach the
presentation attribute to the named shape attribute, and the
AIS_InteractiveObject, which the presentation attribute contains, will
initialize its drivers for the named shape. This can be seen in the example
below.

**Example**

```cpp
DriverTable::Get() -> InitStandardDrivers();
// next, attach your named shape to a label
TPrsStd_AISPresentation::Set(NS);
// here, attach the AISPresentation to NS.
```
Function Services

Function services aggregate data necessary for regeneration of a model. The function mechanism – available in the package *TFunction* – provides links between functions and any execution algorithms, which take their arguments from the data framework, and write their results inside the same framework.

When you edit any application model, you have to regenerate the model by propagating the modifications. Each propagation step calls various algorithms. To make these algorithms independent of your application model, you need to use function services.
Take, for example, the case of a modeling sequence made up of a box with the application of a fillet on one of its edges. If you change the height of the box, the fillet will need to be regenerated as well.
Finding functions, their owners and roots

The class *TFuntion_Function* is an attribute, which stores a link to a function driver in the data framework. In the static table *TFuntion_DriverTable* correspondence links between function attributes and drivers are stored.

You can write your function attribute, a driver for such attribute, which updates the function result in accordance to a given map of changed labels, and set your driver with the GUID to the driver table.

Then the solver algorithm of a data model can find the *Function* attribute on a corresponding label and call the *Execute* driver method to update the result of the function.
Storing and accessing information about function status

For updating algorithm optimization, each function driver has access to the `TFunction_Logbook` object that is a container for a set of touched, impacted and valid labels. Using this object a driver gets to know which arguments of the function were modified.
Propagating modifications

An application must implement its functions, function drivers and the common solver for parametric model creation. For example, check the following model:
The procedure of its creation is as follows:

- create a rectangular planar face $F$ with height 100 and width 200;
- create prism $P$ using face $F$ as a basis;
- create fillet $L$ at the edge of the prism;
- change the width of $F$ from 200 to 300;
- the solver for the function of face $F$ starts;
- the solver detects that an argument of the face $F$ function has been modified;
- the solver calls the driver of the face $F$ function for a regeneration of the face;
- the driver rebuilds face $F$ and adds the label of the face width argument to the logbook as touched and the label of the function of face $F$ as impacted;
- the solver detects the function of $P$ – it depends on the function of $F$;
- the solver calls the driver of the prism $P$ function;
- the driver rebuilds prism $P$ and adds the label of this prism to the logbook as impacted;
- the solver detects the function of $L$ – it depends on the function of $P$;
- the solver calls the $L$ function driver;
- the driver rebuilds fillet $L$ and adds the label of the fillet to the logbook as impacted.
Example of Function Mechanism Usage

Introduction

Let us describe the usage of the Function Mechanism of Open CASCADE Application Framework on a simple example. This example represents a "nail" composed by a cone and two cylinders of different radius and height:

These three objects (a cone and two cylinders) are independent, but the Function Mechanism makes them connected to each other and representing one object – a nail. The object "nail" has the following parameters:

- The position of the nail is defined by the apex point of the cone. The cylinders are built on the cone and therefore they depend on the position of the cone. In this way we define a dependency of the cylinders on the cone.
- The height of the nail is defined by the height of the cone. Let's consider that the long cylinder has 3 heights of the cone and the header cylinder has a half of the height of the cone.
- The radius of the nail is defined by the radius of the cone. The radius of the long cylinder coincides with this value. Let's consider that the
header cylinder has one and a half radiuses of the cone.

So, the cylinders depend on the cone and the cone parameters define the size of the nail.

It means that re-positioning the cone (changing its apex point) moves the nail, the change of the radius of the cone produces a thinner or thicker nail, and the change of the height of the cone shortens or prolongates the nail. It is suggested to examine the programming steps needed to create a 3D parametric model of the "nail".
Step 1: Data Tree

The first step consists in model data allocation in the OCAF tree. In other words, it is necessary to decide where to put the data.

In this case, the data can be organized into a simple tree using references for definition of dependent parameters:

- Nail
  - Cone
    - Position (x,y,z)
    - Radius
    - Height
  - Cylinder (stem)
    - Position = "Cone" position translated for "Cone" height along Z;
    - Radius = "Cone" radius;
    - Height = "Cone" height multiplied by 3;
  - Cylinder (head)
    - Position = "Long cylinder" position translated for "Long cylinder" height along Z;
    - Radius = "Long cylinder" radius multiplied by 1.5;
    - Height = "Cone" height divided by 2.

The "nail" object has three sub-leaves in the tree: the cone and two cylinders.

The cone object is independent.

The long cylinder representing a "stem" of the nail refers to the corresponding parameters of the cone to define its own data (position, radius and height). It means that the long cylinder depends on the cone.

The parameters of the head cylinder may be expressed through the cone parameters only or through the cone and the long cylinder parameters. It is suggested to express the position and the radius of the head cylinder through the position and the radius of the long cylinder, and the height of the head cylinder through the height of the
cone. It means that the head cylinder depends on the cone and the long cylinder.
Step 2: Interfaces

The interfaces of the data model are responsible for dynamic creation of the data tree of the represented at the previous step, data modification and deletion.

The interface called INail should contain the methods for creation of the data tree for the nail, setting and getting of its parameters, computation, visualization and removal.

Creation of the nail

This method of the interface creates a data tree for the nail at a given leaf of OCAF data tree.

It creates three sub-leaves for the cone and two cylinders and allocates the necessary data (references at the sub-leaves of the long and the head cylinders).

It sets the default values of position, radius and height of the nail.

The nail has the following user parameters:

- The position – coincides with the position of the cone
- The radius of the stem part of the nail – coincides with the radius of the cone
- The height of the nail – a sum of heights of the cone and both cylinders

The values of the position and the radius of the nail are defined for the cone object data. The height of the cone is recomputed as 2 * heights of nail and divided by 9.

Computation

The Function Mechanism is responsible for re-computation of the nail. It will be described in detail later in this document.

A data leaf consists of the reference to the location of the real data and a
real value defining a coefficient of multiplication of the referenced data.

For example, the height of the long cylinder is defined as a reference to the height of the cone with coefficient 3. The data leaf of the height of the long cylinder should contain two attributes: a reference to the height of cone and a real value equal to 3.

**Visualization**

The shape resulting of the nail function can be displayed using the standard OCAF visualization mechanism.

**Removal of the nail**

To automatically erase the nail from the viewer and the data tree it is enough to clean the nail leaf from attributes.
Step 3: Functions

The nail is defined by four functions: the cone, the two cylinders and the nail function. The function of the cone is independent. The functions of the cylinders depend on the cone function. The nail function depends on the results of all functions:

A graph of dependencies between functions

Computation of the model starts with the cone function, then the long cylinder, after that the header cylinder and, finally, the result is generated by the nail function at the end of function chain.

The Function Mechanism of Open CASCADE Technology creates this graph of dependencies and allows iterating it following the dependencies. The only thing the Function Mechanism requires from its user is the implementation of pure virtual methods of `TFunction_Driver`:

- `::Arguments()` – returns a list of arguments for the function
- `::Results()` – returns a list of results of the function

These methods give the Function Mechanism the information on the location of arguments and results of the function and allow building a graph of functions. The class `TFunction_Iterator` iterates the functions of the graph in the execution order.

The pure virtual method `TFunction_Driver::Execute()` calculating the function should be overridden.
The method ::MustExecute() calls the method ::Arguments() of the function driver and ideally this information (knowledge of modification of arguments of the function) is enough to make a decision whether the function should be executed or not. Therefore, this method usually shouldn’t be overridden.

The cone and cylinder functions differ only in geometrical construction algorithms. Other parameters are the same (position, radius and height).

It means that it is possible to create a base class – function driver for the three functions, and two descendant classes producing: a cone or a cylinder.

For the base function driver the methods ::Arguments() and ::Results() will be overridden. Two descendant function drivers responsible for creation of a cone and a cylinder will override only the method ::Execute().

The method ::Arguments() of the function driver of the nail returns the results of the functions located under it in the tree of leaves. The method ::Execute() just collects the results of the functions and makes one shape – a nail.

This way the data model using the Function Mechanism is ready for usage. Do not forget to introduce the function drivers for a function driver table with the help of TFunction_DriverTable class.
Example 1: iteration and execution of functions.

This is an example of the code for iteration and execution of functions.

```cpp
// The scope of functions is defined.
Handle(TFunction_Scope) scope = TFunction_Scope::Set(anyLabel);

// The information on modifications in the model is received.
TFunction_Logbook& log = scope-GetLogbook();

// The iterator is initialized by the scope of functions.
TFunction_Iterator iterator(anyLabel);
Iterator.SetUsageOfExecutionOrder(true);

// The function is iterated, its dependency is checked on the modified data and executed if necessary.
for (; iterator.more(); iterator.Next())
{
    // The function iterator may return a list of current functions for execution.
    // It might be useful for multi-threaded execution of functions.
    const TDF_LabelList& currentFunctions = iterator.Current();

    // The list of current functions is iterated.
    TDF_ListIteratorOfLabelList currentIterator(currentFunctions);
    for (; currentIterator.More(); currentIterator.Next())
```
{  
  An interface for the function is created. 
  TFunction_IFunction  interface( 
    currentIterator.Value());

  // The function driver is retrieved. 
  Handle(TFunction_Driver)  driver = 
    interface.GetDriver();

  // The dependency of the function on the modified 
  data is checked. 
  If  (driver-MustExecute( log ))
  {
    // The function is executed. 
    int  ret = driver-Execute( log );
    if ( ret )
      return  false;
  } // end if check on modification
} // end of iteration of current functions
} // end of iteration of functions.
Example 2: Cylinder function driver

This is an example of the code for a cylinder function driver. To make the things clearer, the methods ::Arguments() and ::Results() from the base class are also mentioned.

```cpp
// A virtual method ::Arguments() returns a list of arguments of the function.
CylinderDriver::Arguments( TDF_LabelList& args )
{
    // The direct arguments, located at sub-leaves of the function, are collected (see picture 2).
    TDF_ChildIterator cIterator( Label(), false );
    for (; cIterator.More(); cIterator.Next() )
    {
        // Direct argument.
        TDF_Label sublabel = cIterator.Value();
        Args.Append( sublabel );

        // The references to the external data are checked.
        Handle(TDF_Reference) ref;
        If ( sublabel.FindAttribute( TDF_Reference::GetID(), ref ) )
        {
            args.Append( ref.Get() );
        }
    }

    // A virtual method ::Results() returns a list of result leaves.
    CylinderDriver::Results( TDF_LabelList& res )
    {
        // The result is kept at the function label.
        Res.Append( Label() );
    }
```
// Execution of the function driver.
Int CylinderDriver::Execute( TFunction_Logbook& log )
{

    // Position of the cylinder - position of the first function (cone)
    // is elevated along Z for height values of all previous functions.
    gp_Ax2 axes = .... // out of the scope of this guide.
    // The radius value is retrieved.
    // It is located at second child sub-leaf (see the picture 2).
    TDF_Label radiusLabel = Label().FindChild( 2 );

    // The multiplicator of the radius ()is retrieved.
    Handle(TDataStd_Real) radiusValue;
    radiusLabel.FindAttribute( TDataStd_Real::GetID(), radiusValue);

    // The reference to the radius is retrieved.
    Handle(TDF_Reference) refRadius;
    RadiusLabel.FindAttribute( TDF_Reference::GetID(), refRadius );

    // The radius value is calculated.
    double radius = 0.0;

    if ( refRadius.IsNull() )
    {
        radius = radiusValue.Get();
    }
    else
    {
        // The referenced radius value is retrieved.
        Handle(TDataStd_Real) referencedRadiusValue;
        RefRadius.Get().FindAttribute(TDataStd<Real>::GetID()
radius = referencedRadiusValue-Get() * radiusValue-Get();

// The height value is retrieved.
double height = ... // similar code to taking the radius value.

// The cylinder is created.
TopoDS_Shape cylinder = BRepPrimAPI_MakeCylinder(axes, radius, height);

// The result (cylinder) is set
TNaming_Builder builder( Label() );
Builder.Generated( cylinder );

// The modification of the result leaf is saved in the log.
log.SetImpacted( Label() );

return 0;
XML Support

Writing and reading XML files in OCCT is provided by LDOM package, which constitutes an integral part of XML OCAF persistence, which is the optional component provided on top of Open CASCADE Technology.

The Light DOM (LDOM) package contains classes maintaining a data structure whose main principles conform to W3C DOM Level 1 Recommendations. The purpose of these classes as required by XML OCAF persistence schema is to:

- Maintain a tree structure of objects in memory representing the XML document. The root of the structure is an object of the `LDOM_Document` type. This object contains all the data corresponding to a given XML document and contains one object of the `LDOM_Element` type named "document element". The document element contains other `LDOM_Element` objects forming a tree. Other types of nodes: `LDOM_Attr`, `LDOM_Text`, `LDOM_Comment` and `LDOM_CDATASection` – represent the corresponding XML types and serve as branches of the tree of elements.
- Provide class `LDOM_Parser` to read XML files and convert them to `LDOM_Document` objects.
- Provide class `LDOM_XmlWriter` to convert `LDOM_Document` to a character stream in XML format and store it in file.

This package covers the functionality provided by numerous products known as "DOM parsers". Unlike most of them, LDOM was specifically developed to meet the following requirements:

- To minimize the virtual memory allocated by DOM data structures. In average, the amount of memory of LDOM is the same as the XML file size (UTF-8).
- To minimize the time required for parsing and formatting XML, as well as for access to DOM data structures.

Both these requirements are important when XML files are processed by applications if these files are relatively large (occupying megabytes and even hundreds of megabytes). To meet the requirements, some limitations were imposed on the DOM Level 1 specification; these
limitations are insignificant in applications like OCAF. Some of these limitations can be overridden in the course of future developments. The main limitations are:

- No Unicode support as well as various other encodings; only ASCII strings are used in DOM/XML. Note: There is a data type \textit{TCollection\_ExtendedString} for wide character data. This type is supported by \textit{LDOM\_String} as a sequence of numbers.
- Some superfluous methods are deleted: \textit{getPreviousSibling, getParentNode}, etc.
- No resolution of XML Entities of any kind
- No support for DTD: the parser just checks for observance of general XML rules and never validates documents.
- Only 5 available types of DOM nodes: \textit{LDOM\_Element, LDOM\_Attr, LDOM\_Text, LDOM\_Comment} and \textit{LDOM\_CDATASection}.
- No support of Namespaces; prefixed names are used instead of qualified names.
- No support of the interface \textit{DOMException} (no exception when attempting to remove a non-existing node).

LDOM is dependent on Kernel OCCT classes only. Therefore, it can be used outside OCAF persistence in various algorithms where DOM/XML support may be required.
Document Drivers

The drivers for document storage and retrieval manage conversion between a transient OCAF Document in memory and its persistent reflection in a container (disk, memory, network). For XML Persistence, they are defined in the package XmlDrivers.

The main methods (entry points) of these drivers are:

- `Write()` – for a storage driver;
- `Read()` – for a retrieval driver.

The most common case (which is implemented in XML Persistence) is writing/reading document to/from a regular OS file. Such conversion is performed in two steps:

First it is necessary to convert the transient document into another form (called persistent), suitable for writing into a file, and vice versa. In XML Persistence LDOM_Document is used as the persistent form of an OCAF Document and the DOM_Nodes are the persistent objects. An OCAF Document is a tree of labels with attributes. Its transformation into a persistent form can be functionally divided into two parts:

- Conversion of the labels structure, which is performed by the method `XmlMDF::FromTo()`
- Conversion of the attributes and their underlying objects, which is performed by the corresponding attribute drivers (one driver per attribute type).

The driver for each attribute is selected from a table of drivers, either by attribute type (on storage) or by the name of the corresponding DOM_Element (on retrieval). The table of drivers is created by by methods `XmlDrivers_DocumentStorageDriver::AttributeDrivers()` and `XmlDrivers_DocumentRetrievalDriver::AttributeDrivers()`.

Then the persistent document is written into a file (or read from a file). In standard persistence Storage and FSD packages contain classes for writing/reading the persistent document into a file. In XML persistence `LDOMParser` and `LDOM_XmlWriter` are used instead.
Usually, the library containing document storage and retrieval drivers is loaded at run time by a plugin mechanism. To support this in XML Persistence, there is a plugin `XmlPlugin` and a `Factory()` method in the `XmlDrivers` package. This method compares passed GUIDs with known GUIDs and returns the corresponding driver or generates an exception if the GUID is unknown.

The application defines which GUID is needed for document storage or retrieval and in which library it should be found. This depends on document format and application resources. Resources for XML Persistence and also for standard persistence are found in the `StdResource` unit. They are written for the XmlOcaf document format.
Attribute Drivers

There is one attribute driver for XML persistence for each transient attribute from a set of standard OCAF attributes, with the exception of attribute types, which are never stored (pure transient). Standard OCAF attributes are collected in six packages, and their drivers also follow this distribution. Driver for attribute $T^*_\ast$ is called $XmlM^*_\ast$. Conversion between transient and persistent form of attribute is performed by two methods $Paste()$ of attribute driver.

$XmlMDF_ADriver$ is the root class for all attribute drivers.

At the beginning of storage/retrieval process, one instance of each attribute driver is created and appended to driver table implemented as $XmlMDF_ADriverTable$. During OCAF Data storage, attribute drivers are retrieved from the driver table by the type of attribute. In the retrieval step, a data map is created linking names of $DOM\_Elements$ and attribute drivers, and then attribute drivers are sought in this map by $DOM\_Element$ qualified tag names.

Every transient attribute is saved as a $DOM\_Element$ (root element of OCAF attribute) with attributes and possibly sub-nodes. The name of the root element can be defined in the attribute driver as a string passed to the base class constructor. The default is the attribute type name. Similarly, namespace prefixes for each attribute can be set. There is no default value, but it is possible to pass NULL or an empty string to store attributes without namespace prefixes.

The basic class $XmlMDF_ADriver$ supports errors reporting via the method $WriteMessage(const\ TCollection\_ExtendedString&)$. It sends a message string to its message driver which is initialized in the constructor with a $Handle(CDM\_MessageDriver)$ passed from the application by Document Storage/Retrieval Driver.
XML Document Structure

Every XML Document has one root element, which may have attributes and contain other nodes. In OCAF XML Documents the root element is named "document" and has attribute "format" with the name of the OCAF Schema used to generate the file. The standard XML format is "XmlOcaf". The following elements are sub-elements of <document> and should be unique entries as its sub-elements, in a specific order. The order is:

- **Element info** – contains strings identifying the format version and other parameters of the OCAF XML document. Normally, data under the element is used by persistence algorithms to correctly retrieve and initialize an OCAF document. The data also includes a copyright string.
- **Element comments** – consists of an unlimited number of <comment> sub-elements containing necessary comment strings.
- **Element label** – the root label of the document data structure, with the XML attribute "tag" equal to 0. It contains all the OCAF data (labels, attributes) as tree of XML elements. Every sub-label is identified by a tag (positive integer) defining a unique key for all sub-labels of a label. Every label can contain any number of elements representing OCAF attributes (see OCAF Attributes Representation below).
- **Element shapes** – contains geometrical and topological entities in BRep format. These entities being referenced by OCAF attributes written under the element <label>. This element is empty if there are no shapes in the document. It is only output if attribute driver `XmlMNaming_NamedShapeDriver` has been added to drivers table by the DocumentStorageDriver.

OCAF Attributes Representation

In XML documents, OCAF attributes are elements whose name identifies the OCAF attribute type. These elements may have a simple (string or number) or complex (sub-elements) structure, depending on the architecture of OCAF attribute. Every XML type for OCAF attribute possesses a unique positive integer "id" XML attribute identifying the OCAF attribute throughout the document. To ensure "id" uniqueness, the
attribute name "id" is reserved and is only used to indicate and identify elements which may be referenced from other parts of the OCAF XML document. For every standard OCAF attribute, its XML name matches the name of a C++ class in Transient data model. Generally, the XML name of OCAF attribute can be specified in the corresponding attribute driver. XML types for OCAF attributes are declared with XML W3C Schema in a few XSD files where OCAF attributes are grouped by the package where they are defined.

**Example of resulting XML file**

The following example is a sample text from an XML file obtained by storing an OCAF document with two labels (0: and 0:2) and two attributes — TDataStd_Name (on label 0:) and TNaming_NamedShape (on label 0:2). The <shapes> section contents are replaced by an ellipsis.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<document format="XmlOcaf"
    xmlns="http://www.opencascade.org/OCAF/XML"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opencascade.org/OCAF/XML/XmlOcaf.xsd">
    
  <info date="2001-10-04" schemav="0" objnb="3">
    <iitem>Copyright: Open Cascade, 2001</iitem>
    <iitem>STORAGE_VERSION: PCDM_ReadWriter_1</iitem>
    <iitem>REFERENCE_COUNTER: 0</iitem>
    <iitem>MODIFICATION_COUNTER: 1</iitem>
  </info>
  
  <comments/>
  
  <label tag="0">
    <TDataStd_Name id="1">Document_1</TDataStd_Name>
  </label>
  
  <label tag="2">
    <TNaming_NamedShape id="2" evolution="primitive">
      <olds/>
    </TNaming_NamedShape>
  </label>
</document>
```
**XML Schema**

The XML Schema defines the class of a document.

The full structure of OCAF XML documents is described as a set of XML W3C Schema files with definitions of all XML element types. The definitions provided cannot be overridden. If any application defines new persistence schemas, it can use all the definitions from the present XSD files but if it creates new or redefines existing types, the definition must be done under other namespace(s).

There are other ways to declare XML data, different from W3C Schema, and it should be possible to use them to the extent of their capabilities of expressing the particular structure and constraints of our XML data model. However, it must be noted that the W3C Schema is the primary format for declarations and as such, it is the format supported for future improvements of Open CASCADE Technology, including the development of specific applications using OCAF XML persistence.

The Schema files (XSD) are intended for two purposes:

- documenting the data format of files generated by OCAF;
- validation of documents when they are used by external (non-OCAF) applications, e.g., to generate reports.

The Schema definitions are not used by OCAF XML Persistence algorithms when saving and restoring XML documents. There are internal checks to ensure validity when processing every type of data.

**Management of Namespaces**

Both the XML format and the XML OCAF persistence code are extensible in the sense that every new development can reuse everything that has been created in previous projects. For the XML format, this extensibility is supported by assigning names of XML objects (elements) to different XML Namespaces. Hence, XML elements defined in different projects (in different persistence libraries) can easily be combined into the same XML documents. An example is the XCAF XML persistence built as an extension to the Standard OCAF XML persistence [File XmlXcaf.xsd]. For
the correct management of Namespaces it is necessary to:

- Define *targetNamespace* in the new XSD file describing the format.
- Declare (in XSD files) all elements and types in the targetNamespace to appear without a namespace prefix; all other elements and types use the appropriate prefix (such as "ocaf:").
- Add (in the new *DocumentStorageDriver*) the *targetNamespace* accompanied with its prefix, using method *XmlDrivers_DocumentStorageDriver::AddNamespace*. The same is done for all namespaces objects which are used by the new persistence, with the exception of the "ocaf" namespace.
- Pass (in every OCAF attribute driver) the namespace prefix of the *targetNamespace* to the constructor of *XmlMDF_ADriver*. 
GLOSSARY

- **Application** – a document container holding all documents containing all application data.
- **Application data** – the data produced by an application, as opposed to data referring to it.
- **Associativity of data** – the ability to propagate modifications made to one document to other documents, which refer to such document. Modification propagation is:
  - unidirectional, that is, from the referenced to the referencing document(s), or
  - bi-directional, from the referencing to the referenced document and vice-versa.
- **Attribute** – a container for application data. An attribute is attached to a label in the hierarchy of the data framework.
- **Child** – a label created from another label, which by definition, is the father label.
- **Compound document** – a set of interdependent documents, linked to each other by means of external references. These references provide the associativity of data.
- **Data framework** – a tree-like data structure which in OCAF, is a tree of labels with data attached to them in the form of attributes. This tree of labels is accessible through the services of the `TDocStd_Document` class.
- **Document** – a container for a data framework which grants access to the data, and is, in its turn, contained by an application. A document also allows you to:
  - Manage modifications, providing Undo and Redo functions
  - Manage command transactions
  - Update external links
  - Manage save and restore options
  - Store the names of software extensions.
- **Driver** – an abstract class, which defines the communications protocol with a system.
- **Entry** – an ASCII character string containing the tag list of a label. For example:
  
  0:3:24:7:2:7

- **External links** – references from one data structure to another data
structure in another document. To store these references properly, a label must also contain an external link attribute.

- **Father** – a label, from which other labels have been created. The other labels are, by definition, the children of this label.
- **Framework** – a group of co-operating classes which enable a design to be re-used for a given category of problem. The framework guides the architecture of the application by breaking it up into abstract classes, each of which has different responsibilities and collaborates in a predefined way. Application developer creates a specialized framework by:
  - defining new classes which inherit from these abstract classes
  - composing framework class instances
  - implementing the services required by the framework.

In C++, the application behavior is implemented in virtual functions redefined in these derived classes. This is known as overriding.

- **GUID** – Global Universal ID. A string of 37 characters intended to uniquely identify an object. For example:
  
  2a96b602-ec8b-11d0-bee7-080009dc3333

- **Label** – a point in the data framework, which allows data to be attached to it by means of attributes. It has a name in the form of an entry, which identifies its place in the data framework.
- **Modified label** – containing attributes whose data has been modified.
- **Reference key** – an invariant reference, which may refer to any type of data used in an application. In its transient form, it is a label in the data framework, and the data is attached to it in the form of attributes. In its persistent form, it is an entry of the label. It allows an application to recover any entity in the current session or in a previous session.
- **Resource file** – a file containing a list of each document's schema name and the storage and retrieval plug-ins for that document.
- **Root** – the starting point of the data framework. This point is the top label in the framework. It is represented by the [0] entry and is created at the same time with the document you are working on.
- **Scope** – the set of all the attributes and labels which depend on a given label.
- **Tag list** – a list of integers, which identify the place of a label in the data framework. This list is displayed in an entry.
- **Topological naming** – systematic referencing of topological entities so that these entities can still be identified after the models they belong to have gone through several steps in modeling. In other words, topological naming allows you to track entities through the steps in the modeling process. This referencing is needed when a model is edited and regenerated, and can be seen as a mapping of labels and name attributes of the entities in the old version of a model to those of the corresponding entities in its new version. Note that if the topology of a model changes during the modeling, this mapping may not fully coincide. A Boolean operation, for example, may split edges.

- **Topological tracking** – following a topological entity in a model through the steps taken to edit and regenerate that model.

- **Valid label** – in a data framework, this is a label, which is already recomputed in the scope of regeneration sequence and includes the label containing a feature which is to be recalculated. Consider the case of a box to which you first add a fillet, then a protrusion feature. For recalculation purposes, only valid labels of each construction stage are used. In recalculating a fillet, they are only those of the box and the fillet, not the protrusion feature which was added afterwards.
Samples

Getting Started

At the beginning of your development, you first define an application class by inheriting from the Application abstract class. You only have to create and determine the resources of the application for specifying the format of your documents (you generally use the standard one) and their file extension.

Then, you design the application data model by organizing attributes you choose among those provided with OCAF. You can specialize these attributes using the User attribute. For example, if you need a reflection coefficient, you aggregate a User attribute identified as a reflection coefficient with a Real attribute containing the value of the coefficient (as such, you don't define a new class).

If you need application specific data not provided with OCAF, for example, to incorporate a finite element model in the data structure, you define a new attribute class containing the mesh, and you include its persistent homologue in a new file format.

Once you have implemented the commands which create and modify the data structure according to your specification, OCAF provides you, without any additional programming:

- Persistent reference to any data, including geometric elements — several documents can be linked with such reference;
- Document-View association;
- Ready-to-use functions such as:
  - Undo-redo;
  - Save and open application data.

Finally, you develop the application's graphical user interface using the toolkit of your choice, for example:

- KDE Qt or GNOME GTK+ on Linux;
- Microsoft Foundation Classes (MFC) on Windows Motif on Sun;
- Other commercial products such as Ilog Views.

You can also implement the user interface in the Java language using the Swing-based Java Application Desktop component (JAD) provided with OCAF.
Implementation of Attribute Transformation in a HXX file

```cpp
#include <TDF_Attribute.hxx>
#include <gp_Ax3.hxx>
#include <gp_Pnt.hxx>
#include <gp_Vec.hxx>
#include <gp_Trsf.hxx>

// This attribute implements a transformation data container
class MyPackage_Transformation : public TDF_Attribute
{
public:
    //!@ name Static methods
    
    //! The method returns a unique GUID of this attribute.
    //! By means of this GUID this attribute may be identified
    //! among other attributes attached to the same label.
    Standard_EXPORT static const Standard_GUID& GetID () const;

    //! Finds or creates the attribute attached to <theLabel>.
    //! The found or created attribute is returned.
    Standard_EXPORT static Handle(MyPackage_Transformation) Set (const TDF_Label theLabel);

    //!@ name Methods for access to the attribute data
```
//@ name Methods for setting the data of transformation

/// The method defines a rotation type of transformation.
Standard_EXPORT void SetRotation (const gp_Ax1& theAxis, Standard_Real theAngle);

/// The method defines a translation type of transformation.
Standard_EXPORT void SetTranslation (const gp_Vec& theVector);

/// The method defines a point mirror type of transformation (point symmetry).
Standard_EXPORT void SetMirror (const gp_Pnt& thePoint);

/// The method defines an axis mirror type of transformation (axial symmetry).
Standard_EXPORT void SetMirror (const gp_Ax1& theAxis);

/// The method defines a point mirror type of transformation (planar symmetry).
Standard.Export void SetMirror (const gp_Ax2& thePlane);

/// The method defines a scale type of transformation.
Standard_EXPORT void SetScale (const gp_Pnt& thePoint, Standard_Real theScale);

/// The method defines a complex type of
transformation from one co-ordinate system to another.

Standard_EXPORT void SetTransformation (const gp_Ax3& theCoordinateSystem1, const gp_Ax3& theCoordinateSystem2);

//@ name Overridden methods from TDF_Attribute

//@ The method returns a unique GUID of the attribute.
//@ By means of this GUID this attribute may be identified among other attributes attached to the same label.
Standard_EXPORT const Standard_GUID& ID () const;

//@ The method is called on Undo / Redo.
//@ It copies the content of theAttribute into this attribute (copies the fields).
Standard_EXPORT void Restore (const Handle(TDF_Attribute)& theAttribute);

//@ It creates a new instance of this attribute.
//@ It is called on Copy / Paste, Undo / Redo.
Standard_EXPORT Handle(TDF_Attribute) NewEmpty () const;

//@ The method is called on Copy / Paste.
//@ It copies the content of this attribute into theAttribute (copies the fields).
Standard_EXPORT void Paste (const Handle(TDF_Attribute)& theAttribute, const Handle(TDF_RelocationTable)& theRelocationTable);

//@ Prints the content of this attribute into the stream.
Standard_EXPORT Standard_OStream&
Dump(Standard_OStream& theOS);

//@ name Constructor
//@
//@ The C++ constructor of this attribute class.
//@ Usually it is never called outside this class.
Standard_EXPORT MyPackage_Transformation();

private:
gp_TrsfForm myType;

// Axes (Ax1, Ax2, Ax3)
gp_Ax1 myAx1;
gp_Ax2 myAx2;
gp_Ax3 myFirstAx3;
gp_Ax3 mySecondAx3;

// Scalar values
Standard_Real myAngle;
Standard_Real myScale;

// Points
gp_Pnt myFirstPoint;
gp_Pnt mySecondPoint;

};
Implementation of Attribute Transformation in a CPP file

#include <MyPackage_Transformation.hxx>

//=************************************************************************=
//function: GetID
//purpose:  The method returns a unique GUID of this attribute.
//          By means of this GUID this attribute may be identified
//          among other attributes attached to the same label.
//=************************************************************************=
const Standard_GUID&
MyPackage_Transformation::GetID()
{
    static Standard_GUID ID("4443368E-C808-4468-984D-B26906BA8573");
    return ID;
}

//=************************************************************************=
//function: Set
//purpose:  Finds or creates the attribute attached to <theLabel>.
//          The found or created attribute is returned.
//=************************************************************************=
Handle(MyPackage_Transformation)
MyPackage_Transformation::Set(const TDF_Label&


```cpp
theLabel)
{
    Handle(MyPackage_Transformation) T;
    if (!theLabel.FindAttribute(MyPackage_Transformation::GetID(), T))
    {
        T = new MyPackage_Transformation();
        theLabel.AddAttribute(T);
    }
    return T;
}

//===================================================
//function : Get
//purpose : The method returns the transformation.
//===================================================
gp_Trsf MyPackage_Transformation::Get() const
{
    gp_Trsf transformation;
    switch (myType)
    {
    case gp_Identity:
        break;
    case gp_Rotation:
        transformation.SetRotation(myAx1, myAngle);
        break;
    case gp_Translation:
        transformation.SetTranslation(myFirstPoint, mySecondPoint);
    ```
break;
}
case gp_PntMirror:
{
    transformation.SetMirror(myFirstPoint);
    break;
}
case gp_Ax1Mirror:
{
    transformation.SetMirror(myAx1);
    break;
}
case gp_Ax2Mirror:
{
    transformation.SetMirror(myAx2);
    break;
}
case gp_Scale:
{
    transformation.SetScale(myFirstPoint, myScale);
    break;
}
case gp_CompoundTrsf:
{
    transformation.SetTransformation(myFirstAx3,
        mySecondAx3);
    break;
}
case gp_Other:
{
    break;
}
}
return transformation;
}
void MyPackage_Transformation::SetRotation(const gp_Ax1& theAxis, const Standard_Real theAngle)
{
    Backup();
    myType = gp_Rotation;
    myAx1 = theAxis;
    myAngle = theAngle;
}

void MyPackage_Transformation::SetTranslation(const gp_Vec& theVector)
{
    Backup();
    myType = gp_Translation;
    myFirstPoint.SetCoord(0, 0, 0);
    mySecondPoint.SetCoord(theVector.X(),
                            theVector.Y(), theVector.Z());
}
void MyPackage_Transformation::SetMirror(const gp_Pnt& thePoint)
{
    Backup();
    myType = gp_PntMirror;
    myFirstPoint = thePoint;
}

void MyPackage_Transformation::SetMirror(const gp_Ax1& theAxis)
{
    Backup();
    myType = gp_Ax1Mirror;
    myAx1 = theAxis;
}

void MyPackage_Transformation::SetMirror(const gp_Ax2& thePlane)
{ Backup();
  myType = gp_Ax2Mirror;
  myAx2 = thePlane;
}

//===================================================
//===================================================
//function : SetScale
//purpose : The method defines a scale type of
//          transformation.
//===================================================
//===================================================
void MyPackage_Transformation::SetScale(const gp_Pnt&
                               thePoint, const Standard_Real theScale)
{
  Backup();
  myType = gp_Scale;
  myFirstPoint = thePoint;
  myScale = theScale;
}

//===================================================
//===================================================
//function : SetTransformation
//purpose : The method defines a complex type of
//          transformation from one co-ordinate system to another.
//===================================================
//===================================================
void
    MyPackage_Transformation::SetTransformation(const gp_Ax3&
                                theCoordinateSystem1,
                                const gp_Ax3&
                                theCoordinateSystem2)
{
  Backup();
  myFirstAx3 = theCoordinateSystem1;

mySecondAx3 = theCoordinateSystem2;
}

//=================================================================================
//function : ID
//purpose  : The method returns a unique GUID of the attribute.
//           By means of this GUID this attribute may be identified
//           among other attributes attached to the same label.
//=================================================================================
const Standard_GUID& MyPackage_Transformation::ID() const
{
    return GetID();
}

//=================================================================================
//function : Restore
//purpose  : The method is called on Undo / Redo.
//           It copies the content of <theAttribute> into this attribute (copies the fields).
//=================================================================================
void MyPackage_Transformation::Restore(const Handle(TDF_Attribute)& theAttribute)
{
    Handle(MyPackage_Transformation) theTransformation =
        Handle(MyPackage_Transformation)::DownCast(theAttribute);
    myType = theTransformation->myType;
    myAx1 = theTransformation->myAx1;
myAx2 = theTransformation->myAx2;
myFirstAx3 = theTransformation->myFirstAx3;
mySecondAx3 = theTransformation->mySecondAx3;
myAngle = theTransformation->myAngle;
myScale = theTransformation->myScale;
myFirstPoint = theTransformation->myFirstPoint;
mySecondPoint = theTransformation->mySecondPoint;
}

//===================================================
//===================================================

//function : NewEmpty
//purpose : It creates a new instance of this attribute.
//          It is called on Copy / Paste, Undo / Redo.
//===================================================
Handle(TDF_Attribute) MyPackage_Transformation::NewEmpty() const
{
    return new MyPackage_Transformation();
}

//===================================================
//===================================================

//function : Paste
//purpose : The method is called on Copy / Paste. It copies the content of this attribute into
//          <theAttribute> (copies the fields).
//===================================================
void MyPackage_Transformation::Paste(const Handle(TDF_Attribute)& theAttribute,
                                   const Handle(TDF_RelocationTable)& ) const
{
Handle(MyPackage_Transformation) theTransformation =
   Handle(MyPackage_Transformation)::DownCast(theAttribute);
theTransformation->myType = myType;
theTransformation->myAx1 = myAx1;
theTransformation->myAx2 = myAx2;
theTransformation->myFirstAx3 = myFirstAx3;
theTransformation->mySecondAx3 = mySecondAx3;
theTransformation->myAngle = myAngle;
theTransformation->myScale = myScale;
theTransformation->myFirstPoint = myFirstPoint;
theTransformation->mySecondPoint = mySecondPoint;
}

/*==========================================================================
//function : Dump
//purpose : Prints the content of this attribute into the stream.
/*==========================================================================
Standard_OStream&
   MyPackage_Transformation::Dump(Standard_OStream& anOS) const
{
   anOS = "Transformation: ";
   switch (myType)
   {
   case gp_Identity:
   { 
      anOS = "gp_Identity";
      break;
   }
   case gp_Rotation:
   { 
      anOS = "gp_Rotation";
      break;
   }
   case gp_Scale:
   { 
      anOS = "gp_Scale";
      break;
   }
break;
}
case gp_Translation:
{
    anOS = "gp_Translation"
break;
}
case gp_PntMirror:
{
    anOS = "gp_PntMirror"
break;
}
case gp_Ax1Mirror:
{
    anOS = "gp_Ax1Mirror"
break;
}
case gp_Ax2Mirror:
{
    anOS = "gp_Ax2Mirror"
break;
}
case gp_Scale:
{
    anOS = "gp_Scale"
break;
}
case gp_CompoundTrsf:
{
    anOS = "gp_CompoundTrsf"
break;
}
case gp_Other:
{
    anOS = "gp_Other"
break;
}
return anOS;

MyPackage_Transformation::MyPackage_Transformation(): myType(gp_Identity)
Implementation of typical actions with standard OCAF attributes.

There are four sample files provided in the directory 'OpenCasCade/ros/samples/ocafsamples'. They present typical actions with OCAF services (mainly for newcomers). The method Sample() of each file is not dedicated for execution 'as is', it is rather a set of logical actions using some OCAF services.

TDataStd_Sample.cxx

This sample contains templates for typical actions with the following standard OCAF attributes:

- Starting with data framework;
- TDataStd_Integer attribute management;
- TDataStd_RealArray attribute management;
- TDataStd_Comment attribute management;
- TDataStd_Name attribute management;
- TDataStd_UAttribute attribute management;
- TDF_Reference attribute management;
- TDataXtd_Point attribute management;
- TDataXtd_Plane attribute management;
- TDataXtd_Axis attribute management;
- TDataXtd_Geometry attribute management;
- TDataXtd_Constraint attribute management;
- TDataStd_Directory attribute management;
- TDataStd_TreeNode attribute management.

TDocStd_Sample.cxx

This sample contains template for the following typical actions:

- creating application;
- creating the new document (document contains a framework);
- retrieving the document from a label of its framework;
- filling a document with data;
- saving a document in the file;
• closing a document;
• opening the document stored in the file;
• copying content of a document to another document with possibility to update the copy in the future.

**TPrsStd_Sample.cxx**

This sample contains template for the following typical actions:

• starting with data framework;
• setting the TPrsStd_AISViewer in the framework;
• initialization of aViewer;
• finding TPrsStd_AISViewer attribute in the DataFramework;
• getting AIS_InteractiveContext from TPrsStd_AISViewer;
• adding driver to the map of drivers;
• getting driver from the map of drivers;
• setting TNaming_NamedShape to <ShapeLabel>;
• setting the new TPrsStd_AISPresentation to <ShapeLabel>;
• displaying;
• erasing;
• updating and displaying presentation of the attribute to be displayed;
• setting a color to the displayed attribute;
• getting transparency of the displayed attribute;
• modify attribute;
• updating presentation of the attribute in viewer.

**TNaming_Sample.cxx**

This sample contains template for typical actions with OCAF Topological Naming services. The following scenario is used:

• data framework initialization;
• creating Box1 and pushing it as PRIMITIVE in DF;
• creating Box2 and pushing it as PRIMITIVE in DF;
• moving Box2 (applying a transformation);
• pushing the selected edges of the top face of Box1 in DF;
• creating a Fillet (using the selected edges) and pushing the result as a modification of Box1;
• creating a Cut (Box1, Box2) as a modification of Box1 and push it in DF;
• recovering the result from DF.
TObj Package

Table of Contents

- Introduction
  - Applicability
- TObj Model
  - TObj Model structure
  - Data Model basic features
  - Model Persistence
  - Access to the objects in the model
  - Own model data
  - Object naming
  - API for transaction mechanism
  - Model format and version
  - Model update
  - Model copying
  - Messaging
- Model object
  - Separation of data and interface
  - Basic features
  - Data layout and inheritance
  - Persistence
  - Names of objects
  - References between objects
- Creation and deletion of objects
- Transformation and replication of object data
- Object flags
- Partitions
- Auxiliary classes
- Packaging
Introduction

This document describes the package TObj, which is an add-on to the Open CASCADE Application Framework (OCAF).

This package provides a set of classes and auxiliary tools facilitating the creation of object-oriented data models on top of low-level OCAF data structures. This includes:

- Definition of classes representing data objects. Data objects store their data using primitive OCAF attributes, taking advantage of OCAF mechanisms for Undo/Redo and persistence. At the same time they provide a higher level abstraction over the pure OCAF document structure (labels / attributes).
- Organization of the data model as a hierarchical (tree-like) structure of objects.
- Support of cross-references between objects within one model or among different models. In case of cross-model references the models should depend hierarchically.
- Persistence mechanism for storing TObj objects in OCAF files, which allows storing and retrieving objects of derived types without writing additional code to support persistence.

This document describes basic principles of logical and physical organization of TObj-based data models and typical approaches to implementation of classes representing model objects.
Applicability

The main purpose of the TObj data model is rapid development of the object-oriented data models for applications, using the existing functionality provided by OCAF (Undo/Redo and persistence) without the necessity to redevelop such functionality from scratch.

As opposed to using bare OCAF (at the level of labels and attributes), TObj facilitates dealing with higher level abstracts, which are closer to the application domain. It works best when the application data are naturally organized in hierarchical structures, and is especially useful for complex data models with dependencies between objects belonging to different parts of the model.

It should be noted that TObj is efficient for representing data structures containing a limited number of objects at each level of the data structure (typically less than 1000). A greater number of objects causes performance problems due to list-based organization of OCAF documents. Therefore, other methods of storage, such as arrays, are advisable for data models or their sub-parts containing a great number of uniform objects. However, these methods can be combined with the usage of TObj to represent the high-level structure of the model.
**TObj Model**

**TObj Model structure**

In the *TObj* data model the data are separated from the interfaces that manage them.

It should be emphasized that *TObj* package defines only the interfaces and the basic structure of the model and objects, while the actual contents and structure of the model of a particular application are defined by its specific classes inherited from *TObj* classes. The implementation can add its own features or even change the default behaviour and the data layout, though this is not recommended.

Logically the *TObj* data model is represented as a tree of model objects, with upper-level objects typically being collections of other objects (called *partitions*, represented by the class *TObj_Particle*). The root object of the model is called the *Main partition* and is maintained by the model itself. This partition contains a list of sub-objects called its *children* each sub-object may contain its own children (according to its type), etc.
As the TObj Data Model is based on OCAF (Open CASCADE Application Framework) technology, it stores its data in the underlying OCAF document. The OCAF document consists of a tree of items called labels. Each label has some data attached to it in the form of attributes, and may contain an arbitrary number of sub-labels. Each sub-label is identified by its sequential number called the tag. The complete sequence of tag numbers of the label and its parents starting from the document root constitutes the complete entry of the label, which uniquely identifies its position in the document.

Generally the structure of the OCAF tree of the TObj data model corresponds to the logical structure of the model and can be presented as in the following picture:
All data of the model are stored in the root label (0:1) of the OCAF document. An attribute `<Obj_TModel>` is located in this root label. It stores the object of type `<Obj_Model>`. This object serves as a main interface tool to access all data and functionalities of the data model.

In simple cases all data needed by the application may be contained in a single data model. Moreover, `<Obj>` gives the possibility to distribute the data between several interconnected data models. This can be especially useful for the applications dealing with great amounts of data.
only the data required for the current operation is loaded in the memory at one time. It is presumed that the models have a hierarchical (tree-like) structure, where the objects of the child models can refer to the objects of the parent models, not vice-versa. Provided that the correct order of loading and closing of the models is ensured, the $TObj$ classes will maintain references between the objects automatically.
Data Model basic features

The class TObj_Model describing the data model provides the following functionalities:

- Loading and saving of the model from or in a file (methods Load and Save)
- Closing and removal of the model from memory (method Close)
- Definition of the full file name of the persistence storage for this model (method GetFile)
- Tools to organize data objects in partitions and iterate on objects (methods GetObjects, GetMainPartition, GetChildren, getPartition, getElementPartition)
- Mechanism to give unique names to model objects
- Copy (clone) of the model (methods NewEmpty and Paste)
- Support of earlier model formats for proper conversion of a model loaded from a file written by a previous version of the application (methods GetFormatVersion and SetFormatVersion)
- Interface to check and update the model if necessary (method Update)
- Support of several data models in one application. For this feature use OCAF multi-transaction manager, unique names and GUIDs of the data model (methods GetModelName, GetGUID)
**Model Persistence**

The persistent representation of any OCAF model is contained in an XML or a binary file, which is defined by the format string returned by the method *GetFormat*. The default implementation works with a binary OCAF document format (*BinOcaf*). The other available format is *XmlOcaf*. The class *TObj_Model* declares and provides a default implementation of two virtual methods:

```cpp
virtual Standard_Boolean Load (const char* theFile);
virtual Standard_Boolean SaveAs (const char* theFile);
```

which retrieve and store the model from or in the OCAF file. The descendants should define the following protected method to support Load and Save operations:

```cpp
virtual Standard_Boolean initNewModel (const Standard_Boolean IsNew);
```

This method is called by *Load* after creation of a new model or after its loading from the file; its purpose is to perform the necessary initialization of the model (such as creation of necessary top-level partitions, model update due to version changes etc.). Note that if the specified file does not exist, method *Load* will create a new document and call *initNewModel* with the argument *True*. If the file has been normally loaded, the argument *False* is passed. Thus, a new empty *TObj* model is created by calling *Load* with an empty string or the path to a nonexistent file as argument.

The method *Load* returns *True* if the model has been retrieved successfully (or created a new), or *False* if the model could not be loaded. If no errors have been detected during initialization (model retrieval or creation), the virtual method *AfterRetrieval* is invoked for all objects of the model. This method initializes or updates the objects immediately after the model initialization. It could be useful when some object data should be imported from an OCAF attribute into transient fields which could be changed outside of the OCAF transaction mechanism. Such fields can be stored into OCAF attributes for saving
into persistent storage during the save operation.

To avoid memory leaks, the `TObj_Model` class destructor invokes `Close` method which clears the OCAF document and removes all data from memory before the model is destroyed.

For XML and binary persistence of the `TObj` data model the corresponding drivers are implemented in `BinLDrivers`, `BinMObj` and `XmlLDrivers`, `XmlMObj` packages. These packages contain retrieval and storage drivers for the model, model objects and custom attributes from the `TObj` package. The schemas support persistence for the standard OCAF and `TObj` attributes. This is sufficient for the implementation of simple data models, but in some cases it can be reasonable to add specific OCAF attributes to facilitate the storage of the data specific to the application. In this case the schema should be extended using the standard OCAF mechanism.
Access to the objects in the model

All objects in the model are stored in the main partition and accessed by iterators. To access all model objects use:

```cpp
virtual Handle(TObj_ObjectIterator) GetObjects () const;
```

This method returns a recursive iterator on all objects stored in the model.

```cpp
virtual Handle(TObj_ObjectIterator) GetChildren () const;
```

This method returns an iterator on child objects of the main partition. Use the following method to get the main partition:

```cpp
Handle(TObj_Partition) GetMainPartition() const;
```

To receive the iterator on objects of a specific type \texttt{ATYPE} use the following call:

```cpp
GetMainPartition() -
  &gt;GetChildren(STANDARD_TYPE(ATYPE) );
```

The set of protected methods is provided for descendant classes to deal with partitions:

```cpp
virtual Handle(TObj_Partition) getPartition (const TDF_Label, const Standard_Boolean theHidden) const;
```

This method returns (creating if necessary) a partition in the specified label of the document. The partition can be created as hidden (\texttt{TObj_HIDDENPartition} class). A hidden partition can be useful to distinguish the data that should not be visible to the user when browsing the model in the application.
The following two methods allow getting (creating) a partition in the sub-label of the specified label in the document (the label of the main partition for the second method) and with the given name:

```cpp
virtual Handle(TObj_Partition) getPartition (const TDF_Label, const Standard_Integer theIndex,
const TCollection_ExtendedString& theName, const Standard_Boolean theHidden) const;
virtual Handle(TObj_Partition) getPartition (const Standard_Integer theIndex,
const TCollection_ExtendedString& theName, const Standard_Boolean theHidden) const;
```

If the default object naming and the name register mechanism is turned on, the object can be found in the model by its unique name:

```cpp
Handle(TObj_Object) FindObject (const Handle(TCollection_HExtendedString)& theName,
const Handle(TObj_TNameContainer)& theDictionary) const;
```
Own model data

The model object can store its own data in the Data label of its main partition, however, there is no standard API for setting and getting these data types. The descendants can add their own data using standard OCAF methods. The enumeration DataTag is defined in TObj_Model to avoid conflict of data labels used by this class and its descendants, similarly to objects (see below).
Object naming

The basic implementation of TObj_Model provides the default naming mechanism: all objects must have unique names, which are registered automatically in the data model dictionary. The dictionary is a TObj_TNameContainer attribute whose instance is located in the model root label. If necessary, the developer can add several dictionaries into the specific partitions, providing the name registration in the correct name dictionary and restoring the name map after document is loaded from file. To ignore name registering it is necessary to redefine the methodsSetName, AfterRetrieval of the TObj_Object class and skip the registration of the object name. Use the following methods for the naming mechanism:

```cpp
Standard_Boolean IsRegisteredName (const Handle(TCollection_HExtendedString)& theName, const Handle(TObj_TNameContainer)& theDictionary ) const;
```

Returns True if the object name is already registered in the indicated (or model) dictionary.

```cpp
void RegisterName (const Handle(TCollection_HExtendedString)& theName, const TDF_Label& theLabel, const Handle(TObj_TNameContainer)& theDictionary ) const;
```

Registers the object name with the indicated label where the object is located in the OCAF document. Note that the default implementation of the methodSetName of the object registers the new name automatically (if the name is not yet registered for any other object)

```cpp
void UnRegisterName (const Handle(TCollection_HExtendedString)& theName, const Handle(TObj_TNameContainer)& theDictionary ) const;
```
Unregisters the name from the dictionary. The names of \textit{TObj} model objects are removed from the dictionary when the objects are deleted from the model.

\begin{verbatim}
Handle(TObj_TNameContainer) GetDictionary() const;
\end{verbatim}

Returns a default instance of the model dictionary (located at the model root label). The default implementation works only with one dictionary. If there are a necessity to have more than one dictionary for the model objects, it is recommended to redefine the corresponding virtual method of \textit{TObj}\_Object that returns the dictionary where names of objects should be registered.
API for transaction mechanism

Class TObj_Model provides the API for transaction mechanism (supported by OCAF):

```
Standard_Boolean HasOpenCommand() const;
```

Returns True if a Command transaction is open

```
void OpenCommand() const;
```

Opens a new command transaction.

```
void CommitCommand() const;
```

Commits the Command transaction. Does nothing If there is no open Command transaction.

```
void AbortCommand() const;
```

Aborts the Command transaction. Does nothing if there is no open Command transaction.

```
Standard_Boolean IsModified() const;
```

Returns True if the model document has a modified status (has changes after the last save)

```
void SetModified( const Standard_Boolean );
```

Changes the modified status by force. For synchronization of transactions within several TObj_Model documents use class TDocStd_MultiTransactionManager.
Model format and version

Class *TObj_Model* provides the descendant classes with a means to control the format of the persistent file by choosing the schema used to store or retrieve operations.

```cpp
virtual TCollection_ExtendedString GetFormat () const;
```

Returns the string *TObjBin* or *TObjXml* indicating the current persistent mechanism. The default value is *TObjBin*. Due to the evolution of functionality of the developed application, the contents and the structure of its data model vary from version to version. *TObj* package provides a basic mechanism supporting backward versions compatibility, which means that newer versions of the application will be able to read Data Model files created by previous versions (but not vice-versa) with a minimum loss of data. For each type of Data Model, all known versions of the data format should be enumerated in increasing order, incremented with every change of the model format. The current version of the model format is stored in the model file and can be checked upon retrieval.

```cpp
Standard_Integer GetFormatVersion () const;
```

Returns the format version stored in the model file

```cpp
void SetFormatVersion (const Standard_Integer theVersion);
```

Defines the format version used for save.

Upon loading a model, the method *initNewModel()*, called immediately after opening a model from disk (on the level of the OCAF document), provides a specific code that checks the format version stored in that model. If it is older than the current version of the application, the data update can be performed. Each model can have its own specific conversion code that performs the necessary data conversion to make them compliant with the current version.

When the conversion ends the user is advised of that by the messenger
interface provided by the model (see messaging chapter for more details), and the model version is updated. If the version of data model is not supported (it is newer than the current or too old), the load operation should fail. The program updating the model after version change can be implemented as static methods directly in C++ files of the corresponding Data Model classes, not exposing it to the other parts of the application. These codes can use direct access to the model and objects data (attributes) not using objects interfaces, because the data model API and object classes could have already been changed.

Note that this mechanism has been designed to maintain version compatibility for the changes of data stored in the model, not for the changes of low-level format of data files (such as the storage format of a specific OCAF attribute). If the format of data files changes, a specific treatment on a case-by-case basis will be required.
Model update

The following methods are used for model update to ensure its consistency with respect to the other models in case of cross-model dependencies:

```cpp
virtual Standard_Boolean Update();
```

This method is usually called after loading of the model. The default implementation does nothing and returns True.

```cpp
virtual Standard_Boolean initNewModel( const Standard_Boolean IsNew);
```

This method performs model initialization, check and updates (as described above).

```cpp
virtual void updateBackReferences( const Handle(TObj_Object)& theObj);
```

This method is called from the previous method to update back references of the indicated object after the retrieval of the model from file (see data model - object relationship chapter for more details)
Model copying

To copy the model between OCAF documents use the following methods:

```cpp
virtual Standard_Boolean Paste (Handle(TObj_Model) theModel, Handle(TDF_RelocationTable) theRelocTable = 0 );
```

Pastes the current model to the new model. The relocation table ensures correct copying of the sub-data shared by several parts of the model. It stores a map of processed original objects of relevant types in their copies.

```cpp
virtual Handle(TObj_Model) NewEmpty() = 0;
```

Redefines a pure virtual method to create a new empty instance of the model.

```cpp
void CopyReferences ( const Handle(TObj_Model)& theTarget, const Handle(TDF_RelocationTable)& theRelocTable);
```

Copies the references from the current model to the target model.
Messaging

The messaging is organised using Open CASCADE Messenger from the package Message. The messenger is stored as the field of the model instance and can be set and retrieved by the following methods:

```c++
void SetMessenger( const Handle(Message_Messenger)& );
Handle(Message_Messenger) Messenger() const;
```

A developer should create his own instance of the Messenger bound to the application user interface, and attribute it to the model for future usage. In particular the messenger is used for reporting errors and warnings in the persistence mechanism. Each message has a unique string identifier (key). All message keys are stored in a special resource file TObj.msg. This file should be loaded at the start of the application by call to the appropriate method of the class Message_MsgFile.
Model object

Class *TObj_Object* provides basic interface and default implementation of important features of *TObj* model objects. This implementation defines basic approaches that are recommended for all descendants, and provides tools to facilitate their usage.
Separation of data and interface

In the TObj data model, the data are separated from the interfaces that manage them. The data belonging to a model object are stored in its root label and sub-labels in the form of standard OCAF attributes. This allows using standard OCAF mechanisms for work with these data, and eases the implementation of the persistence mechanism.

The instance of the interface which serves as an API for managing object data (e.g. represents the model object) is stored in the root label of the object, and typically does not bring its own data. The interface classes are organized in a hierarchy corresponding to the natural hierarchy of the model objects according to the application.

In the text below the term 'object' is used to denote either the instance of the interface class or the object itself (both interface and data stored in OCAF).

The special type of attribute TObj_TObject is used for storing instances of objects interfaces in the OCAF tree. TObj_TObject is a simple container for the object of type TObj_Object. All objects (interfaces) of the data model inherit this class.
Basic features

The *TObj_Object* class provides some basic features that can be inherited (or, if necessary, redefined) by the descendants:

- Gives access to the model to which the object belongs (method *GetModel*) and to the OCAF label in which the object is stored (method *GetLabel*).
- Supports references (and back references) to other objects in the same or in another model (methods *getReference*, *setReference*, *addReference*, *GetReferences*, *GetBackReferences*, *AddBackReference*, *RemoveBackReference*, *ReplaceReference*)
- Provides the ability to contain child objects, as it is actual for partition objects (methods *GetChildren*, *GetFatherObject*)
- Organizes its data in the OCAF structure by separating the sub-labels of the main label intended for various kinds of data and providing tools to organize these data (see below). The kinds of data stored separately are:
  - Child objects stored in the label returned by the method *GetChildLabel*
  - References to other objects stored in the label returned by the method *GetReferenceLabel*
  - Other data, both common to all objects and specific for each subtype of the model object, are stored in the label returned by the method *GetDataLabel*
- Provides unique names of all objects in the model (methods *GetDictionary*, *GetName*, *SetName*)
- Provides unified means to maintain persistence (implemented in descendants with the help of macros *DECLARE_TOBJOCAF_PERSISTENCE* and *IMPLEMENT_TOBJOCAF_PERSISTENCE*)
- Allows an object to remove itself from the OCAF document and check the depending objects can be deleted according to the back references (method *Detach*)
- Implements methods for identification and versioning of objects
- Manages the object interaction with OCAF Undo/Redo mechanism (method *IsAlive*, *AfterRetrieval*, *BeforeStoring*)
- Allows make a clone (methods *Clone*, *CopyReferences*, *CopyChildren*, *copyData*)
• Contains additional word of bit flags (methods GetFlags, SetFlags, TestFlags, ClearFlags)
• Defines the interface to sort the objects by rank (methods GetOrder, SetOrder)
• Provides a number of auxiliary methods for descendants to set/get the standard attribute values, such as int, double, string, arrays etc.

An object can be received from the model by the following methods:

```cpp
static Standard_Boolean GetObj ( const TDF_Label& theLabel, Handle(TObj_Object)& theResObject,
                                   const Standard_Boolean isSuper = Standard_False );
```

Returns True if the object has been found in the indicated label (or in the upper level label if isSuper is True).

```cpp
Handle(TObj_Object) GetFatherObject ( const Handle(Standard_Type)& theType = NULL ) const;
```

Returns the father object of the indicated type for the current object (the direct father object if the type is NULL).
Data layout and inheritance

As far as the data objects are separated from the interfaces and stored in the OCAF tree, the functionality to support inheritance is required. Each object has its own data and references stored in the labels in the OCAF tree. All data are stored in the sub-tree of the main object label. If it is necessary to inherit a class from the base class, the descendant class should use different labels for data and references than its ancestor.

Therefore each TObj class can reserve the range of tags in each of Data, References, and Child sub-labels. The reserved range is declared by the enumeration defined in the class scope (called DataTag, RefTag, and ChildTag, respectively). The item First of the enumeration of each type is defined via the Last item of the corresponding enumeration of the parent class, thus ensuring that the tag numbers do not overlap. The item Last of the enumeration defines the last tag reserved by this class. Other items of the enumeration define the tags used for storing particular data items of the object. See the declaration of the TObj_Partition class for the example.

TObj_Object class provides a set of auxiliary methods for descendants to access the data stored in sub-labels by their tag numbers:

```cpp
TDF_Label getDataLabel (const Standard_Integer theRank1, const Standard_Integer theRank2 = 0) const;
TDF_Label getReferenceLabel (const Standard_Integer theRank1, const Standard_Integer theRank2 = 0) const;
```

Returns the label in Data or References sub-labels at a given tag number (theRank1). The second argument, theRank2, allows accessing the next level of hierarchy (theRank2-th sub-label of theRank1-th data label). This is useful when the data to be stored are represented by multiple OCAF attributes of the same type (e.g. sequences of homogeneous data or references).

The get/set methods allow easily accessing the data located in the specified data label for the most widely used data types (Standard_Real,
Standard_Integer, TCollection_HExtendedString, TColStd_HArray1OfReal, TColStd_HArray1OfInteger, TColStd_HArray1OfExtendedString). For instance, methods provided for real numbers are:

```cpp
Standard_Real getReal (const Standard_Integer theRank1, const Standard_Integer theRank2 = 0) const;
Standard_Boolean setReal (const Standard_Real theValue, const Standard_Integer theRank1, const Standard_Integer theRank2 = 0, const Standard_Real theTolerance = 0.) const;
```

Similar methods are provided to access references to other objects:

```cpp
Handle(TObj_Object) getReference (const Standard_Integer theRank1, const Standard_Integer theRank2 = 0) const;
Standard_Boolean setReference (const Handle(TObj_Object) &theObject, const Standard_Integer theRank1, const Standard_Integer theRank2 = 0);
```

The method `addReference` gives an easy way to store a sequence of homogeneous references in one label.

```cpp
TDF_Label addReference (const Standard_Integer theRank1, const Handle(TObj_Object) &theObject);
```

Note that while references to other objects should be defined by descendant classes individually according to the type of object, `TObj_Object` provides methods to manipulate (check, remove, iterate) the existing references in the uniform way, as described below.
Persistence

The persistence of the TObj Data Model is implemented with the help of standard OCAF mechanisms (a schema defining necessary plugins, drivers, etc.). This implies the possibility to store/retrieve all data that are stored as standard OCAF attributes. The corresponding handlers are added to the drivers for TObj-specific attributes.

The special tool is provided for classes inheriting from TObj_Object to add the new types of persistence without regeneration of the OCAF schema. The class TObj_Persistence provides basic means for that:

- automatic run-time registration of object types
- creation of a new object of the specified type (one of the registered types)

Two macros defined in the file TObj_Persistence.hxx have to be included in the definition of each model object class inheriting TObj_Object to activate the persistence mechanism:

```c
DECLARE_TOBJOCAF_PERSISTENCE (classname, ancestorname)
```

Should be included in the private section of declaration of each class inheriting TObj_Object (hxx file). This macro adds an additional constructor to the object class, and declares an auxiliary (private) class inheriting TObj_Persistence that provides a tool to create a new object of the proper type.

```c
IMPLEMENT_TOBJOCAF_PERSISTENCE (classname)
```

Should be included in .cxx file of each object class that should be saved and restored. This is not needed for abstract types of objects. This macro implements the functions declared by the previous macro and creates a static member that automatically registers that type for persistence.

When the attribute TObj_TObject that contains the interface object is saved, its persistence handler stores the runtime type of the object class. When the type is restored the handler dynamically recognizes the type
and creates the corresponding object using mechanisms provided by TObj_Persistence.
Names of objects

All TObj model objects have names by which the user can refer to the object. Upon creation, each object receives a default name, constructed from the prefix corresponding to the object type (more precisely, the prefix is defined by the partition to which the object belongs), and the index of the object in the current partition. The user has the possibility to change this name. The uniqueness of the name in the model is ensured by the naming mechanism (if the name is already used, it cannot be attributed to another object). This default implementation of TObj package works with a single instance of the name container (dictionary) for name registration of objects and it is enough in most simple projects. If necessary, it is easy to redefine a couple of object methods (for instance GetDictionary()) and to take care of construction and initialization of containers.

This functionality is provided by the following methods:

```cpp
virtual Handle(TObj_TNameContainer) GetDictionary() const;

Handle(TCollection_HExtendedString) GetName() const;
Standard_Boolean GetName( TCollection_ExtendedString& theName ) const;
Standard_Boolean GetName( TCollection_AsciiString& theName ) const;

virtual Standard_Boolean SetName( const Handle(TCollection_HExtendedString)& theName ) const;
Standard_Boolean SetName( const Handle(TCollection_HAsciiString)& theName )
```

Returns the name container where the name of object should be registered. The default implementation returns the model name container.

Returns the object name. The methods with in / out argument return False if the object name is not defined.
Attributes a new name to the object and returns **True** if the name has been attributed successfully. Returns False if the name has been already attributed to another object. The last two methods are short-cuts to the first one.
References between objects

Class *TObj_Object* allows creating references to other objects in the model. Such references describe relations among objects which are not adequately reflected by the hierarchical objects structure in the model (parent-child relationship).

The references are stored internally using the attribute *TObj_TReference*. This attribute is located in the sub-label of the referring object (called *master*) and keeps reference to the main label of the referred object. At the same time the referred object can maintain the back reference to the master object.
The back references are stored not in the OCAF document but as a transient field of the object; they are created when the model is restored from file, and updated automatically when the references are manipulated. The class `TObj_TReference` allows storing references between objects from different `TObj` models, facilitating the construction of complex relations between objects.

The most used methods for work with references are:

```cpp
virtual Standard_Boolean HasReference( const Handle(TObj_Object)& theObject ) const;
```

Returns True if the current object refers to the indicated object.

```cpp
virtual Handle(TObj_ObjectIterator) GetReferences ( const Handle(Standard_Type)& theType = NULL ) const;
```

Returns an iterator on the object references. The optional argument `theType` restricts the types of referred objects, or does not if it is NULL.

```cpp
virtual void RemoveAllReferences();
```

Removes all references from the current object.

```cpp
virtual void RemoveReference( const Handle(TObj_Object)& theObject );
```

Removes the reference to the indicated object.

```cpp
virtual Handle(TObj_ObjectIterator) GetBackReferences ( const Handle(Standard_Type)& theType = NULL ) const;
```

Returns an iterator on the object back references. The argument `theType` restricts the types of master objects, or does not if it is NULL.

```cpp
virtual void ReplaceReference ( const Handle(TObj_Object)& theOldObject, const Handle(TObj_Object)& theNewObject );
```
Replaces the reference to theOldObject by the reference to theNewObject. The handle theNewObject may be NULL to remove the reference.

```cpp
virtual Standard_Boolean RelocateReferences ( const TDF_Label& theFromRoot, const TDF_Label& theToRoot, const Standard_Boolean theUpdateackRefs = Standard_True );
```

Replaces all references to a descendant label of theFromRoot by the references to an equivalent label under theToRoot. Returns False if the resulting reference does not point at a TObj_Object. Updates back references if theUpdateackRefs is True.

```cpp
virtual Standard_Boolean CanRemoveReference ( const Handle(TObj_Object)& theObj ) const;
```

Returns True if the reference can be removed and the master object will remain valid (weak reference). Returns False if the master object cannot be valid without the referred object (strong reference). This affects the behaviour of objects removal from the model – if the reference cannot be removed, either the referred object will not be removed, or both the referred and the master objects will be removed (depends on the deletion mode in the method Detach)
Creation and deletion of objects

It is recommended that all objects inheriting from \textit{TObj\_Object} should implement the same approach to creation and deletion.

The object of the \textit{TObj} data model cannot be created independently of the model instance, as far as it stores the object data in OCAF data structures. Therefore an object class cannot be created directly as its constructor is protected.

Instead, each object should provide a static method \textit{Create()}, which accepts the model, with the label, which stores the object and other type-dependent parameters necessary for proper definition of the object. This method creates a new object with its data (a set of OCAF attributes) in the specified label, and returns a handle to the object's interface.

The method \textit{Detach()} is provided for deletion of objects from OCAF model. Object data are deleted from the corresponding OCAF label; however, the handle on object remains valid. The only operation available after object deletion is the method \textit{IsAlive()} checking whether the object has been deleted or not, which returns False if the object has been deleted.

When the object is deleted from the data model, the method checks whether there are any alive references to the object. Iterating on references the object asks each referring (master) object whether the reference can be removed. If the master object can be unlinked, the reference is removed, otherwise the master object will be removed too or the referred object will be kept alive. This check is performed by the method \textit{Detach}, but the behavior depends on the deletion mode \textit{TObj\_DeletingMode}:

- \textbf{TObj\_FreeOnly} – the object will be destroyed only if it is free, i.e. there are no references to it from other objects
- \textbf{TObj\_KeepDepending} – the object will be destroyed if there are no strong references to it from master objects (all references can be unlinked)
- \textbf{TObj\_Force} – the object and all depending master objects that have strong references to it will be destroyed.
The most used methods for object removing are:

```cpp
virtual Standard_Boolean CanDetachObject (const TObj_DeletingMode theMode = TObj_FreeOnly );
```

Returns **True** if the object can be deleted with the indicated deletion mode.

```cpp
virtual Standard_Boolean Detach ( const TObj_DeletingMode theMode = TObj_FreeOnly );
```

Removes the object from the document if possible (according to the indicated deletion mode). Unlinks references from removed objects. Returns **True** if the objects have been successfully deleted.
Transformation and replication of object data

*TObj_Object* provides a number of special virtual methods to support replications of objects. These methods should be redefined by descendants when necessary.

```cpp
virtual Handle(TObj_Object) Clone (const TDF_Label& theTargetLabel, Handle(TDF_RelocationTable) theRelocTable = 0);
```

Copies the object to theTargetLabel. The new object will have all references of its original. Returns a handle to the new object (null handle if fail). The data are copied directly, but the name is changed by adding the postfix *_copy*. To assign different names to the copies redefine the method:

```cpp
virtual Handle(TCollection_HExtendedString) GetNameForClone (const Handle(TObj_Object)& ) const;
```

Returns the name for a new object copy. It could be useful to return the same object name if the copy will be in the other model or in the other partition with its own dictionary. The method *Clone* uses the following public methods for object data replications:

```cpp
virtual void CopyReferences (const const Handle(TObj_Object)& theTargetObject, const Handle(TDF_RelocationTable) theRelocTable);
```

Adds to the copy of the original object its references.

```cpp
virtual void CopyChildren (TDF_Label& theTargetLabel, const Handle(TDF_RelocationTable) theRelocTable);
```

Copies the children of an object to the target child label.
Object flags

Each instance of `TObj_Object` stores a set of bit flags, which facilitate the storage of auxiliary logical information assigned to the objects (object state). Several typical state flags are defined in the enumeration `ObjectState`:

- `ObjectState_Hidden` – the object is marked as hidden
- `ObjectState_Saved` – the object has (or should have) the corresponding saved file on disk
- `ObjectState_Imported` – the object is imported from somewhere
- `ObjectState_ImportedByFile` – the object has been imported from file and should be updated to have correct relations with other objects
- `ObjectState_Ordered` – the partition contains objects that can be ordered.

The user (developer) can define any new flags in descendant classes. To set/get an object, the flags use the following methods:

```plaintext
Standard_Integer GetFlags() const;
void SetFlags( const Standard_Integer theMask );
Standard_Boolean TestFlags( const Standard_Integer theMask ) const;
void ClearFlags( const Standard_Integer theMask = 0 );
```

In addition, the generic virtual interface stores the logical properties of the object class in the form of a set of bit flags. Type flags can be received by the method:

```plaintext
virtual Standard_Integer GetTypeFlags() const;
```

The default implementation returns the flag `Visible` defined in the enumeration `TypeFlags`. This flag is used to define visibility of the object for the user browsing the model (see class `TObj_HIDDENPartition`). Other flags can be added by the applications.
Partitions

The special kind of objects defined by the class *TObj_Partition* (and its descendant *TObj_HiddenPartition*) is provided for partitioning the model into a hierarchical structure. This object represents the container of other objects. Each *TObj* model contains the main partition that is placed in the same OCAF label as the model object, and serves as a root of the object's tree. A hidden partition is a simple partition with a predefined hidden flag.

The main partition object methods:

```cpp
TDF_Label NewLabel() const;
```

Allocates and returns a new label for creation of a new child object.

```cpp
voidSetNamePrefix(const Handle(TCollection_HExtendedString)& thePrefix);
```

Defines the prefix for automatic generation of names of the newly created objects.

```cpp
Handle(TCollection_HExtendedString) GetNamePrefix() const;
```

Returns the current name prefix.

```cpp
Handle(TCollection_HExtendedString) GetNewName(const Standard_Boolean theIsToChangeCount) const;
```

Generates the new name and increases the internal counter of child objects if theIsToChangeCount is **True**.

```cpp
Standard_Integer GetLastIndex() const;
```

Returns the last reserved child index.
void SetLastIndex( const Standard_Integer theIndex );

Sets the last reserved index.
Auxiliary classes

Apart from the model and the object, package TObj provides a set of auxiliary classes:

- **TObj_Application** – defines OCAF application supporting existence and operation with TObj documents.
- **TObj_Assistant** – class provides an interface to the static data to be used during save and load operations on models. In particular, in case of cross-model dependencies it allows passing information on the parent model to the OCAF loader to correctly resolve the references when loading a dependent model.
- **TObj_TReference** – OCAF attribute describes the references between objects in the TObj model(s). This attribute stores the label of the referred model object, and provides transparent cross-model references. At runtime, these references are simple Handles; in persistence mode, the cross-model references are automatically detected and processed by the persistence mechanism of TObj_TReference attribute.
- Other classes starting with TObj_T... – define OCAF attributes used to store TObj-specific classes and some types of data on OCAF labels.
- **Iterators** – a set of classes implementing TObj_ObjectIterator interface, used for iterations on TObj objects:
  - **TObj_ObjectIterator** – a basic abstract class for other TObj iterators. Iterates on TObj_Object instances.
  - **TObj_LabelIterator** – iterates on object labels in the TObj model document
  - **TObj_ModelIterator** – iterates on all objects in the model. Works with sequences of other iterators.
  - **TObj_OcafObjectIterator** – Iterates on TObj data model objects. Can iterate on objects of a specific type.
  - **TObj_ReferenceIterator** – iterates on object references.
  - **TObj_SequenceIterator** – iterates on a sequence of TObj objects.
  - **TObj_CheckModel** – a tool that checks the internal consistency of the model. The basic implementation checks only the consistency of references between objects.
The structure of \textit{TObj} iterators hierarchy is presented below:
Packaging

The TObj sources are distributed in the following packages:

- **TObj** – defines basic classes that implement TObj interfaces for OCAF-based modelers.
- **BinLDrivers, XmlLDivers** – binary and XML driver of TObj package
- **BinLPlugin, XmlLPlugin** – plug-in for binary and XML persistence
- **BinMObj, XmlMObj** – binary and XML drivers to store and retrieve specific TObj data to or from OCAF document
- **TKBinL, TKXmlL** – toolkits of binary and XML persistence
Draw Test Harness

Table of Contents

- Introduction
  - Overview
- Contents of this documentation
- Getting started
  - Launching DRAW Test Harness
  - Plug-in resource file
  - Activation of commands implemented in the plug-in
- The Command Language
  - Overview
  - Syntax of TCL
  - Accessing variables in TCL and Draw
    - set, unset
    - dset, dval
    - del, dall
  - lists
    - Control Structures
      - if
      - while, for, foreach
      - break, continue
Procedures
  proc
  global, upvar

Basic Commands
  General commands
  help
  source
  spy
  cpulimit
  wait
  chrono

Variable management commands
  isdraw, directory
  whatis, dump
  renamevar, copy
  datadir, save, restore

User defined commands
  set
  get

Graphic Commands
  Axonometric viewer
  view, delete
  axo, pers, top, ...
  mu, md, 2dmu, 2dmd, zoom, 2dzoom
  pu, pd, pl, pr, 2dpu, 2dpd, 2dpl, 2dpr
  fit, 2dfit
  u, d, l, r
  focal, fu, fd
color
dtext
hardcopy, hcolor, xwd
wclick, pick
autodisplay
display, donly
erase, clear, 2dclear
disp, don, era
repaint, dflush

AIS viewer -- view commands
vinit
vhelp
vtop
vaxo
vsetbg
vclear
vrepaint
vfit
vzfit
vreadpixel
vselect
vmoveto
vviewparams
vchangepselected
vzclipping
vnbsselected
vpurgedisplay
vhlr
vhlrtypen
vcamera
vsstereo
vfrustumculling

AIS viewer -- display commands
AIS viewer -- object commands

- vtrihedron
- vplanetri
- vsize
- vaxis
- vaxispara
- vaxisortho
- vpoint
- vplane
- vplane
- vplaneparapara
ivtkdispmode
ivtksetselmode
ivtkmoveto
ivtkselect
ivtkdump
ivtkbgcolor

OCAF commands

Application commands

NewDocument
IsInSession
ListDocuments
Open
Close
Save
SaveAs

Basic commands

Label
NewChild
Children
ForgetAll
Application commands

Main
UndoLimit
Undo
Redo
OpenCommand
CommitCommand
NewCommand
AbortCommand
Copy
UpdateLink
CopyWithLink
UpdateXLinks
DumpDocument
Data Framework commands
- MakeDF
- ClearDF
- CopyDF
- CopyLabel
- MiniDumpDF
- XDumpDF

General attributes commands
- SetInteger
- GetInteger
- SetReal
- GetReal
- SetIntArray
- GetIntArray
- SetRealArray
- GetRealArray
- SetComment
- GetComment
- SetExtStringArray
- GetExtStringArray
- setName
- getName
- SetReference
- GetReference
- SetUAttribute
- GetUAttribute
- SetFunction
- GetFunction
- NewShape
- SetShape
- GetShape

Geometric attributes commands
- SetPoint
- GetPoint
<table>
<thead>
<tr>
<th>Tree attributes commands</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SetAxis</td>
<td>--</td>
</tr>
<tr>
<td>GetAxis</td>
<td>--</td>
</tr>
<tr>
<td>SetPlane</td>
<td>--</td>
</tr>
<tr>
<td>GetPlane</td>
<td>--</td>
</tr>
<tr>
<td>SetGeometry</td>
<td>--</td>
</tr>
<tr>
<td>GetGeometryType</td>
<td>--</td>
</tr>
<tr>
<td>SetConstraint</td>
<td>--</td>
</tr>
<tr>
<td>GetConstraint</td>
<td>--</td>
</tr>
<tr>
<td>SetVariable</td>
<td>--</td>
</tr>
<tr>
<td>GetVariable</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Standard presentation commands</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AISInitViewer</td>
<td>--</td>
</tr>
<tr>
<td>AISRepaint</td>
<td>--</td>
</tr>
<tr>
<td>AISDisplay</td>
<td>--</td>
</tr>
<tr>
<td>AISUpdate</td>
<td>--</td>
</tr>
<tr>
<td>AISUpdate</td>
<td>--</td>
</tr>
<tr>
<td>AISRemove</td>
<td>--</td>
</tr>
<tr>
<td>AISRemove</td>
<td>--</td>
</tr>
<tr>
<td>AISSet</td>
<td>--</td>
</tr>
<tr>
<td>AISDriver</td>
<td>--</td>
</tr>
<tr>
<td>AISUnset</td>
<td>--</td>
</tr>
</tbody>
</table>
AISTransparency
AISHasOwnTranspare
AISMaterial
AISHasOwnMaterial
AISColor
AISHasOwnColor
Geometry commands
Overview
Curve creation
- point
- line
- circle
- ellipse
- hyperbola
- parabola
- beziercurve, 2dbeziercurve
- bsplinecurve, 2dbsplinecurve, pbsplinecurve, 2dpbsplinecurve
- uiso, viso
- to3d, to2d
- project
Surface creation
- plane
- cylinder
- cone
- sphere
- torus
- beziersurf
- bsplinesurf, upbsplinesurf, vpbsplinesurf, uvpbsplinesurf
- trim, trimu, trimv
- offset
Curve and surface modifications
- reverse, ureverse, vreverse
- exchuv
- segment, segsur
- iincudeg, incvdeg
- cmovep, movep, movecolp, moverowp
- insertpole, rempole, remcolpole, remrowpole
- insertknot, insertuknot, insertvknot
- remknot, remuknot, remvknot
- setperiodic, setnotperiodic, setupperiodic, setvperiodic, setvnotperiodic
- setorigin, setuorigin, setvorigin

Transformations
- translate, dtranslate
- rotate, 2drotate
- pmirror, lmirror, smirror, dpmirror,
dlmirror  
\downarrow pscale, dpscale
\downarrow Curve and surface analysis  
\downarrow coord
\downarrow cvalue, 2dcvalue
\downarrow svalue
\downarrow localprop, minmaxcurandinf
\downarrow parameters
\downarrow proj, 2dproj
\downarrow surface_radius
\downarrow Intersections  
\downarrow intersect
\downarrow dintersect
\downarrow Approximations  
\downarrow appro, dapprox
\downarrow surfapp, grilapp
\downarrow Projections  
\downarrow projponf
\downarrow Constraints  
\downarrow cirtang
\downarrow lintan
\downarrow Display  
\downarrow dmod, discr, defle
\downarrow nbiso
\downarrow clpoles, shpoles
\downarrow clknots, shknots
\downarrow Topology commands
\downarrow Basic topology  
\downarrow isos, discretisation
\downarrow orientation, complement, invert, normals, range
¬ explode, exwire, nbshapes
¬ emptycopy, add, compound
¬ compare
¬ issubshape

¬ Curve and surface topology
¬ vertex
¬ mkpoint
¬ edge, mkedge, uisoedge, visoedge
¬ wire, polyline, polyvertex
¬ profile
¬ bsplineprof
¬ mkoffset
¬ mkplane, mkface
¬ mkcurve, mksurface
¬ pcurve
¬ chfi2d
¬ nproject

¬ Primitives
¬ box, wedge
¬ pcylinder, pcone, psphere, ptorus
¬ halfspace

¬ Sweeping
¬ prism
¬ revol
¬ pipe
¬ mksweep, addsweep, setsweep, deletesweep, buildsweep,
simulsweep
thrusections

Topological transformation
tcopy
tmove, treset
ttranslate, trotate
tmirror, tscale

Old Topological operations
fuse, cut, common
section, psection
sewing

New Topological operations
bparallelmode
bop, bopfuse, bopcut, boptuc, bopcommon
bopsection
bopcheck, bopargshape

Drafting and blending
depouille
chamf
blend
bfuseblend
bcutblend
mkevol,
updatevol,
buildevol

Analysis of topology and geometry
lprops, sprops, vprops
bounding
↓distmini
↓xdistef, xdistcs, xdistcc, xdistcc2dc2dss, xdistcc2ds
↓checkshape
↓tolsphere
↓validrange
↓Surface creation
↓gplate,
↓filling, fillingparam
↓Complex Topology
↓offsetshape, offsetcompshape
↓featprism, featdprism, featrevol, featlf, featrf
↓draft
↓deform
↓nurbsconvert
↓edgestofaces
↓Texture Mapping to a Shape
↓vttexture
↓vtexscale
↓vtexorigin
↓vtexrepeat
↓vtexdefault
↓General Fuse Algorithm commands
↓Definitions
↓General commands
↓Commands for Intersection Part
↓bopds
↓bopdsdump
↓bopindex
bopiterator
bopinterf
bopsp
bopcb
bopfin
bopfon
bopwho
bopnews

Commands for the Building Part
bopim

Data Exchange commands

IGES commands
igesread
tplostttrim
brepiges

STEP commands
stepread
stepwrite

General commands
count
data
elabel
entity
enum
estatus
fromshape
givecount
givelist
listcount
listitems
listtypes
newmodel
param
sumcount
Overview of XDE commands
  - ReadIges
  - ReadStep
  - WriteIges
  - WriteStep
  - XFileCur
  - XFileList
  - XFileSet
  - XFromShape

XDE general commands
  - XNewDoc
  - XShow
  - XStat
  - XWdump
  - Xdump

XDE shape commands
  - XAddComponent
  - XAddShape
  - XFindComponent
  - XFindShape
  - XGetFreeShapes
  - XGetOneShape
  - XGetReferredShape
  - XGetShape
  - XGetTopLevelShapes
  - XLabelInfo
  - XNewShape
  - XRemoveComponent
  - XRemoveShape
- XShapeMassProps
- XShapeVolume

- Shape Healing commands

- General commands
  - bsplres
  - checkfclass2d
  - checkoverlappedges
  - comtol
  - convtorevol
  - directfaces
  - expshape
  - fixsmall
  - fixsmalledges
  - fixshape
  - fixw gaps
  - offsetcurve, offset2dcurve
  - projcurve
  - projp curve
  - projface
  - scaleshape
  - settolerance
  - splitface
  - statshape
  - tolerance

- Conversion commands
  - DT_ClosedSplit
  - DT_ShapeConvert, DT_ShapeConvertRev
  - DT_ShapeDivide
  - DT_SplitAngle
  - DT_SplitCurve
  - DT_SplitCurve2d
  - DT_SplitSurface
  - DT_ToBspl
Performance evaluation commands

VDrawSphere

Simple vector algebra and measurements

Vector algebra commands

vec
2dvec
pln
module
2dmodule
norm
2dnorm
inverse
2dinverse
2dort
2dort
2distpp
2ddistpp
distplp
distlp
2ddistlp
distlpp
2ddistlpp
barycen
2dbarycen
cross
2dcross
dot
2ddot
scale
2dscale

Measurements commands

pnt
pntc
2dpntc
Inspector commands
	tinspector

Extending Test Harness with custom commands

Custom command implementation

Registration of commands in Test Harness

Creating a toolkit (library) as a plug-in

Creation of the plug-in resource file

Dynamic loading and activation
Introduction

This manual explains how to use Draw, the test harness for Open CASCADE Technology (OCCT). Draw is a command interpreter based on TCL and a graphical system used to test and demonstrate Open CASCADE Technology modeling libraries.
Overview

Draw is a test harness for Open CASCADE Technology. It provides a flexible and easy to use means of testing and demonstrating the OCCT modeling libraries.

Draw can be used interactively to create, display and modify objects such as curves, surfaces and topological shapes.

Scripts may be written to customize Draw and perform tests. New types of objects and new commands may be added using the C++ programing language.

Draw consists of:

- A command interpreter based on the TCL command language.
- A 3d graphic viewer based on the X system.
- A basic set of commands covering scripts, variables and graphics.
- A set of geometric commands allowing the user to create and modify curves and surfaces and to use OCCT geometry algorithms. This set of commands is optional.
- A set of topological commands allowing the user to create and modify BRep shapes and to use the OCCT topology algorithms.

There is also a set of commands for each delivery unit in the modeling libraries:

- GEOMETRY,
- TOPOLOGY,
- ADVALGOS,
- GRAPHIC,
- PRESENTATION.
Contents of this documentation

This documentation describes:

- The command language.
- The basic set of commands.
- The graphical commands.
- The Geometry set of commands.
- The Topology set of commands.
- OCAF commands.
- Data Exchange commands
- Shape Healing commands

This document is a reference manual. It contains a full description of each command. All descriptions have the format illustrated below for the exit command.

```
exit
```

Terminates the Draw, TCL session. If the commands are read from a file using the source command, this will terminate the file.

**Example:**

```
# this is a very short example
exit
```
Getting started

Install Draw and launch Emacs. Get a command line in Emacs using Esc x and key in woksh.

All DRAW Test Harness can be activated in the common executable called DRAWEXE. They are grouped in toolkits and can be loaded at runtime thereby implementing dynamically loaded plug-ins. Thus, it is possible to work only with the required commands adding them dynamically without leaving the Test Harness session.

Declaration of available plug-ins is done through the special resource file(s). The pload command loads the plug-in in accordance with the specified resource file and activates the commands implemented in the plug-in.

Launching DRAW Test Harness

Test Harness executable DRAWEXE is located in the $CASROOT/<platform>/bin directory (where <platform> is Win for Windows and Linux for Linux operating systems). Prior to launching it is important to make sure that the environment is correctly setup (usually this is done automatically after the installation process on Windows or after launching specific scripts on Linux).

Plug-in resource file

Open CASCADE Technology is shipped with the DrawPlugin resource file located in the $CASROOT/src/DrawResources directory.

The format of the file is compliant with standard Open CASCADE Technology resource files (see the Resource_Manager.hxx file for details).

Each key defines a sequence of either further (nested) keys or a name of the dynamic library. Keys can be nested down to an arbitrary level. However, cyclic dependencies between the keys are not checked.

Example: (excerpt from DrawPlugin):
### Activation of commands implemented in the plug-in

To load a plug-in declared in the resource file and to activate the commands the following command must be used in Test Harness:

```
pload [-PluginFileName] [[Key1] [Key2]...]  
```

where:

- `-PluginFileName` – defines the name of a plug-in resource file (prefix "," is mandatory) described above. If this parameter is omitted then the default name `DrawPlugin` is used.
- `Key` – defines the key(s) enumerating plug-ins to be loaded. If no keys are specified then the key named `DEFAULT` is used (if there is no such key in the file then no plug-ins are loaded).

According to the OCCT resource file management rules, to access the resource file the environment variable `CSF_PluginFileNameDefaults` (and optionally `CSF_PluginFileNameUserDefaults`) must be set and point to the directory storing the resource file. If it is omitted then the plug-in resource file will be searched in the `$CASROOT/src/DrawResources` directory.

```
Draw[]  pload -DrawPlugin OCAF  
```

This command will search the resource file `DrawPlugin` using variable `CSF_DrawPluginDefaults` (and `CSF_DrawPluginUserDefaults`) and will start with the OCAF key. Since the `DrawPlugin` is the file shipped with Open CASCADE Technology it will be found in the `$CASROOT/src/DrawResources` directory (unless this location is redefined by user's variables). The OCAF key will be recursively extracted into two toolkits/plug-ins: `TKDCAF` and `TKViewerTest` (e.g. on Windows they correspond to `TKDCAF.dll` and `TKViewerTest.dll`). Thus,
commands implemented for Visualization and OCAF will be loaded and activated in Test Harness.

\textbf{Draw[] pload} (equivalent to pload -DrawPlugin DEFAULT).

This command will find the default DrawPlugin file and the DEFAULT key. The latter finally maps to the TKTopTest toolkit which implements basic modeling commands.
The Command Language

Overview

The command language used in Draw is Tcl. Tcl documentation such as "TCL and the TK Toolkit" by John K. Ousterhout (Addison-Wesley) will prove useful if you intend to use Draw extensively.

This chapter is designed to give you a short outline of both the TCL language and some extensions included in Draw. The following topics are covered:

- Syntax of the TCL language.
- Accessing variables in TCL and Draw.
- Control structures.
- Procedures.
Syntax of TCL

TCL is an interpreted command language, not a structured language like C, Pascal, LISP or Basic. It uses a shell similar to that of csh. TCL is, however, easier to use than csh because control structures and procedures are easier to define. As well, because TCL does not assign a process to each command, it is faster than csh.

The basic program for TCL is a script. A script consists of one or more commands. Commands are separated by new lines or semicolons.

```
set a 24
set b 15
set a 25; set b 15
```

Each command consists of one or more words; the first word is the name of a command and additional words are arguments to that command.

Words are separated by spaces or tabs. In the preceding example each of the four commands has three words. A command may contain any number of words and each word is a string of arbitrary length.

The evaluation of a command by TCL is done in two steps. In the first step, the command is parsed and broken into words. Some substitutions are also performed. In the second step, the command procedure corresponding to the first word is called and the other words are interpreted as arguments. In the first step, there is only string manipulation, The words only acquire meaning in the second step by the command procedure.

The following substitutions are performed by TCL:

Variable substitution is triggered by the $ character (as with csh), the content of the variable is substituted; {} may be used as in csh to enclose the name of the variable.

Example:

```
# set a variable value
```
set file documentation
puts $file #to display file contents on the screen

# a simple substitution, set psfile to documentation.ps
set psfile $file.ps
puts $psfile

# another substitution, set pfile to documentationPS
set pfile ${file}PS

# a last one,
# delete files NEWdocumentation and OLDdocumentation
foreach prefix {NEW OLD} {rm $prefix$file}

Command substitution is triggered by the [] characters. The brackets must enclose a valid script. The script is evaluated and the result is substituted.

Compare command construction in csh.

Example:

cset degree 30
cset pi 3.14159265
cset expr is a command evaluating a numeric expression
cset radian [expr $pi*$degree/180]

Backslash substitution is triggered by the backslash character. It is used to insert special characters like $, [, ], etc. It is also useful to insert a new line, a backslash terminated line is continued on the following line.

TCL uses two forms of quoting to prevent substitution and word breaking.

Double quote quoting enables the definition of a string with space and tabs as a single word. Substitutions are still performed inside the inverted commas " ".

Example:
Braces *quoting* prevents all substitutions. Braces are also nested. The main use of braces is to defer evaluation when defining procedures and control structures. Braces are used for a clearer presentation of TCL scripts on several lines.

**Example:**

```tcl
set x 0
# this will loop for ever
# because while argument is ;0 < 3;
while ;$x < 3; {set x [expr $x+1]}
# this will terminate as expected because
# while argument is {$x < 3}
while {$x < 3} {set x [expr $x+1]}
# this can be written also
while {$x < 3} {
set x [expr $x+1]
}
# the following cannot be written
# because while requires two arguments
while {$x < 3} {
set x [expr $x+1]
}
```

Comments start with a # character as the first non-blank character in a command. To add a comment at the end of the line, the comment must be preceded by a semi-colon to end the preceding command.

**Example:**

```tcl
# This is a comment
set a 1 # this is not a comment
set b 1; # this is a comment
```
The number of words is never changed by substitution when parsing in TCL. For example, the result of a substitution is always a single word. This is different from csh but convenient as the behavior of the parser is more predictable. It may sometimes be necessary to force a second round of parsing. **eval** accomplishes this: it accepts several arguments, concatenates them and executes the resulting script.

**Example:**

```tcl
# I want to delete two files
set files ;foo bar;

# this will fail because rm will receive only one argument
# and complain that ;foo bar; does not exit
exec rm $files

# a second evaluation will do it
```
Accessing variables in TCL and Draw

TCL variables have only string values. Note that even numeric values are stored as string literals, and computations using the `expr` command start by parsing the strings. Draw, however, requires variables with other kinds of values such as curves, surfaces or topological shapes.

TCL provides a mechanism to link user data to variables. Using this functionality, Draw defines its variables as TCL variables with associated data.

The string value of a Draw variable is meaningless. It is usually set to the name of the variable itself. Consequently, preceding a Draw variable with a `$` does not change the result of a command. The content of a Draw variable is accessed using appropriate commands.

There are many kinds of Draw variables, and new ones may be added with C++. Geometric and topological variables are described below.

Draw numeric variables can be used within an expression anywhere a Draw command requires a numeric value. The `expr` command is useless in this case as the variables are stored not as strings but as floating point values.

Example:

```plaintext
# dset is used for numeric variables
# pi is a predefined Draw variable
dset angle pi/3 radius 10
point p radius*cos(angle) radius*sin(angle) 0
```

It is recommended that you use TCL variables only for strings and Draw for numerals. That way, you will avoid the `expr` command. As a rule, Geometry and Topology require numbers but no strings.

**set, unset**

Syntax:
set varname [value]
unset varname [varname varname ...]

`set` assigns a string value to a variable. If the variable does not already exist, it is created.

Without a value, `set` returns the content of the variable.

`unset` deletes variables. It is also used to delete Draw variables.

**Example:**

```tcl
set a "Hello world"
set b "Goodbye"
set a
== "Hello world"
unset a b
set a
```

**Note,** that the `set` command can set only one variable, unlike the `dset` command.

**dset, dval**

**Syntax**

```tcl
dset var1 value1 vr2 value2 ...
dval name
```

`dset` assigns values to Draw numeric variables. The argument can be any numeric expression including Draw numeric variables. Since all Draw commands expect a numeric expression, there is no need to use `$` or `expr`. The `dset` command can assign several variables. If there is an odd number of arguments, the last variable will be assigned a value of 0. If the variable does not exist, it will be created.

`dval` evaluates an expression containing Draw numeric variables and returns the result as a string, even in the case of a single variable. This is not used in Draw commands as these usually interpret the expression. It is used for basic TCL commands expecting strings.
Example:

```
# z is set to 0
dset x 10 y 15 z
== 0

# no $ required for Draw commands
point p x y z

# "puts" prints a string
puts ;x = [dval x], cos(x/pi) = [dval cos(x/pi)];
== x = 10, cos(x/pi) = -0.99913874099467914
```

**Note**, that in TCL, parentheses are not considered to be special characters. Do not forget to quote an expression if it contains spaces in order to avoid parsing different words. (a + b) is parsed as three words: " (a + b)" or (a+b) are correct.

**del, dall**

Syntax:

```
del varname_pattern [varname_pattern ...]
dall
```

*del* command does the same thing as *unset*, but it deletes the variables matched by the pattern.

*dall* command deletes all variables in the session.
lists

TCL uses lists. A list is a string containing elements separated by spaces or tabs. If the string contains braces, the braced part accounts as one element.

This allows you to insert lists within lists.

Example:

```tcl
# a list of 3 strings
;a b c;

# a list of two strings the first is a list of 2
;{a b} c;
```

Many TCL commands return lists and `foreach` is a useful way to create loops on list elements.

Control Structures

TCL allows looping using control structures. The control structures are implemented by commands and their syntax is very similar to that of their C counterparts (`if`, `while`, `switch`, etc.). In this case, there are two main differences between TCL and C:

- You use braces instead of parentheses to enclose conditions.
- You do not start the script on the next line of your command.

if

Syntax

```
if condition script [elseif script .... else script]
```

If evaluates the condition and the script to see whether the condition is true.
Example:

```bash
if {$x > 0} {
  puts ;positive;
} elseif {$x == 0} {
  puts ;null;
} else {
  puts ;negative;
}
```

while, for, foreach

Syntax:

- **while** condition script
- **for** init condition reinit script
- **foreach** varname list script

The three loop structures are similar to their C or csh equivalent. It is important to use braces to delay evaluation. **foreach** will assign the elements of the list to the variable before evaluating the script.

Example:

```bash
# while example
dset x 1.1
while {{[dval x] < 100}} {
  circle c 0 0 x
  dset x x*x
}

# for example
# incr var d, increments a variable of d (default 1)
for {set i 0} {$i < 10} {incr i} {
  dset angle $i*pi/10
  point p$i cos(angle0 sin(angle) 0
}

# foreach example
foreach object {crapo tomson lucas} {display $object}
```
**break, continue**

Syntax:

```plaintext
break
continue
```

Within loops, the `break` and `continue` commands have the same effect as in C.

`break` interrupts the innermost loop and `continue` jumps to the next iteration.

**Example:**

```plaintext
# search the index for which t$i has value ;secret;
for {set i 1} {$i <= 100} {incr i} {
  if {[[set t$i] == ;secret;]} break;
}
```
Procedures

TCL can be extended by defining procedures using the `proc` command, which sets up a context of local variables, binds arguments and executes a TCL script.

The only problematic aspect of procedures is that variables are strictly local, and as they are implicitly created when used, it may be difficult to detect errors.

There are two means of accessing a variable outside the scope of the current procedures: `global` declares a global variable (a variable outside all procedures); `upvar` accesses a variable in the scope of the caller. Since arguments in TCL are always string values, the only way to pass variables is by reference, i.e. passing the name of the variable and using the `upvar` command as in the following examples.

As TCL is not a strongly typed language it is very difficult to detect programming errors and debugging can be tedious. TCL procedures are, of course, not designed for large scale software development but for testing and simple command or interactive writing.

**proc**

Syntax:

```
proc argumentlist script
```

`proc` defines a procedure. An argument may have a default value. It is then a list of the form `{argument value}`. The script is the body of the procedure.

`return` gives a return value to the procedure.

**Example:**

```
# simple procedure
proc hello {} {
    puts ;hello world;
```
# procedure with arguments and default values
proc distance {x1 y1 {x2 0} {y2 0}} {
    set d [expr (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1)]
    return [expr sqrt(d)]
}

proc fact n {
    if {$n == 0} {return 1} else {
        return [expr n*[fact [expr n -1]]]
    }
}

---

global, upvar

Syntax:

```
global varname [varname ...]
upvar varname localname [varname localname ...]
```

global accesses high level variables. Unlike C, global variables are not visible in procedures.

upvar gives a local name to a variable in the caller scope. This is useful when an argument is the name of a variable instead of a value. This is a call by reference and is the only way to use Draw variables as arguments.

Note that in the following examples the $ character is always necessarily used to access the arguments.

Example:

```
# convert degree to radian
# pi is a global variable
proc deg2rad (degree) {
    return [dval pi*$degree/2.]
}

# create line with a point and an angle
proc linang {linename x y angle} {
```
upvar linename 1
line 1 $x$ $y$ cos($angle$) sin($angle$)
}
Basic Commands

This chapter describes all the commands defined in the basic Draw package. Some are TCL commands, but most of them have been formulated in Draw. These commands are found in all Draw applications. The commands are grouped into four sections:

- General commands, which are used for Draw and TCL management.
- Variable commands, which are used to manage Draw variables such as storing and dumping.
- Graphic commands, which are used to manage the graphic system, and so pertain to views.
- Variable display commands, which are used to manage the display of objects within given views.

Note that Draw also features a GUI task bar providing an alternative way to give certain general, graphic and display commands.
General commands

This section describes several useful commands:

- **help** to get information,
- **source** to eval a script from a file,
- **spy** to capture the commands in a file,
- **cpulimit** to limit the process cpu time,
- **wait** to waste some time,
- **chrono** to time commands.

**help**

Syntax:

```
help [command [helpstring group]]
```

Provides help or modifies the help information.

**help** without arguments lists all groups and the commands in each group.

Specifying the command returns its syntax and in some cases, information on the command. The joker * is automatically added at the end so that all completing commands are returned as well.

**Example:**

```
# Gives help on all commands starting with *a*
```

**source**

Syntax:

```
source filename
```

Executes a file.

The **exit** command will terminate the file.
spy

Syntax:

spy [filename]

Saves interactive commands in the file. If spying has already been performed, the current file is closed. spy without an argument closes the current file and stops spying. If a file already exists, the file is overwritten. Commands are not appended.

If a command returns an error it is saved with a comment mark.

The file created by spy can be executed with the source command.

Example:

```
# all commands will be saved in the file ;session;
spy session
# the file ;session; is closed and commands are not saved
spy
```

cpulimit

Syntax:

cpulimit [nbseconds]

cpulimit**limits a process after the number of seconds specified in nbseconds. It is used in tests to avoid infinite loops. **cpulimit without arguments removes all existing limits.

Example:

```
# limit cpu to one hour
cpulimit 3600
```

wait
Syntax:

\[\text{wait [nbseconds]}\]

Suspends execution for the number of seconds specified in \textit{nbseconds}. The default value is ten (10) seconds. This is a useful command for a slide show.

\[
\# \text{ You have ten seconds ...} \\
\text{wait}
\]

\textbf{chrono}

Syntax:

\[\text{chrono [ name start/stop/reset/show/restart/[counter text]]}\]

Without arguments, \textbf{chrono} activates Draw chronometers. The elapsed time, cpu system and cpu user times for each command will be printed.

With arguments, \textbf{chrono} is used to manage activated chronometers. You can perform the following actions with a chronometer.

- run the chronometer (start).
- stop the chronometer (stop).
- reset the chronometer to 0 (reset).
- restart the chronometer (restart).
- display the current time (show).
- display the current time with specified text (output example - \textit{COUNTER text: N}), command \textit{testdiff} will compare such outputs between two test runs (counter).

\textbf{Example:}

\begin{verbatim}
chrono  
==Chronometers activated. 
ptorus t 20 5  
==Elapsed time: 0 Hours 0 Minutes 0.0318 Seconds  
==CPU user time: 0.01 seconds
\end{verbatim}
==CPU system time: 0 seconds
Variable management commands

**isdraw, directory**

Syntax:

```
isdraw varname
directory [pattern]
```

`isdraw` tests to see if a variable is a Draw variable. `isdraw` will return 1 if there is a Draw value attached to the variable.

Use `directory` to return a list of all Draw global variables matching a pattern.

**Example:**

```
set a 1
isdraw a
=== 0

dset a 1
isdraw a
=== 1

circle c 0 0 1 0 5
isdraw c
=== 1

# to destroy all Draw objects with name containing curve
foreach var [directory *curve*] {unset $var}
```

**whatis, dump**

Syntax:
whatis varname [varname ...]
dump varname [varname ...]

**whatis** returns short information about a Draw variable. This is usually the type name.

**dump** returns a brief type description, the coordinates, and if need be, the parameters of a Draw variable.

**Example:**

circle c 0 0 1 0 5  
whatis c  
c is a 2d curve  
dump c  

***** Dump of c *****  
Circle  
Center :0, 0  
XAxis :1, 0  
YAxis : -0, 1  
Radius : 5

**Note** The behavior of **whatis** on other variables (not Draw) is not excellent.

**renamevar, copy**

Syntax:

renamevar varname tovarname [varname tovarname ...]
copy varname tovarname [varname tovarname ...]

- **renamevar** changes the name of a Draw variable. The original variable will no longer exist. Note that the content is not modified. Only the name is changed.
- **copy** creates a new variable with a copy of the content of an existing variable. The exact behavior of **copy** is type dependent; in the case...
of certain topological variables, the content may still be shared.

Example:

```tcl
circle c1 0 0 1 0 5
renamevar c1 c2

# curves are copied, c2 will not be modified
copy c2 c3
```

data dir, save, restore

Syntax:

```tcl
datadir [directory]
save variable [filename]
restore filename [variablename]
```

- **datadir** without arguments prints the path of the current data directory.
- **datadir** with an argument sets the data directory path. If the path starts with a dot (.) only the last directory name will be changed in the path.

- **save** writes a file in the data directory with the content of a variable. By default the name of the file is the name of the variable. To give a different name use a second argument.
- **restore** reads the content of a file in the data directory in a local variable. By default, the name of the variable is the name of the file. To give a different name, use a second argument.

The exact content of the file is type-dependent. They are usually ASCII files and so, architecture independent.

Example:

```tcl
# note how TCL accesses shell environment variables
# using $env()
datadir
```
datadir $env(WBCONTAINER)/data/default
==/adv_20/BAG/data/default

box b 10 20 30
save b theBox
==/adv_20/BAG/data/default/theBox

# when TCL does not find a command it tries a shell command
ls [datadir]
== theBox

restore theBox
== theBox
User defined commands

*DrawTrSurf* provides commands to create and display a Draw geometric variable from a *Geom_Geometry* object and also get a *Geom_Geometry* object from a Draw geometric variable name.

*DBRep* provides commands to create and display a Draw topological variable from a *TopoDS.Shape* object and also get a *TopoDS.Shape* object from a Draw topological variable name.

set

In *DrawTrSurf* package:

```c++
void Set(Standard_CString& Name, const gp_Pnt& G) ;
void Set(Standard_CString& Name, const gp_Pnt2d& G) ;
void Set(Standard_CString& Name, const Handle(Geom_Geometry)& G) ;
void Set(Standard_CString& Name, const Handle(Geom2d.Curve)& C) ;
void Set(Standard_CString& Name, const Handle(Poly_Triangulation)& T) ;
void Set(Standard_CString& Name, const Handle(Poly_Polygon3D)& P) ;
void Set(Standard_CString& Name, const Handle(Poly_Polygon2D)& P) ;
```

In *DBRep* package:

```c++
void Set(const Standard_CString Name, const TopoDS.Shape& S) ;
```

Example of *DrawTrSurf*

```c++
Handle(Geom2d_Circle) C1 = new Geom2d_Circle (gce_MakeCirc2d (gp_Pnt2d(50,0,)) 25));
```
DrawTrSurf::Set(char*, C1);

Example of *DBRep*

TopoDS_Solid B;
B = BRepPrimAPI_MakeBox (10,10,10);
DBRep::Set(char*,B);

**get**

In *DrawTrSurf* package:

Handle_Geom_Geometry Get(Standard_CString& Name);

In *DBRep* package:

TopoDS_Shape Get(Standard_CString& Name,
const TopAbs_ShapeEnum Typ = TopAbs_SHAPE,
const Standard_Boolean Complain = Standard_True);

Example of *DrawTrSurf*

Standard_Integer MyCommand
(Draw_Interpretor& theCommands,
Standard_Integer argc, char** argv)
{
......
// Creation of a Geom_Geometry from a Draw geometric
// name
Handle (Geom_Geometry) aGeom=
    DrawTrSurf::Get(argv[1]);
}

Example of *DBRep*

Standard_Integer MyCommand
(Draw_Interpretor& theCommands,
Standard_Integer argc, char** argv)
// Creation of a TopoDS_Shape from a Draw topological name
TopoDS_Solid B = DBRep::Get(argv[1]);
Graphic Commands

Graphic commands are used to manage the Draw graphic system. Draw provides a 2d and a 3d viewer with up to 30 views. Views are numbered and the index of the view is displayed in the window’s title. Objects are displayed in all 2d views or in all 3d views, depending on their type. 2d objects can only be viewed in 2d views while 3d objects – only in 3d views correspondingly.
Axonometric viewer

view, delete

Syntax:

```
view index type [X Y W H]
delete [index]
```

**view** is the basic view creation command: it creates a new view with the given index. If a view with this index already exits, it is deleted. The view is created with default parameters and X Y W H are the position and dimensions of the window on the screen. Default values are 0, 0, 500, 500.

As a rule it is far simpler either to use the procedures **axo, top, left** or to click on the desired view type in the menu under **Views** in the task bar..

**delete** deletes a view. If no index is given, all the views are deleted.

Type selects from the following range:

- **AXON**: Axonometric view
- **PERS**: Perspective view
- **+X+Y**: View on both axes (i.e. a top view), other codes are -X+Y, +Y-Z, etc.
- **-2D-**: 2d view

The index, the type, the current zoom are displayed in the window title.

**Example:**

```
# this is the content of the mu4 procedure
proc mu4 {} {
    delete
    view 1  +X+Z  320  20  400  400
    view 2  +X+Y  320  450  400  400
    view 3  +Y+Z  728  20  400  400
```
See also: axo, pers, top, bottom, left, right, front, back, mu4, v2d, av2d, smallview

axo, pers, top, ...

Syntax:

axo
pers
...
smallview type

All these commands are procedures used to define standard screen layout. They delete all existing views and create new ones. The layout usually complies with the European convention, i.e. a top view is under a front view.

- axo creates a large window axonometric view;
- pers creates a large window perspective view;
- top, bottom, left, right, front, back create a large window axis view;
- mu4 creates four small window views: front, left, top and axo.
- v2d creates a large window 2d view.
- av2d creates two small window views, one 2d and one axo
- smallview creates a view at the bottom right of the screen of the given type.

See also: view, delete

mu, md, 2dmu, 2dmd, zoom, 2dzoom

Syntax:

mu [index] value
2dmu [index] value
zoom [index] value
wzoom

- **mu** (magnify up) increases the zoom in one or several views by a factor of 10%.
- **md** (magnify down) decreases the zoom by the inverse factor. **2dmu** and **2dmd** perform the same on one or all 2d views.
- **zoom** and **2dzoom** set the zoom factor to a value specified by you. The current zoom factor is always displayed in the window’s title bar. Zoom 20 represents a full screen view in a large window; zoom 10, a full screen view in a small one.
- **wzoom** (window zoom) allows you to select the area you want to zoom in on with the mouse. You will be prompted to give two of the corners of the area that you want to magnify and the rectangle so defined will occupy the window of the view.

Example:

```
# set a zoom of 2.5
zoom 2.5

# magnify by 10%
mu 1

# magnify by 20%
```

See also: **fit, 2dfit**

**pu, pd, pl, pr, 2dpu, 2dpd, 2dpl, 2dpr**

Syntax:

```
pu [index]
pd [index]
```

The _p_ commands are used to pan. **pu** and **pd** pan up and down respectively; **pl** and **pr** pan to the left and to the right respectively. Each time the view is displaced by 40 pixels. When no index is given, all views will pan in the direction specified.

```
# you have selected one anonometric view
```
See also: **fit, 2dfit**

**fit, 2dfit**

Syntax:

```plaintext
fit [index]
2dfit [index]
```

**fit** computes the best zoom and pans on the content of the view. The content of the view will be centered and fit the whole window.

When fitting all views a unique zoom is computed for all the views. All views are on the same scale.

**Example:**

```plaintext
# fit only view 1
fit 1
# fit all 2d views
2dfit
```

See also: **zoom, mu, pu**

**u, d, l, r**

Syntax:

```plaintext
u [index]
d [index]
l [index]
r [index]
```
**r [index]**

**u, d, l, r** Rotate the object in view around its axis by five degrees up, down, left or right respectively. This command is restricted to axonometric and perspective views.

**Example:**

```
# rotate the view up
u
```

**focal, fu, fd**

Syntax:

```
focal [f]
fu [index]
fd [index]
```

- **focal** changes the vantage point in perspective views. A low f value increases the perspective effect; a high one gives a perspective similar to that of an axonometric view. The default value is 500.
- **fu** and **fd** increase or decrease the focal value by 10%. **fd** makes the eye closer to the object.

**Example:**

```
pers
repeat 10 fd
```

**Note:** Do not use a negative or null focal value.

See also: **pers**

**color**

Syntax:

```
color index name
```
color sets the color to a value. The index of the color is a value between 0 and 15. The name is an X window color name. The list of these can be found in the file rgb.txt in the X library directory.

The default values are: 0 White, 1 Red, 2 Green, 3 Blue, 4 Cyan, 5 Gold, 6 Magenta, 7 Marron, 8 Orange, 9 Pink, 10 Salmon, 11 Violet, 12 Yellow, 13 Khaki, 14 Coral.

Example:

```plaintext
# change the value of blue
color 3 "navy blue"
```

Note that the color change will be visible on the next redraw of the views, for example, after fit or mu, etc.

dtext

Syntax:

```plaintext
dtext [x y [z]] string
```

dtext displays a string in all 3d or 2d views. If no coordinates are given, a graphic selection is required. If two coordinates are given, the text is created in a 2d view at the position specified. With 3 coordinates, the text is created in a 3d view.

The coordinates are real space coordinates.

Example:

```plaintext
# mark the origins
dtext 0 0 bebop
dtext 0 0 0 bebop
```

hardcopy, hcolor, xwd

Syntax:

```plaintext
hardcopy [index]
```
hardcopy creates a postscript file called a4.ps in the current directory. This file contains the postscript description of the view index, and will allow you to print the view.

hcolor lets you change the aspect of lines in the postscript file. It allows to specify a width and a gray level for one of the 16 colors. width is measured in points with default value as 1, gray is the gray level from 0 = black to 1 = white with default value as 0. All colors are bound to the default values at the beginning.

xwd creates an X window xwd file from an active view. By default, the index is set to 1. To visualize an xwd file, use the unix command xwud.

Example:

```
# all blue lines (color 3)
# will be half-width and gray
hcolor 3 0.5

# make a postscript file and print it
hardcopy
lpr a4.ps

# make an xwd file and display it
xwd theview
xwud -in theview
```

Note: When more than one view is present, specify the index of the view. Only use a postscript printer to print postscript files.

See also: color

wclick, pick

Syntax:
**wc**lick defers an event until the mouse button is clicked. The message `just click` is displayed.

Use the **pick** command to get graphic input. The arguments must be names for variables where the results are stored.

- `index`: index of the view where the input was made.
- `X,Y,Z`: 3d coordinates in real world.
- `b`: `b` is the mouse button 1,2 or 3.

When there is an extra argument, its value is not used and the command does not wait for a click; the value of `b` may then be 0 if there has not been a click.

This option is useful for tracking the pointer.

**Note** that the results are stored in Draw numeric variables.

**Example:**

```plaintext
# make a circle at mouse location
pick index x y z b
circle c x y z 0 0 1 1 0 0 0 30

# make a dynamic circle at mouse location
# stop when a button is clicked
# (see the repaint command)

dset b 0
while {[dval b] == 0} {
  pick index x y z b nowait
  circle c x y z 0 0 1 1 0 0 0 30
  repaint
}
```

See also: **repaint**
Draw provides commands to manage the display of objects.

- **display, donly** are used to display,
- **erase, clear, 2dclear** to erase.
- **autodisplay** command is used to check whether variables are displayed when created.

The variable name "." (dot) has a special status in Draw. Any Draw command expecting a Draw object as argument can be passed a dot. The meaning of the dot is the following.

- If the dot is an input argument, a graphic selection will be made. Instead of getting the object from a variable, Draw will ask you to select an object in a view.
- If the dot is an output argument, an unnamed object will be created. Of course this makes sense only for graphic objects: if you create an unnamed number you will not be able to access it. This feature is used when you want to create objects for display only.
- If you do not see what you expected while executing loops or sourcing files, use the **repaint** and **dflush** commands.

**Example:**

```plaintext
# OK use dot to dump an object on the screen dump .
point . x y z

# Not OK. display points on a curve c
# with dot no variables are created
for {set i 0} {$i <= 10} {incr i} {
  cvalue c $i/10 x y z
  point . x y z
}

# point p x y z
# would have displayed only one point
# because the precedent variable content is erased
```
# point p$i x y z
# is an other solution, creating variables
# p0, p1, p2, ....

# give a name to a graphic object
renamevar . x

autodisplay

Syntax:

autodisplay [0/1]

By default, Draw automatically displays any graphic object as soon as it is created. This behavior known as autodisplay can be removed with the command autodisplay. Without arguments, autodisplay toggles the autodisplay mode. The command always returns the current mode.

When autodisplay is off, using the dot return argument is ineffective.

Example:

# c is displayed
circle c 0 0 1 0 5

# toggle the mode
autodisplay
== 0
circle c 0 0 1 0 5

# c is erased, but not displayed
display c

display, donly

Syntax:

display varname [varname ...]
donly varname [varname ...]

- **display** makes objects visible.
- **donly display only** makes objects visible and erases all other objects. It is very useful to extract one object from a messy screen.

**Example:**

```
\# to see all objects
foreach var [directory] {display $var}
```

```
\# to select two objects and erase the other ones
donly . .
```

**erase, clear, 2dclear**

**Syntax:**

```
erase [varname varname ...]
clear
2dclear
```

**erase** removes objects from all views. **erase** without arguments erases everything in 2d and 3d.

**clear** erases only 3d objects and **2dclear** only 2d objects. **erase** without arguments is similar to **clear; 2dclear**.

**Example:**

```
# erase everything with a name starting with c_
foreach var [directory c_*] {erase $var}
```

```
# clear 2d views
2dclear
```

**disp, don, era**

These commands have the same meaning as correspondingly display,
donly and erase, but with the difference that they evaluate the arguments using glob pattern rules. For example, to display all objects with names d_1, d_2, d_3, etc. it is enough to run the command:

\[
\text{disp } d_*
\]

**repa\text{int}, dflush**

Syntax:

```
repa\text{int}
dflush
```

- **repa\text{int** forces repainting of views.
- **dflush** flushes the graphic buffers.

These commands are useful within loops or in scripts.

When an object is modified or erased, the whole view must be repainted. To avoid doing this too many times, Draw sets up a flag and delays the repaint to the end of the command in which the new prompt is issued. In a script, you may want to display the result of a change immediately. If the flag is raised, **repa\text{int** will repaint the views and clear the flag.

Graphic operations are buffered by Draw (and also by the X system). Usually the buffer is flushed at the end of a command and before graphic selection. If you want to flush the buffer from inside a script, use the **dflush** command.

See also: **pick command.**
AIS viewer -- view commands

vinit

Syntax:

```vinit```

Creates a new View window with the specified `view_name`. By default the view is created in the viewer and in the graphic driver shared with the active view.

```name = {driverName/viewerName/viewName | viewerName/viewName | viewName}```

If `driverName` is not specified the driver will be shared with the active view. If `viewerName` is not specified the viewer will be shared with the active view.

vhelp

Syntax:

```vhelp```

Displays help in the 3D viewer window. The help consists in a list of hotkeys and their functionalities.

vtop

Syntax:

```vtop```

Displays top view in the 3D viewer window. Orientation +X+Y.

Example:
vinit
box b 10 10 10
vdisplay b
vfit
vttop

vaxo
Syntax:

vaxo
Displays axonometric view in the 3D viewer window. Orientation +X-Y+Z.

Example:

vinit
box b 10 10 10
vdisplay b
vfit
vaxo

vsetbg
Syntax:

vsetbg imagefile [filltype]
Loads image file as background. filltype must be NONE, CENTERED, TILED or STRETCH.

Example:

vinit
vsetbg myimage.brep CENTERED

vclear
Syntax:
vclear

Removes all objects from the viewer.

vrepaint

Syntax:

vrepaint

Forcibly redisplays the shape in the 3D viewer window.

vfit

Syntax:

vfit

Automatic zoom/panning. Objects in the view are visualized to occupy the maximum surface.

vzfit

Syntax:

vzfit

Automatic depth panning. Objects in the view are visualized to occupy the maximum 3d space.

vreadpixel

Syntax:

vreadpixel xPixel yPixel

[rgb|rgba|depth|hls|rgbf|rgbaf]=rgba] [name]

Read pixel value for active view.
vselect

Syntax:

vselect x1 y1 [x2 y2 [x3 y3 ... xn yn]] [-allowoverlap 0|1] [shift_selection = 0|1]

Emulates different types of selection:

- single mouse click selection
- selection with a rectangle having the upper left and bottom right corners in (x1,y1) and (x2,y2) respectively
- selection with a polygon having the corners in pixel positions (x1,y1), (x2,y2),..., (xn,yn)
- -allowoverlap manages overlap and inclusion detection in rectangular selection. If the flag is set to 1, both sensitives that were included completely and overlapped partially by defined rectangle will be detected, otherwise algorithm will chose only fully included sensitives. Default behavior is to detect only full inclusion.
- any of these selections if shift_selection is set to 1.

vmoveto

Syntax:

vmoveto x y

Emulates cursor movement to pixel position (x,y).

vviewparams

Syntax:

vviewparams [-scale [s]] [-eye [x y z]] [-at [x y z]] [-up [x y z]] [-proj [x y z]] [-center x y] [-size sx]

Gets or sets the current view parameters.

- If called without arguments, all view parameters are printed.
The options are:
- `-scale [s]`: prints or sets the relative scale of viewport.
- `-eye [x y z]`: prints or sets the eye location.
- `-at [x y z]`: prints or sets the view center.
- `-up [x y z]`: prints or sets the up vector direction.
- `-proj [x y z]`: prints or sets the view direction.
- `-center x y`: sets the screen center location in pixels.
- `-size [sx]`: prints viewport projection width and height sizes or changes the size of its maximum dimension.

**vchangeselected**

Syntax:
```
vchangeselected shape
```

Adds a shape to selection or removes one from it.

**vzclipping**

Syntax:
```
vzclipping [mode] [depth width]
```

Gets or sets ZClipping mode, width and depth, where

- `mode = OFF|BACK|FRONT|SLICE`
- `depth` is a real value from segment [0,1]
- `width` is a real value from segment [0,1]

**vnbselected**

Syntax:
```
vnbselected
```

Returns the number of selected objects in the interactive context.

**vpurgedisplay**
Syntax:

vpurgedisplay [CollectorToo = 0|1]

Removes structures which do not belong to objects displayed in neutral point.

vhlr

Syntax:

vhlr is_enabled={on|off} [show_hidden={1|0}]

Hidden line removal algorithm:

- *is_enabled* applies HLR algorithm.
- *show_hidden* if equals to 1, hidden lines are drawn as dotted ones.

vhlrttype

Syntax:

vhlrttype algo_type={algo|polyalgo} [shape_1 ... shape_n]

Changes the type of HLR algorithm used for shapes. If the algo_type is algo, the exact HLR algorithm is used, otherwise the polygonal algorithm is used for defined shapes.

If no shape is specified through the command arguments, the given HLR algorithm_type is applied to all AIS_Shape instances in the current context, and the command also changes the default HLR algorithm type.

**Note** that this command works with instances of AIS_Shape or derived classes only, other interactive object types are ignored.

vcamera

Syntax:
vcamera [-ortho] [-projtype]
  [-persp]
  [-fovy [Angle]] [-distance [Distance]]
  [-stereo] [-leftEye] [-rightEye]
  [-iod [Distance]] [-iodType
  [absolute|relative]]
  [-zfocus [Value]] [-zfocusType
  [absolute|relative]]

Manages camera parameters. Prints the current value when the option is called without argument.

Orthographic camera:

- -ortho – activates orthographic projection.

Perspective camera:

- -persp – activated perspective projection (mono);
- -fovy – field of view in y axis, in degrees;
- -distance – distance of eye from the camera center.

Stereoscopic camera:

- -stereo – perspective projection (stereo);
- -leftEye – perspective projection (left eye);
- -rightEye – perspective projection (right eye);
- -iod – intraocular distance value;
- -iodType – distance type, absolute or relative;
- -zfocus – stereographic focus value;
- -zfocusType – focus type, absolute or relative.

Example:

vinit
box b 10 10 10
vdisplay b
vfit
vcamera -persp
**vstereo**

Syntax:

```
vstereo [0|1] [-mode Mode] [-reverse {0|1}] [-anaglyph Filter]
```

 Defines the stereo output mode. The following modes are available:

- **quadBuffer** – OpenGL QuadBuffer stereo, requires driver support. Should be called BEFORE `vinit`!
- **anaglyph** – Anaglyph glasses;
- **rowInterlaced** – row-interlaced display;
- **columnInterlaced** – column-interlaced display;
- **chessBoard** – chess-board output;
- **sideBySide** – horizontal pair;
- **overUnder** – vertical pair; Available Anaglyph filters for `-anaglyph`:
  - redCyan, redCyanSimple, yellowBlue, yellowBlueSimple, greenMagentaSimple.

**Example:**

```
vinit
box b 10 10 10
vdisplay b
vstereo 1
vfit
vcamera -stereo -iod 1
vcamera -lefteye
vcamera -righteye
```

**vfrustumculling**

Syntax:

```
vfrustumculling [toEnable]
```

Enables/disables objects clipping.
AIS viewer -- display commands

vdisplay

Syntax:

```
vdisplay [-noupdate|-update] [-local] [-mutable] [-neutral]
    [-trsfPers
     {pan|zoom|rotate|trihedron|full|none}=none] [-
     trsfPersPos X Y [Z]] [-3d|-2d|-2dTopDown]
    [-dispMode mode] [-highMode mode]
    [-layer index] [-top|-topmost|-overlay|-underlay]
    [-redisplay]
    name1 [name2] ... [name n]
```

Displays named objects. Option -local enables display of objects in the local selection context. Local selection context will be opened if there is not any.

- **noupdate** suppresses viewer redraw call.
- **mutable** enables optimization for mutable objects.
- **neutral** draws objects in the main viewer.
- **layer** sets z-layer for objects. It can use -overlay|-underlay|-top|-topmost instead of -layer index for the default z-layers.
- **top** draws objects on top of main presentations but below the topmost level.
- **topmost** draws in overlay for 3D presentations with independent Depth.
- **overlay** draws objects in overlay for 2D presentations (On-Screen-Display).
- **underlay** draws objects in underlay for 2D presentations (On-Screen-Display).
- **selectable|-noselect** controls selection of objects.
- **trsfPers** sets transform persistence flags. Flag full allows to pan, zoom and rotate.
- **trsfPersPos** sets an anchor point for transform persistence.
- **2d|-2dTopDown** displays object in screen coordinates.
- **dispmode** sets display mode for objects.
- **highmode** sets highlight mode for objects.
- **redisplay** recomputes presentation of objects.

**Example:**

```plaintext
vinit
box b 40 40 40 10 10 10
psphere s 20
vdisplay s b
vfit
```

**vdisplayall**

Syntax:

```plaintext
vdisplayall [-local]
```

Displays all erased interactive objects (see vdir and vstate). Option `-local` enables displaying objects in the local selection context.
**Example:**

```plaintext
vinit
box b 40 40 40 10 10 10
psphere s 20
vdisplayall
vfit
```

**verase**

Syntax:

```
verase [name1] [name2] ... [name n]
```

Erases some selected or named objects. If there are no selected or named objects, the whole viewer is erased.

**Example:**

```plaintext
vinit
box b1 40 40 40 10 10 10
box b2 -40 -40 -40 10 10 10
psphere s 20
vdisplayall
vfit
# erase only first box
verase b1
# erase second box and sphere
verase
```

**veraseall**

Syntax:

```
veraseall
```

Erases all objects displayed in the viewer.

**Example:**
vinit
box b1 40 40 40 10 10 10
box b2 -40 -40 -40 10 10 10
psphere s 20
vdisplayall
vfit
# erase only first box
verase b1
# erase second box and sphere
verseall

vsetdispmode

Syntax:

vsetdispmode [name] mode(0,1,2,3)

Sets display mode for all, selected or named objects.

- 0 (WireFrame),
- 1 (Shading),
- 2 (Quick HideLineremoval),
- 3 (Exact HideLineremoval).

Example:

vinit
box b 10 10 10
vdisplay b
vsetdispmode 1
vfit

vdisplaytype

Syntax:

vdisplaytype type

Displays all objects of a given type. The following types are possible:
Point, Axis, Trihedron, PlaneTrihedron, Line, Circle, Plane, Shape, ConnectedShape, MultiConn.Shape, ConnectedInter., MultiConn., Constraint and Dimension.

verasetype

Syntax:

```
verasetype type
```

Erases all objects of a given type. Possible type is Point, Axis, Trihedron, PlaneTrihedron, Line, Circle, Plane, Shape, ConnectedShape, MultiConn.Shape, ConnectedInter., MultiConn., Constraint and Dimension.

vtypes

Syntax:

```
vtypes
```

Makes a list of known types and signatures in AIS.

vaspects

Syntax:

```
vaspects [-noupdate|-update] [name1 [name2 [...]] | -defaults] 
 [-setVisibility 0|1] 
 [-setColor ColorName] [-setcolor R G B] [-unsetColor] 
 [-setMaterial MatName] [-unsetMaterial] 
 [-setTransparency Transp] [-unsetTransparency] 
 [-setWidth LineWidth] [-unsetWidth] 
 [-setLineType {solid|dash|dot|dotDash}] [-unsetLineType] 
 [-freeBoundary {off/on | 0/1}]
```
[-setFreeBoundaryWidth Width] [-unsetFreeBoundaryWidth]
[-setFreeBoundaryColor {ColorName | R G B}] [-unsetFreeBoundaryColor]
[-subshapes subname1 [subname2 [...]]]
[-isoontriangulation 0|1]
[-setMaxParamValue {value}]

Manages presentation properties of all, selected or named objects.

- **subshapes** – assigns presentation properties to the specified sub-shapes.
- **defaults** – assigns presentation properties to all objects that do not have their own specified properties and to all objects to be displayed in the future. If **-defaults** option is used there should not be any names of objects and **-subshapes** specifier.

Aliases:

**vsetcolor** [-noupdate|-update] [name] ColorName

Manages presentation properties (color, material, transparency) of all objects, selected or named.

**Color.** The **ColorName** can be: **BLACK, MATRAGRAY, MATRABLUE, ALICEBLUE, ANTIQUEWHITE, ANTIQUEWHITE1, ANTIQUEWHITE2, ANTIQUEWHITE3, ANTIQUEWHITE4, AQUAMARINE1, AQUAMARINE2, AQUAMARINE4, AZURE, AZURE2, AZURE3, AZURE4, BEIGE, BISQUE, BISQUE2, BISQUE3, BISQUE4, BLANCHEDALMOND, BLUE1, BLUE2, BLUE3, BLUE4, BLUEVIOLET, BROWN, BROWN1, BROWN2, BROWN3, BROWN4, BURLYWOOD, BURLYWOOD1, BURLYWOOD2, BURLYWOOD3, BURLYWOOD4, CADETBLUE, CADETBLUE1, CADETBLUE2, CADETBLUE3, CADETBLUE4, CHARTREUSE, CHARTREUSE1, CHARTREUSE2, CHARTREUSE3, CHARTREUSE4, CHOCOLATE, CHOCOLATE1, CHOCOLATE2, CHOCOLATE3, CHOCOLATE4, CORAL, CORAL1, CORAL2, CORAL3, CORAL4, CORNFLOWERBLUE, CORNSILK1, CORNSILK2, CORNSILK3, CORNSILK4, CYAN1, CYAN2, CYAN3, CYAN4, DARKGOLDENROD, DARKGOLDENROD1, DARKGOLDENROD2, DARKGOLDENROD3, DARKGOLDENROD4,
DARKGREEN, DARKKHAKI, DARKOLIVEGREEN, DARKOLIVEGREEN1, DARKOLIVEGREEN2, DARKOLIVEGREEN3, DARKOLIVEGREEN4, DARKORANGE, DARKORANGE1, DARKORANGE2, DARKORANGE3, DARKORANGE4, DARKORCHID, DARKORCHID1, DARKORCHID2, DARKORCHID3, DARKORCHID4, DARKSALMON, DARKSEAGREEN, DARKSEAGREEN1, DARKSEAGREEN2, DARKSEAGREEN3, DARKSEAGREEN4, DARKSLATEBLUE, DARKSLATEGREY1, DARKSLATEGREY2, DARKSLATEGREY3, DARKSLATEGREY4, DARKSLATEGREY, DARKTURQUOISE, DARKVIOLET, DEEPPINK, DEEPPINK2, DEEPPINK3, DEEPPINK4, DEEPSKYBLUE1, DEEPSKYBLUE2, DEEPSKYBLUE3, DEEPSKYBLUE4, DODGERBLUE1, DODGERBLUE2, DODGERBLUE3, DODGERBLUE4, FIREBRICK, FIREBRICK1, FIREBRICK2, FIREBRICK3, FIREBRICK4, FLORALWHITE, FORESTGREEN, GAINSBORO, GHOSTWHITE, GOLD, GOLD1, GOLD2, GOLD3, GOLD4, GOLDENROD, GOLDENROD1, GOLDENROD2, GOLDENROD3, GOLDENROD4, GRAY, GRAY0, GRAY1, GRAY10, GRAY11, GRAY12, GRAY13, GRAY14, GRAY15, GRAY16, GRAY17, GRAY18, GRAY19, GRAY2, GRAY20, GRAY21, GRAY22, GRAY23, GRAY24, GRAY25, GRAY26, GRAY27, GRAY28, GRAY29, GRAY3, GRAY30, GRAY31, GRAY32, GRAY33, GRAY34, GRAY35, GRAY36, GRAY37, GRAY38, GRAY39, GRAY4, GRAY40, GRAY41, GRAY42, GRAY43, GRAY44, GRAY45, GRAY46, GRAY47, GRAY48, GRAY49, GRAY5, GRAY50, GRAY51, GRAY52, GRAY53, GRAY54, GRAY55, GRAY56, GRAY57, GRAY58, GRAY59, GRAY6, GRAY60, GRAY61, GRAY62, GRAY63, GRAY64, GRAY65, GRAY66, GRAY67, GRAY68, GRAY69, GRAY7, GRAY70, GRAY71, GRAY72, GRAY73, GRAY74, GRAY75, GRAY76, GRAY77, GRAY78, GRAY79, GRAY8, GRAY80, GRAY81, GRAY82, GRAY83, GRAY85, GRAY86, GRAY87, GRAY88, GRAY89, GRAY9, GRAY90, GRAY91, GRAY92, GRAY93, GRAY94, GRAY95, GREEN, GREEN1, GREEN2, GREEN3, GREEN4, GREENYELLOW, GRAY97, GRAY98, GRAY99, HONEYDEW, HONEYDEW2, HONEYDEW3, HONEYDEW4, HOTPINK, HOTPINK1, HOTPINK2, HOTPINK3, HOTPINK4, INDIANRED, INDIANRED1, INDIANRED2, INDIANRED3, INDIANRED4, IVORY, IVORY2, IVORY3, IVORY4, KHAKI, KHAKI1, KHAKI2, KHAKI3, KHAKI4, LAVENDER, LAVENDERBLUSH1, LAVENDERBLUSH2, LAVENDERBLUSH3, LAVENDERBLUSH4, LAWNGREEN, LEMONCHIFFON1, LEMONCHIFFON2, LEMONCHIFFON3, LEMONCHIFFON4, LIGHTBLUE, LIGHTBLUE1, LIGHTBLUE2,
SANDYBROWN, SEAGREEN, SEAGREEN1, SEAGREEN2, SEAGREEN3, SEAGREEN4, SEASHELL, SEASHELL2, SEASHELL3, SEASHELL4, BEET, TEAL, SIENNA, SIENNA1, SIENNA2, SIENNA3, SIENNA4, SKYBLUE, SKYBLUE1, SKYBLUE2, SKYBLUE3, SKYBLUE4, SLATEBLUE, SLATEBLUE1, SLATEBLUE2, SLATEBLUE3, SLATEBLUE4, SLATEGREY1, SLATEGREY2, SLATEGREY3, SLATEGREY4, SLATEGREY, SNOW, SNOW2, SNOW3, SNOW4, SPRINGGREEN, SPRINGGREEN2, SPRINGGREEN3, SPRINGGREEN4, STEELBLUE, STEELBLUE1, STEELBLUE2, STEELBLUE3, STEELBLUE4, TAN, TAN1, TAN2, TAN3, TAN4, THISTLE, THISTLE1, THISTLE2, THISTLE3, THISTLE4, TOMATO, TOMATO1, TOMATO2, TOMATO3, TOMATO4, TURQUOISE, TURQUOISE1, TURQUOISE2, TURQUOISE3, TURQUOISE4, VIOLET, VIOLETRED, VIOLETRED1, VIOLETRED2, VIOLETRED3, VIOLETRED4, WHEAT, WHEAT1, WHEAT2, WHEAT3, WHEAT4, WHITE, WHITESMOKE, YELLOW, YELLOW1, YELLOW2, YELLOW3, YELLOW4 and YELLOWGREEN.

```prolog
vaspects [name] [-setcolor ColorName] [-setcolor R G B] [-unsetcolor]
vsetcolor [name] ColorName
v unsetcolor [name]
```

Transparency. The Transp may be between 0.0 (opaque) and 1.0 (fully transparent). **Warning**: at 1.0 the shape becomes invisible.

```prolog
vaspects [name] [-settransparency Transp] [-unsettransparency]
vsettransparency [name] Transp
v unsettransparency [name]
```

Material. The MatName can be BRASS, BRONZE, COPPER, GOLD, PEWTER, PLASTER, PLASTIC, SILVER, STEEL, STONE, SHINY_PLASTIC, SATIN, METALIZED, NEON_GNC, CHROME, ALUMINIUM, OBSIDIAN, NEON_PHC, JADE, WATER, GLASS, DIAMOND or CHARCOAL.

```prolog
vaspects [name] [-setmaterial MatName] [-unsetmaterial]
vsetmaterial [name] MatName
```
Line width. Specifies width of the edges. The LineWidth may be between 0.0 and 10.0.

```
vsetmaterial [name]
```

```
vaspects [name] [-setwidth LineWidth] [-unsetwidth]
vsetwidth [name] LineWidth
vunsetwidth [name]
```

**Example:**

```
vinit
box b 10 10 10
vdisplay b
vfit

vsetdispmode b 1
vaspects -setcolor red -settransparency 0.2
vrotate 10 10 10
```

**vsetshading**

**Syntax:**

```
vsetshading shapename [coefficient]
```

Sets deflection coefficient that defines the quality of the shape's representation in the shading mode. Default coefficient is 0.0008.

**Example:**

```
vinit
psphere s 20
vdisplay s
vfit
vsetdispmode 1
vsetshading s 0.005
```
vunsetshading

Syntax:

vunsetshading [shapename]

Sets default deflection coefficient (0.0008) that defines the quality of the shape’s representation in the shading mode.

vsetam

Syntax:

vsetam [shapename] mode

Activates selection mode for all selected or named shapes:

- 0 for shape itself,
- 1 (vertices),
- 2 (edges),
- 3 (wires),
- 4 (faces),
- 5 (shells),
- 6 (solids),
- 7 (compounds).

Example:

vinit
box b 10 10 10
vdisplay b
vfit
vsetam b 2

vunsetam

Syntax:

vunsetam
Deactivates all selection modes for all shapes.

**vdump**

Syntax:

```
vdump <filename>.{png|bmp|jpg|gif} [-width Width -height Height]
       [-buffer rgb|rgba|depth=rgb]
       [-stereo mono|left|right|blend|sideBySide|overUnder=mono]
```

Extracts the contents of the viewer window to a image file.

**vdir**

Syntax:

```
vdir
```

Displays the list of displayed objects.

**vsub**

Syntax:

```
vsub 0/1(on/off)[shapename]
```

Hilights/unhilights named or selected objects which are displayed at neutral state with subintensity color.

**Example:**

```
vinit
box b 10 10 10
psphere s 20
vdisplay b s
vfit
vsetdispmode 1
```
vsensdis

Syntax:

vsensdis

Displays active entities (sensitive entities of one of the standard types corresponding to active selection modes).

Standard entity types are those defined in Select3D package:

- sensitive box
- sensitive face
- sensitive curve
- sensitive segment
- sensitive circle
- sensitive point
- sensitive triangulation
- sensitive triangle Custom (application-defined) sensitive entity types are not processed by this command.

vsensera

Syntax:

vsensera

Erases active entities.

vr

Syntax:

vr filename

Reads shape from BREP-format file and displays it in the viewer.

Example:
vinit
vr myshape.brep

vstate
Syntax:
```
vstate [-entities] [-hasSelected] [name1] ... [nameN]
```
Reports show/hidden state for selected or named objects:
- **entities** – prints low-level information about detected entities;
- **hasSelected** – prints 1 if the context has a selected shape and 0 otherwise.

vraytrace
Syntax:
```
vraytrace [0/1]
```
Turns on/off ray tracing renderer.

vrenderparams
Syntax:
```
vrenderparams [-rayTrace|-raster] [-rayDepth 0..10]
[-shadows {on|off}]
[-reflections {on|off}] [-fsaa {on|off}]
[-gleam {on|off}]
[-gi {on|off}]
[-brng {on|off}]
[-env {on|off}]
[-shadin {color|flat|gouraud|phong}]
```
Manages rendering parameters:
- **rayTrace** – Enables GPU ray-tracing
- **raster** – Disables GPU ray-tracing
- rayDepth – Defines maximum ray-tracing depth
- shadows – Enables/disables shadows rendering
- reflections – Enables/disables specular reflections
- fsaa – Enables/disables adaptive anti-aliasing
- gleam – Enables/disables transparency shadow effects
- gi – Enables/disables global illumination effects
- brng – Enables/disables blocked RNG (fast coherent PT)
- env – Enables/disables environment map background
- shadingModel – Controls shading model from enumeration color, flat, gouraud, phong

Unlike vcaps, these parameters dramatically change visual properties. The command is intended to control presentation quality depending on hardware capabilities and performance.

Example:

```
vinit
box b 10 10 10
vdisplay b
vfit
vraytrace 1
vrenderparams -shadows 1 -reflections 1 -fsaa 1
```

**vshaderprog**

Syntax:

```
'vshaderprog [name] pathToVertexShader
    pathToFragmentShader'
```

or 'vshaderprog [name] off' to disable GLSL program

or 'vshaderprog [name] phong' to enable per-pixel lighting calculations

Enables rendering using a shader program.

**vsetColorbg**

Syntax:
vsetcolorbg r g b

Sets background color.

**Example:**

```
vinit
vsetcolorbg 200 0 200
```
AIS viewer -- object commands

vtrihedron

Syntax:

```
vtrihedron name [-dispMode {wf|sh|wireframe|shading}] [-origin x y z ] [-zaxis u v w -xaxis u v w ] [-drawaxes {X|Y|Z|XY|YZ|XZ|XYZ}] [-hidelabels {on|off}]" [-label {XAxis|YAxis|ZAxis} value]" [-attribute {XAxisLength|YAxisLength|ZAxisLength|TubeRadiusPercent|ConeRadiusPercent|ConeLengthPercent|OriginRadiusPercent|ShadingNumberOfFacettes} value]" [-color {Origin|XAxis|YAxis|ZAxis|XOYAxis|YOZAxis|XOZAxis|Whole} {r g b | colorName}]" [-textcolor {r g b | colorName}]" [-arrowscolor {r g b | colorName}]" [-priority {Origin|XAxis|YAxis|ZAxis|XArrow|YArrow|XOYAxis|YOZAxis|XOZAxis|Whole} value]
```

Creates a new AIS_Trihedron object or changes existing trihedron. If no argument is set, the default trihedron (0XYZ) is created.
Example:

vinit
vtrihedron tr1

vtrihedron t2 -dispmode shading -origin -200 -200 -300
vtrihedron t2 -color XAxis Quantity_NOC_RED
vtrihedron t2 -color YAxis Quantity_NOC_GREEN
vtrihedron t2 -color ZAxis|Origin Quantity_NOC_BLUE1

vplanetri

Syntax:

vplanetri name

Creates a plane from a trihedron selection. If no arguments are set, the default plane is created.

vsize

Syntax:

vsize [name] [size]

Changes the size of a named or selected trihedron. If the name is not defined: it affects the selected trihedrons otherwise nothing is done. If the value is not defined, it is set to 100 by default.

Example:

vinit
vtrihedron tr1
vtrihedron tr2 0 0 0 1 0 0 1 0 0
vsize tr2 400

vaxis
Syntax:

\texttt{vaxis name [Xa Ya Za Xb Yb Zb]}

Creates an axis. If the values are not defined, an axis is created by interactive selection of two vertices or one edge

\textbf{Example:}

\begin{verbatim}
vinit
vtrihedron tr
vaxis axe1 0 0 0 1 0 0
\end{verbatim}

\textbf{vaxispara}

Syntax:

\texttt{vaxispara name}

Creates an axis by interactive selection of an edge and a vertex.

\textbf{vaxisortho}

Syntax:

\texttt{vaxisortho name}

Creates an axis by interactive selection of an edge and a vertex. The axis will be orthogonal to the selected edge.

\textbf{vpoint}

Syntax:

\texttt{vpoint name [Xa Ya Za]}

Creates a point from coordinates. If the values are not defined, a point is created by interactive selection of a vertex or an edge (in the center of the edge).
**Example:**

```
vinit
vpoint p 0 0 0
```

**vplane**

Syntax:

```
vplane name [AxisName] [PointName]
vplane name [PointName] [PointName] [PointName]
vplane name [PlaneName] [PointName]
```

Creates a plane from named or interactively selected entities.

**TypeOfSensitivity:**

- 0 – Interior
- 1 – Boundary

**Example:**

```
vinit
vpoint p1 0 50 0
vaxis axe1 0 0 0 0 0 1
vtrihedron tr
vplane plane1 axe1 p1
```

**vplanepara**

Syntax:

```
vplanepara name
```

Creates a plane from interactively selected vertex and face.

**vplaneortho**

Syntax:
vplaneortho name

Creates a plane from interactive selected face and coplanar edge.

vline

Syntax:

vline name [PointName] [PointName]
vline name [Xa Ya Za Xb Yb Zb]

Creates a line from coordinates, named or interactively selected vertices.

Example:

vinit
vtrihedron tr
vpoint p1 0 50 0
vpoint p2 50 0 0
vline line1 p1 p2
vline line2 0 0 0 50 0 1

vcircle

Syntax:

vcircle name [PointName PointName PointName IsFilled]
vcircle name [PlaneName PointName Radius IsFilled]

Creates a circle from named or interactively selected entities. Parameter IsFilled is defined as 0 or 1.

Example:

vinit
vtrihedron tr
vpoint p1 0 50 0
vpoint p2 50 0 0
vpoint p3 0 0 0
vcircle circle1 p1 p2 p3 1

vtri2d

Syntax:

vtri2d name

Creates a plane with a 2D trihedron from an interactively selected face.

vselmode

Syntax:

vselmode [object] mode_number is_turned_on=(1|0)

Sets the selection mode for an object. If the object value is not defined, the selection mode is set for all displayed objects. Mode_number is a non-negative integer encoding different interactive object classes. For shapes the following mode_number values are allowed:

- 0 – shape
- 1 – vertex
- 2 – edge
- 3 – wire
- 4 – face
- 5 – shell
- 6 – solid
- 7 – compsolid
- 8 – compound is_turned_on is:
  - 1 if mode is to be switched on
  - 0 if mode is to be switched off

Example:

vinit
vpoint p1 0 0 0
vpoint p2 50 0 0
vpoint p3 25 40 0
vtriangle triangle1 p1 p2 p3
**vconnect**

Syntax:

```
vconnect vconnect name Xo Yo Zo object1 object2 ...
   [color=NAME]
```

Creates *AIS_ConnectedInteractive* object from the input object and location and displays it.

**Example:**

```
vinit
vpoint p1 0 0 0
vpoint p2 50 0 0
vsegment segment p1 p2
restore CrankArm.brep obj
vdisplay obj
vconnect new obj 100100100 1 0 0 0 0 1
```

**vtriangle**

Syntax:

```
vtriangle name PointName PointName PointName
```

Creates and displays a filled triangle from named points.

**Example:**

```
vinit
vpoint p1 0 0 0
vpoint p2 50 0 0
vpoint p3 25 40 0
vtriangle triangle1 p1 p2 p3
```
Syntax:

vsegment name PointName PointName

Creates and displays a segment from named points.

Example:

Vinit
vpoint p1 0 0 0
vpoint p2 50 0 0
vsegment segment p1 p2

vpointcloud

Syntax:

vpointcloud name shape [-randColor] [-normals] [-noNormals]

Creates an interactive object for an arbitrary set of points from the triangulated shape. Additional options:

- **randColor** – generates a random color per point;
- **normals** – generates a normal per point (default);
- **noNormals** – does not generate a normal per point.

vpointcloud name x y z r npts {surface|volume} [-randColor] [-normals] [-noNormals]

Creates an arbitrary set of points (npts) randomly distributed on a spheric surface or within a spheric volume (x y z r). Additional options:

- **randColor** – generates a random color per point;
- **normals** – generates a normal per point (default);
- **noNormals** – does not generate a normal per point.

Example:

Vinit
vpointcloud pc 0 0 0 100 100000 surface -randColor
tf

vclipplane

Syntax:

vclipplane maxplanes <view_name> -- gets plane limit for the view.
vclipplane create <plane_name> -- creates a new plane.
vclipplane delete <plane_name> -- deletes a plane.
vclipplane clone <source_plane> <plane_name> -- clones the plane definition.
vclipplane set/unset <plane_name> object <object list> -- sets/unsets the plane for an IO.
vclipplane set/unset <plane_name> view <view list> -- sets/unsets plane for a view.
vclipplane change <plane_name> on/off -- turns clipping on/off.
vclipplane change <plane_name> equation <a> <b> <c> <d> -- changes plane equation.
vclipplane change <plane_name> capping on/off -- turns capping on/off.
vclipplane change <plane_name> capping color <r> <g> <b> -- sets color.
vclipplane change <plane_name> capping texname <texture> -- sets texture.
vclipplane change <plane_name> capping texscale <sx> <sy> -- sets texture scale.
vclipplane change <plane_name> capping texorigin <tx> <ty> -- sets texture origin.
vclipplane change <plane_name> capping texrotate <angle> -- sets texture rotation.
vclipplane change <plane_name> capping hatch on/off/<id> -- sets hatching mask.
Manages clipping planes

**Example:**

```plaintext
vinit
vclipplane create pln1
vclipplane change pln1 equation 1 0 0 -0.1
vclipplane set pln1 view Driver1/Viewer1/View1
box b 100 100 100
vdisplay b
vsetdispmode 1
vfit
vrotate 10 10 10
vselect 100 100
```

**vdimension**

Syntax:

```plaintext
vdimension name { -angle|-length|-radius|-diameter } -shapes shape1 [ shape2 [ shape3 ] ]
   { -text 3d|2d wf|sh|wireframe|shading IntegerSize }
   { -label left|right|hcenter|hfit top|bottom|vcenter|vfit }
   { -arrow external|internal|fit } [{ -arrowlength|-arlen } RealArrowLength]
   { -arrowangle|-arangle } ArrowAngle(degrees)
   { -plane xoy|yoz|zox } FloatValue
   { -flyout FloatValue -extension }
   { -autovalue } [-value CustomRealValue]
   [-textvalue CustomTextValue]
   [ -dispunits DisplayUnitsString ]
   [ -modelunits ModelUnitsString ] [-showunits | -hideunits]
```

Builds angle, length, radius or diameter dimension interactive object
name.

**Attention:** length dimension can't be built without working plane.

**Example:**

```
vinit
vpoint p1 0 0 0
vpoint p2 50 50 0
vdimension dim1 -length -plane xoy -shapes p1 p2
vpoint p3 100 0 0
vdimension dim2 -angle -shapes p1 p2 p3
vcircle circle p1 p2 p3 0
vdimension dim3 -radius -shapes circle
vfit
```

**vdimparam**

Syntax:

```
vdimparam name [-text 3d|2d wf|sh|wireframe|shading IntegerSize]
   [-label left|right|hcenter|hfit top|bottom|vcenter|vfit]
   [-arrow external|internal|fit]
   [{-arrowlength|-arlen} RealArrowLength]
   [{-arrowangle|-arangle} ArrowAngle(degrees)]
   [-plane xoy|yoz|zox]
   [-flyout FloatValue -extension FloatValue]
   [-autovalue]
   [-value CustomRealValue]
   [-textvalue CustomTextValue]
   [-dispunits DisplayUnitsString]
```
Sets parameters for angle, length, radius and diameter dimension **name**.

**Example:**

```plaintext
vinit
vpoint p1 0 0 0
vpoint p2 50 50 0
vdimension dim1 -length -plane xoy -shapes p1 p2
vdimparam dim1 -flyout -15 -arrowlength 4 -showunits -value 10
vfit
vdimparam dim1 -textvalue "w_1"
vdimparam dim1 -autovalue
```

**vangleparam**

Syntax:

```plaintext
vangleparam name [-type interior|exterior]
                  [-showarrow first|second|both|none]
```

Sets parameters for angle dimension **name**.

**Example:**

```plaintext
vinit
vpoint p1 0 0 0
vpoint p2 10 0 0
vpoint p3 10 5 0
vdimension dim1 -angle -plane xoy -shapes p1 p2 p3
vfit
vangleparam dim1 -type exterior -showarrow first
```

**vlengthparam**
Syntax:

vlengthparam name [-type interior|exterior]
                [-showarrow first|second|both|none]

Sets parameters for length dimension **name**.

**Example:**

```plaintext
vinit
vpoint p1 20 20 0
vpoint p2 80 80 0
vdimension dim1 -length -plane xoy -shapes p1 p2
vtop
vfit
vzoom 0.5
vlengthparam dim1 -direction ox
```

**vmovedim**

Syntax:

vmovedim [name] [x y z]

Moves picked or named (if **name** parameter is defined) dimension to picked mouse position or input point with coordinates **x,y,z**. Text label of dimension **name** is moved to position, another parts of dimension are adjusted.

**Example:**

```plaintext
vinit
vpoint p1 0 0 0
vpoint p2 50 50 0
vdimension dim1 -length -plane xoy -shapes p1 p2
vmovedim dim1 -10 30 0
```
AIS viewer -- Mesh Visualization Service

**MeshVS (Mesh Visualization Service)** component provides flexible means of displaying meshes with associated pre- and post-processor data.

**meshfromstl**

Syntax:

```
meshfromstl meshname file
```

Creates a *MeshVS_Mesh* object based on STL file data. The object will be displayed immediately.

**Example:**

```
meshfromstl mesh myfile.stl
```

**meshdispmode**

Syntax:

```
meshdispmode meshname displaymode
```

Changes the display mode of object *meshname*. The **displaymode** is integer, which can be:

- 1 for *wireframe*,
- 2 for *shading* mode, or
- 3 for *shrink* mode.

**Example:**

```
vinit
meshfromstl mesh myfile.stl
meshdispmode mesh 2
```
**meshselmode**

Syntax:

```plaintext
cmd/meshselmode meshname selectionmode
```

Changes the selection mode of object **meshname**. The *selectionmode* is an integer OR-combination of mode flags. The basic flags are the following:

- 1 – node selection;
- 2 – 0D elements (not supported in STL);
- 4 – links (not supported in STL);
- 8 – faces.

**Example:**

```plaintext
vinit
meshfromstl mesh myfile.stl
meshselmode mesh 1
```

**meshshadcolor**

Syntax:

```plaintext
cmd/meshshadcolor meshname red green blue
```

Changes the face interior color of object **meshname**. The *red*, *green* and *blue* are real values between 0 and 1.

**Example:**

```plaintext
evinit
meshfromstl mesh myfile.stl
meshshadcolormode mesh 0.5 0.5 0.5
```

**meshlinkcolor**

Syntax:
meshlinkcolor meshname red green blue

Changes the color of face borders for object meshname. The red, green and blue are real values between 0 and 1.

Example:

vinit
meshfromstl mesh myfile.stl
meshlinkcolormode mesh 0.5 0.5 0.5

meshmat

Syntax:

meshmat meshname material

Changes the material of object meshname.

material is represented with an integer value as follows (equivalent to enumeration Graphic3d_NameOfMaterial):

- 0 – BRASS,
- 1 – BRONZE,
- 2 – COPPER,
- 3 – GOLD,
- 4 – PEWTER,
- 5 – PLASTER,
- 6 – PLASTIC,
- 7 – SILVER,
- 8 – STEEL,
- 9 – STONE,
- 10 – SHINY_PLASTIC,
- 11 – SATIN,
- 12 – METALIZED,
- 13 – NEON_GNC,
- 14 – CHROME,
- 15 – ALUMINIUM,
- 16 – OBSIDIAN,
- 17 – NEON_PHC,
- 18 – JADE,
- 19 – DEFAULT,
- 20 – UserDefined

**Example:**

```plaintext
vinit
meshfromstl mesh myfile.stl
meshmat mesh JADE
```

**meshshrcoef**

Syntax:

```plaintext
meshshrcoef meshname shrinkcoefficient
```

Changes the value of shrink coefficient used in the shrink mode. In the shrink mode the face is shown as a congruent part of a usual face, so that *shrinkcoefficient* controls the value of this part. The *shrinkcoefficient* is a positive real number.

**Example:**

```plaintext
vinit
meshfromstl mesh myfile.stl
meshshrcoef mesh 0.05
```

**meshshow**

Syntax:

```plaintext
meshshow meshname
```

Displays *meshname* in the viewer (if it is erased).

**Example:**

```plaintext
vinit
meshfromstl mesh myfile.stl
```
**meshhide**

Syntax:

```
meshhide meshname
```

Hides **meshname** in the viewer.

**Example:**

```
vinit
meshfromstl mesh myfile.stl
meshhide mesh
```

**meshhidesel**

Syntax:

```
meshhidesel meshname
```

Hides only selected entities. The other part of **meshname** remains visible.

**meshshowsel**

Syntax:

```
meshshowsel meshname
```

Shows only selected entities. The other part of **meshname** becomes invisible.

**meshshowall**

Syntax:

```
meshshowall meshname
```
Changes the state of all entities to visible for \texttt{meshname}.

\textbf{meshdelete}

Syntax:

\verbatim
meshdelete meshname
\endverbatim

Deletes MeshVS\_Mesh object \texttt{meshname}.

\textbf{Example:}

\verbatim
vinit
meshfromstl mesh myfile.stl
meshdelete mesh
\endverbatim
VIS Viewer commands

A specific plugin with alias VIS should be loaded to have access to VIS functionality in DRAW Test Harness:

> pload VIS

**ivtkinit**

Syntax:

ivtkinit

Creates a window for VTK viewer.

**ivtkdisplay**

Syntax:

ivtkdisplay name1 [name2] ...[name n]

Displays named objects.

Example:

ivtkinit
# create cone
pcone c 5 0 10
ivtkdisplay c

ivtkerase

Syntax:

ivtkerase [name1] [name2] ... [name n]

Erases named objects. If no arguments are passed, erases all displayed objects.

Example:

ivtkinit
# create a sphere
psphere s 10
# create a cone
pcone c 5 0 10
# create a cylinder
pcylinder cy 5 10
# display objects
ivtkdisplay s c cy
# erase only the cylinder
ivtkerase cy
```plaintext
## erase the sphere and the cone
ivtkerase s c

### ivtkfit

Syntax:

```plaintext
ivtkfit
```

Automatic zoom/panning.

### ivtkdispmode

Syntax:

```plaintext
ivtksetdispmode [name] {0|1}
```

Sets display mode for a named object. If no arguments are passed, sets the given display mode for all displayed objects. The possible modes are: 0 (WireFrame) and 1 (Shading).

### Example:

```plaintext
ivtkinit
# create a cone
pccone c 5 0 10
# display the cone
ivtkdisplay c
# set shading mode for the cone
ivtksetdispmode c 1
```
ivtksetselmode

Syntax:

```
ivtksetselmode [name] mode {0|1}
```

Sets selection mode for a named object. If no arguments are passed, sets the given selection mode for all the displayed objects.

**Example:**

```
ivtkinit
# load a shape from file
restore CrankArm.brep a
# display the loaded shape
ivtkdisplay a
# set the face selection mode
ivtksetselmode a 4 1
```
ivtkmoveto

Syntax:

`ivtkmoveto x y`

Imitates mouse cursor moving to point with the given display coordinates x,y.

Example:

```
ivtkinit
pcone c 5 0 10
ivtkdisplay c
ivtkmoveto 40 50
```

ivtkselect

Syntax:

`ivtkselect x y`

Imitates mouse cursor moving to point with the given display coordinates and performs selection at this point.
**Example:**

```
ivtkinit
pcone c 5 0 10
ivtkdisplay c
ivtkselect 40 50
```

**ivtkdump**

**Syntax:**

```
ivtkdump *filename* [buffer={rgb|rgba|depth}] [width height] [stereoproj={L|R}]
```

Dumps the contents of VTK viewer to image. It supports:

- dumping in different raster graphics formats: PNG, BMP, JPEG, TIFF or PNM.
- dumping of different buffers: RGB, RGBA or depth buffer.
- defining of image sizes (width and height in pixels).
- dumping of stereo projections (left or right).

**Example:**

```
ivtkinit
pcone c 5 0 10
ivtkdisplay c
ivtkdump D:/ConeSnapshot.png rgb 768 768
```

**ivtkbgcolor**

**Syntax:**

```
ivtkbgcolor r g b [r2 g2 b2]
```

Sets uniform background color or gradient background if second triple of parameters is set. Color parameters r,g,b have to be chosen in the interval [0..255].
Example:

ivtkinit
ivtkbgcolor 200 220 250

![Image 1](image1)

ivtkbgcolor 10 30 80 255 255 255

![Image 2](image2)
OCAF commands

This chapter contains a set of commands for Open CASCADE Technology Application Framework (OCAF).
Application commands

NewDocument

Syntax:

NewDocument docname [format]

Creates a new docname document with MDTV-Standard or described format.

Example:

# Create new document with default (MDTV-Standard) format
NewDocument D

# Create new document with BinOcaf format
NewDocument D2 BinOcaf

IsInSession

Syntax:

IsInSession path

Returns 0, if path document is managed by the application session, 1 – otherwise.

Example:

IsInSession /myPath/myFile.std

ListDocuments

Syntax:
ListDocuments

Makes a list of documents handled during the session of the application.

**Open**

Syntax:

```
Open path docname [-stream]
```

Retrieves the document of file `docname` in the path `path`. Overwrites the document, if it is already in session.

Option `-stream` activates usage of alternative interface of OCAF persistence working with C++ streams instead of file names.

**Example:**

```
Open /myPath/myFile.std D
```

**Close**

Syntax:

```
Close docname
```

Closes `docname` document. The document is no longer handled by the applicative session.

**Example:**

```
Close D
```

**Save**

Syntax:

```
Save docname
```

Saves `docname` active document.
Example:

Save D

SaveAs

Syntax:

SaveAs docname path [-stream]

Saves the active document in the file docname in the path path. Overwrites the file if it already exists.

option -stream activates usage of alternative interface of OCAF persistence working with C++ streams instead of file names.

Example:

SaveAs D /myPath/myFile.std
Basic commands

Label

Syntax:

Label docname entry

Creates the label expressed by <entry> if it does not exist.

Example

Label D 0:2

NewChild

Syntax:

NewChild docname [taggerlabel = Root label]

Finds (or creates) a TagSource attribute located at father label of <taggerlabel> and makes a new child label.

Example

# Create new child of root label
NewChild D

# Create new child of existing label
Label D 0:2
NewChild D 0:2

Children

Syntax:

Children docname label
Returns the list of attributes of label.

Example

Children D 0:2

**ForgetAll**

Syntax:

`ForgetAll docname label`

Forgets all attributes of the label.

Example

`ForgetAll D 0:2`

**Application commands**

**Main**

Syntax:

`Main docname`

Returns the main label of the framework.

**Example:**

`Main D`

**UndoLimit**

Syntax:

`UndoLimit docname [value=0]`

Sets the limit on the number of Undo Delta stored. 0 will disable Undo on
the document. A negative value means that there is no limit. Note that by default Undo is disabled. Enabling it will take effect with the next call to NewCommand. Of course, this limit is the same for Redo

**Example:**

```
UndoLimit D 100
```

**Undo**

Syntax:

```
Undo docname [value=1]
```

Undoes value steps.

**Example:**

```
Undo D
```

**Redo**

Syntax:

```
Redo docname [value=1]
```

Redoes value steps.

**Example:**

```
Redo D
```

**OpenCommand**

Syntax:

```
OpenCommand docname
```

Opens a new command transaction.
Example:

OpenCommand  D

CommitCommand

Syntax:

CommitCommand  docname

Commits the Command transaction.

Example:

CommitCommand  D

NewCommand

Syntax:

NewCommand  docname

This is a shortcut for Commit and Open transaction.

Example:

NewCommand  D

AbortCommand

Syntax:

AbortCommand  docname

Aborts the Command transaction.

Example:

AbortCommand  D
Copy

Syntax:

**Copy** docname entry Xdocname Xentry

Copies the contents of *entry* to *Xentry*. No links are registered.

**Example:**

```
Copy D1 0:2 D2 0:4
```

UpdateLink

Syntax:

**UpdateLink** docname [entry]

Updates external reference set at *entry*.

**Example:**

```
UpdateLink D
```

CopyWithLink

Syntax:

**CopyWithLink** docname entry Xdocname Xentry

Aborts the Command transaction. Copies the content of *entry* to *Xentry*. The link is registered with an *Xlink* attribute at *Xentry* label.

**Example:**

```
CopyWithLink D1 0:2 D2 0:4
```

UpdateXLinks
Syntax:

```
UpdateXLinks docname entry
```

Sets modifications on labels impacted by external references to the `entry`. The `document` becomes invalid and must be recomputed.

**Example:**

```
UpdateXLinks D 0:2
```

**DumpDocument**

Syntax:

```
DumpDocument docname
```

Displays parameters of `docname` document.

**Example:**

```
DumpDocument D
```
Data Framework commands

MakeDF

Syntax:

```
MakeDF dfname
```

Creates a new data framework.

Example:

```
MakeDF D
```

ClearDF

Syntax:

```
ClearDF dfname
```

Clears a data framework.

Example:

```
ClearDF D
```

CopyDF

Syntax:

```
CopyDF dfname1 entry1 [dfname2] entry2
```

Copies a data framework.

Example:

```
CopyDF D 0:2 0:4
```
CopyLabel

Syntax:

CopyLabel dfname fromlabel tolabel

Copies a label.

Example:

CopyLabel D1 0:2 0:4

MiniDumpDF

Syntax:

MiniDumpDF dfname

Makes a mini-dump of a data framework.

Example:

MiniDumpDF D

XDumpDF

Syntax:

XDumpDF dfname

Makes an extended dump of a data framework.

Example:

XDumpDF D
General attributes commands

**SetInteger**

Syntax:

```
SetInteger dfname entry value
```

Finds or creates an Integer attribute at `entry` label and sets `value`.

**Example:**

```
SetInteger D 0:2 100
```

**GetInteger**

Syntax:

```
GetInteger dfname entry [drawname]
```

Gets a value of an Integer attribute at `entry` label and sets it to `drawname` variable, if it is defined.

**Example:**

```
GetInteger D 0:2 Int1
```

**SetReal**

Syntax:

```
SetReal dfname entry value
```

Finds or creates a Real attribute at `entry` label and sets `value`.

**Example:**

```
SetReal D 0:2 100.
```
GetReal

Syntax:

```
GetReal dfname entry [drawname]
```

Gets a value of a Real attribute at `entry` label and sets it to `drawname` variable, if it is defined.

**Example:**

```
GetReal D 0:2 Real1
```

SetIntArray

Syntax:

```
SetIntArray dfname entry lower upper value1 value2 ...
```

Finds or creates an IntegerArray attribute at `entry` label with lower and upper bounds and sets `**value1**, value2...`

**Example:**

```
SetIntArray D 0:2 1 4 100 200 300 400
```

GetIntArray

Syntax:

```
GetIntArray dfname entry
```

Gets a value of an IntegerArray attribute at `entry` label.

**Example:**

```
GetIntArray D 0:2
```

SetRealArray
Syntax:

SetRealArray dfname entry lower upper value1 value2 ...

Finds or creates a RealArray attribute at entry label with lower and upper bounds and sets value1, *value2*...

Example:

GetRealArray D 0:2 1 4 100. 200. 300. 400.

GetRealArray

Syntax:

GetRealArray dfname entry

Gets a value of a RealArray attribute at entry label.

Example:

GetRealArray D 0:2

SetComment

Syntax:

SetComment dfname entry value

Finds or creates a Comment attribute at entry label and sets value.

Example:

SetComment D 0:2 "My comment"

GetComment

Syntax:

GetComment dfname entry
Gets a value of a Comment attribute at entry label.

**Example:**

GetComment D 0:2

**SetExtStringArray**

Syntax:

\[
\text{SetExtStringArray dfname entry lower upper value1 value2 ...}
\]

Finds or creates an ExtStringArray attribute at entry label with lower and upper bounds and sets value1, *value2*...

**Example:**

SetExtStringArray D 0:2 1 3 *string1* *string2* *string3*

**GetExtStringArray**

Syntax:

\[
\text{GetExtStringArray dfname entry}
\]

Gets a value of an ExtStringArray attribute at entry label.

**Example:**

GetExtStringArray D 0:2

**SetName**

Syntax:

\[
\text{SetName dfname entry value}
\]
Finds or creates a Name attribute at *entry* label and sets *value*.

**Example:**

SetName D 0:2 *My name*

### GetName

Syntax:

```
GetName dfname entry
```

Gets a value of a Name attribute at *entry* label.

**Example:**

GetName D 0:2

### SetReference

Syntax:

```
SetReference dfname entry reference
```

Creates a Reference attribute at *entry* label and sets *reference*.

**Example:**

SetReference D 0:2 0:4

### GetReference

Syntax:

```
GetReference dfname entry
```

Gets a value of a Reference attribute at *entry* label.

**Example:**
**GetReference D 0:2**

**SetUAttribute**

Syntax:

```
SetUAttribute dfname entry localGUID
```

Creates a UAttribute attribute at *entry* label with *localGUID*.

**Example:**

```
set localGUID "c73bd076-22ee-11d2-acde-080009dc4422"
SetUAttribute D 0:2 ${localGUID}
```

**GetUAttribute**

Syntax:

```
GetUAttribute dfname entry localGUID
```

Finds a UAttribute at *entry* label with *localGUID*.

**Example:**

```
set localGUID "c73bd076-22ee-11d2-acde-080009dc4422"
GetUAttribute D 0:2 ${localGUID}
```

**SetFunction**

Syntax:

```
SetFunction dfname entry ID failure
```

Finds or creates a Function attribute at *entry* label with driver ID and *failure* index.

**Example:**
GetFunction

Syntax:

GetFunction dfname entry ID failure

Finds a Function attribute at entry label and sets driver ID to ID variable and failure index to failure variable.

Example:

GetFunction D 0:2 ID failure

NewShape

Syntax:

NewShape dfname entry [shape]

Finds or creates a Shape attribute at entry label. Creates or updates the associated NamedShape attribute by shape if shape is defined.

Example:

box b 10 10 10
NewShape D 0:2 b

SetShape

Syntax:

SetShape dfname entry shape

Creates or updates a NamedShape attribute at entry label by shape.

Example:
GetShape

Syntax:

GetShape2 dfname entry shape

Sets a shape from NamedShape attribute associated with entry label to shape draw variable.

Example:

GetShape2 D 0:2 b
Geometric attributes commands

**SetPoint**

Syntax:

```
SetPoint dfname entry point
```

Finds or creates a Point attribute at `entry` label and sets `point` as generated in the associated `NamedShape` attribute.

**Example:**

```
point p 10 10 10
SetPoint D 0:2 p
```

**GetPoint**

Syntax:

```
GetPoint dfname entry [drawname]
```

Gets a vertex from `NamedShape` attribute at `entry` label and sets it to `drawname` variable, if it is defined.

**Example:**

```
GetPoint D 0:2 p
```

**SetAxis**

Syntax:

```
SetAxis dfname entry axis
```

Finds or creates an Axis attribute at `entry` label and sets `axis` as generated in the associated `NamedShape` attribute.
Example:

```
line l 10 20 30 100 200 300
SetAxis D 0:2 l
```

GetAxis

Syntax:

```
GetAxis dfname entry [drawname]
```

Gets a line from *NamedShape* attribute at *entry* label and sets it to *drawname* variable, if it is defined.

Example:

```
GetAxis D 0:2 l
```

SetPlane

Syntax:

```
SetPlane dfname entry plane
```

Finds or creates a Plane attribute at *entry* label and sets *plane* as generated in the associated *NamedShape* attribute.

Example:

```
plane pl 10 20 30 -1 0 0
SetPlane D 0:2 pl
```

GetPlane

Syntax:

```
GetPlane dfname entry [drawname]
```

Gets a plane from *NamedShape* attribute at *entry* label and sets it to
*drawname* variable, if it is defined.

**Example:**

\[
\text{GetPlane D 0:2 pl}
\]

**SetGeometry**

Syntax:

\[
\text{SetGeometry dfname entry [type] [shape]}
\]

Creates a Geometry attribute at *entry* label and sets *type* and *shape* as generated in the associated *NamedShape* attribute if they are defined. *type* must be one of the following: *any*, *pnt*, *lin*, *cir*, *ell*, *spl*, *pln*, *cyl*.

**Example:**

\[
\begin{align*}
\text{point p 10 10 10} \\
\text{SetGeometry D 0:2 pnt p}
\end{align*}
\]

**GetGeometryType**

Syntax:

\[
\text{GetGeometryType dfname entry}
\]

Gets a geometry type from Geometry attribute at *entry* label.

**Example:**

\[
\text{GetGeometryType D 0:2}
\]

**SetConstraint**

Syntax:

\[
\text{SetConstraint dfname entry keyword geometrie} \\
\quad \text{[geometrie ...]}
\]
SetConstraint dfname entry "plane" geometrie
SetConstraint dfname entry "value" value

1. Creates a Constraint attribute at entry label and sets keyword constraint between geometry(ies). keyword must be one of the following: rad, dia, minr, majr, tan, par, perp, concentric, equal, dist, angle, equad, symm, midp, eqdist, fix, rigid, or from, axis, mate, alignf, aligna, axesa, facesa, round, offset
2. Sets plane for the existing constraint.
3. Sets value for the existing constraint.

Example:

SetConstraint D 0:2 "value" 5

GetConstraint

Syntax:

GetConstraint dfname entry

Dumps a Constraint attribute at entry label

Example:

GetConstraint D 0:2

SetVariable

Syntax:

SetVariable dfname entry isconstant(0/1) units

Creates a Variable attribute at entry label and sets isconstant flag and units as a string.

Example:

SetVariable D 0:2 1 "mm"
GetVariable

Syntax:

```
GetVariable dfname entry isconstant units
```

Gets an `isconstant` flag and units of a Variable attribute at `entry` label.

**Example:**

```
GetVariable D 0:2 isconstant units
puts "IsConstant=${isconstant}"
puts "Units=${units}" 
```
Tree attributes commands

RootNode

Syntax:

RootNode dfname treenodeentry [ID]

Returns the ultimate father of *TreeNode* attribute identified by its *treenodeentry* and its *ID* (or default *ID*, if *ID* is not defined).

SetNode

Syntax:

SetNode dfname treenodeentry [ID]

Creates a *TreeNode* attribute on the *treenodeentry* label with its tree *ID* (or assigns a default *ID*, if the *ID* is not defined).

AppendNode

Syntax:

AppendNode dfname fatherentry childentry [fatherID]

Inserts a *TreeNode* attribute with its tree *fatherID* (or default *ID*, if *fatherID* is not defined) on *childentry* as last child of *fatherentry*.

PrependNode

Syntax:

PrependNode dfname fatherentry childentry [fatherID]

Inserts a *TreeNode* attribute with its tree *fatherID* (or default *ID*, if *fatherID* is not defined) on *childentry* as first child of *fatherentry*. 
**InsertNodeBefore**

Syntax:

```
InsertNodeBefore dfname treenodeentry beforetreenode [ID]
```

Inserts a *TreeNode* attribute with tree ID (or default ID, if ID is not defined) *beforetreenode* before *treenodeentry*.

**InsertNodeAfter**

Syntax:

```
InsertNodeAfter dfname treenodeentry aftertreenode [ID]
```

Inserts a *TreeNode* attribute with tree ID (or default ID, if ID is not defined) *aftertreenode* after *treenodeentry*.

**DetachNode**

Syntax:

```
DetachNode dfname treenodeentry [ID]
```

Removes a *TreeNode* attribute with tree ID (or default ID, if ID is not defined) from *treenodeentry*.

**ChildNodeIterate**

Syntax:

```
ChildNodeIterate dfname treenodeentry alllevels(0/1) [ID]
```

Iterates on the tree of *TreeNode* attributes with tree ID (or default ID, if ID is not defined). If *alllevels* is set to 1 it explores not only the first, but all the sub Step levels.
Example:

Label D 0:2
Label D 0:3
Label D 0:4
Label D 0:5
Label D 0:6
Label D 0:7
Label D 0:8
Label D 0:9

# Set root node
SetNode D 0:2

AppendNode D 0:2 0:4
AppendNode D 0:2 0:5
PrependNode D 0:4 0:3
PrependNode D 0:4 0:8
PrependNode D 0:4 0:9

InsertNodeBefore D 0:5 0:6
InsertNodeAfter D 0:4 0:7

DetachNode D 0:8

# List all levels
ChildNodeIterate D 0:2 1

==0:4
==0:9
==0:3
==0:7
==0:6
==0:5
# List only first levels
ChildNodeIterate D 0:2 1

==0:4
==0:7
==0:6
==0:5

**InitChildNodeIterator**

Syntax:

```plaintext
InitChildNodeIterator dfname treenodeentry
  alllevels(0/1) [ID]
```

Initializes the iteration on the tree of *TreeNode* attributes with tree ID (or default ID, if ID is not defined). If `alllevels` is set to 1 it explores not only the first, but also all sub Step levels.

**Example:**

```plaintext
InitChildNodeIterate D 0:5 1
set aChildNumber 0
for {set i 1} {$i < 100} {incr i} {
  if {[ChildNodeMore] == *TRUE*} {
    puts *Tree node = [ChildNodeValue]*
    incr aChildNumber
    ChildNodeNext
  }
}
puts "aChildNumber=$aChildNumber"
```

**ChildNodeMore**

Syntax:

```plaintext
ChildNodeMore
```
Returns TRUE if there is a current item in the iteration.

**ChildNodeNext**

Syntax:

ChildNodeNext

Moves to the next Item.

**ChildNodeValue**

Syntax:

ChildNodeValue

Returns the current treenode of *ChildNodeIterator*.

**ChildNodeNextBrother**

Syntax:

ChildNodeNextBrother

Moves to the next *Brother*. If there is none, goes up. This method is interesting only with *allLevels* behavior.
Standard presentation commands

AISInitViewer

Syntax:

```
AISInitViewer docname
```

Creates and sets AISViewer attribute at root label, creates AIS viewer window.

Example:

```
AISInitViewer D
```

AISRepaint

Syntax:

```
AISRepaint docname
```

Updates the AIS viewer window.

Example:

```
AISRepaint D
```

AISDisplay

Syntax:

```
AISDisplay docname entry [not_update]
```

Displays a presentation of AISobject from entry label in AIS viewer. If not_update is not defined then AISobject is recomputed and all visualization settings are applied.

Example:
**AISDisplay D 0:5**

**AISUpdate**

Syntax:

```
AISUpdate docname entry
```

Recomputes a presentation of AISObject from entry label and applies the visualization setting in AIS viewer.

**Example:**

```
AISUpdate D 0:5
```

**AISErase**

Syntax:

```
AISErase docname entry
```

Erases AISObject of entry label in AIS viewer.

**Example:**

```
AISErase D 0:5
```

**AISRemove**

Syntax:

```
AISRemove docname entry
```

Erases AISObject of entry label in AIS viewer, then AISObject is removed from AIS_InteractiveContext.

**Example:**

```
AISRemove D 0:5
```
**AISSet**

Syntax:

```
AISSet docname entry ID
```

Creates *AISPresentation* attribute at *entry* label and sets as driver ID. ID must be one of the following: *A* (axis), *C* (constraint), *NS* (namedshape), *G* (geometry), *PL* (plane), *PT* (point).

**Example:**

```
AISSet D 0:5 NS
```

**AISDriver**

Syntax:

```
AISDriver docname entry [ID]
```

Returns DriverGUID stored in *AISPresentation* attribute of an *entry* label or sets a new one. ID must be one of the following: *A* (axis), *C* (constraint), *NS* (namedshape), *G* (geometry), *PL* (plane), *PT* (point).

**Example:**

```
# Get Driver GUID
AISDriver D 0:5
```

**AISUnset**

Syntax:

```
AISUnset docname entry
```

Deletes *AISPresentation* attribute (if it exists) of an *entry* label.

**Example:**

```
AISUnset D 0:5
```
**AISTransparency**

Syntax:

```
AISTransparency docname entry [transparency]
```

Sets (if transparency is defined) or gets the value of transparency for AISPresentation attribute of an entry label.

**Example:**

```
AISTransparency D 0:5 0.5
```

**AISHasOwnTransparency**

Syntax:

```
AISHasOwnTransparency docname entry
```

Tests AISPresentation attribute of an entry label by own transparency.

**Example:**

```
AISHasOwnTransparency D 0:5
```

**AISMateral**

Syntax:

```
AISMateral docname entry [material]
```

Sets (if material is defined) or gets the value of transparency for AISPresentation attribute of an entry label. material is integer from 0 to 20 (see meshmat command).

**Example:**

```
AISMateral D 0:5 5
```
**AISHasOwnMaterial**

Syntax:

```
AISHasOwnMaterial docname entry
```

Tests *AISPresentation* attribute of an *entry* label by own material.

**Example:**

```
AISHasOwnMaterial  D 0:5
```

**AISColor**

Syntax:

```
AISColor docname entry [color]
```

Sets (if *color* is defined) or gets value of color for *AISPresentation* attribute of an *entry* label. *color* is integer from 0 to 516 (see color names in *vsetcolor*).

**Example:**

```
AISColor  D 0:5 25
```

**AISHasOwnColor**

Syntax:

```
AISHasOwnColor docname entry
```

Tests *AISPresentation* attribute of an *entry* label by own color.

**Example:**

```
AISHasOwnColor  D 0:5
```
Geometry commands

Overview

Draw provides a set of commands to test geometry libraries. These commands are found in the TGEOMETRY executable, or in any Draw executable which includes GeometryTest commands.

In the context of Geometry, Draw includes the following types of variable:

- 2d and 3d points
- The 2d curve, which corresponds to Curve in Geom2d.
- The 3d curve and surface, which correspond to Curve and Surface in Geom package.

Draw geometric variables never share data; the copy command will always make a complete copy of the content of the variable.

The following topics are covered in the nine sections of this chapter:

- **Curve creation** deals with the various types of curves and how to create them.
- **Surface creation** deals with the different types of surfaces and how to create them.
- **Curve and surface modification** deals with the commands used to modify the definition of curves and surfaces, most of which concern modifications to bezier and bspline curves.
- **Geometric transformations** covers translation, rotation, mirror image and point scaling transformations.
- **Curve and Surface Analysis** deals with the commands used to compute points, derivatives and curvatures.
- **Intersections** presents intersections of surfaces and curves.
- **Approximations** deals with creating curves and surfaces from a set of points.
- **Constraints** concerns construction of 2d circles and lines by constraints such as tangency.
- **Display** describes commands to control the display of curves and surfaces.
Where possible, the commands have been made broad in application, i.e. they apply to 2d curves, 3d curves and surfaces. For instance, the circle command may create a 2d or a 3d circle depending on the number of arguments given.

Likewise, the translate command will process points, curves or surfaces, depending on argument type. You may not always find the specific command you are looking for in the section where you expect it to be. In that case, look in another section. The trim command, for example, is described in the surface section. It can, nonetheless, be used with curves as well.
Curve creation

This section deals with both points and curves. Types of curves are:

- Analytical curves such as lines, circles, ellipses, parabolas, and hyperbolas.
- Polar curves such as bezier curves and bspline curves.
- Trimmed curves and offset curves made from other curves with the `trim` and `offset` commands. Because they are used on both curves and surfaces, the `trim` and `offset` commands are described in the surface creation section.
- NURBS can be created from other curves using `convert` in the Surface Creation section.
- Curves can be created from the isoparametric lines of surfaces by the `u iso` and `v iso` commands.
- 3d curves can be created from 2d curves and vice versa using the `to3d` and `to2d` commands. The `project` command computes a 2d curve on a 3d surface.

Curves are displayed with an arrow showing the last parameter.

**point**

Syntax:

```
point name x y [z]
```

Creates a 2d or 3d point, depending on the number of arguments.

**Example:**

```
# 2d point
point p1 1 2

# 3d point
point p2 10 20 -5
```

**line**
Syntax:

```
line name x y [z] dx dy [dz]
```

Creates a 2d or 3d line. $x, y, z$ are the coordinates of the line’s point of origin; $dx, dy, dz$ give the direction vector.

A 2d line will be represented as $x y dx dy$, and a 3d line as $x y z dx dy dz$. A line is parameterized along its length starting from the point of origin along the direction vector. The direction vector is normalized and must not be null. Lines are infinite, even though their representation is not.

Example:

```
# a 2d line at 45 degrees of the X axis
line l 2 0 1 1

# a 3d line through the point 10 0 0 and parallel to Z
line l 10 0 0 0 0 1
```

circle

Syntax:

```
circle name x y [z [dx dy dz]] [ux uy [uz]] radius
```

Creates a 2d or a 3d circle.

In 2d, $x, y$ are the coordinates of the center and $ux, uy$ define the vector towards the point of origin of the parameters. By default, this direction is (1,0). The X Axis of the local coordinate system defines the origin of the parameters of the circle. Use another vector than the x axis to change the origin of parameters.

In 3d, $x, y, z$ are the coordinates of the center; $dx, dy, dz$ give the vector normal to the plane of the circle. By default, this vector is (0,0,1) i.e. the Z axis (it must not be null). $ux, uy, uz$ is the direction of the origin; if not given, a default direction will be computed. This vector must neither be null nor parallel to $dx, dy, dz$. 
The circle is parameterized by the angle in [0,2*pi] starting from the origin and. Note that the specification of origin direction and plane is the same for all analytical curves and surfaces.

**Example:**

```plaintext
# A 2d circle of radius 5 centered at 10,-2
circle c1 10 -2 5

# another 2d circle with a user defined origin
# the point of parameter 0 on this circle will be
# 1+sqrt(2),1+sqrt(2)
circle c2 1 1 1 1 2

# a 3d circle, center 10 20 -5, axis Z, radius 17
circle c3 10 20 -5 17

# same 3d circle with axis Y
circle c4 10 20 -5 0 1 0 17

# full 3d circle, axis X, origin on Z
circle c5 10 20 -5 1 0 0 0 0 1 17
```

**ellipse**

**Syntax:**

```plaintext
ellipse name x y [z [dx dy dz]] [ux uy [uz]]
    firstradius secondradius
```

Creates a 2d or 3d ellipse. In a 2d ellipse, the first two arguments define the center; in a 3d ellipse, the first three. The axis system is given by `firstradius`, the major radius, and `secondradius`, the minor radius. The parameter range of the ellipse is [0,2*pi] starting from the X axis and going towards the Y axis. The Draw ellipse is parameterized by an angle:

\[
P(u) = 0 + \text{firstradius} \times \cos(u) \times \text{Xdir} + \text{secondradius} \times \sin(u) \times \text{Ydir}
\]
where:

- $P$ is the point of parameter $u$,
- $O$, $Xdir$ and $Ydir$ are respectively the origin, $X$ Direction and $Y$ Direction of its local coordinate system.

**Example:**

```plaintext
# default 2d ellipse
ellipse e1 10 5 20 10

# 2d ellipse at angle 60 degree
ellipse e2 0 0 1 2 30 5

# 3d ellipse, in the XY plane
ellipse e3 0 0 0 25 5

# 3d ellipse in the X,Z plane with axis 1, 0 ,1
ellipse e4 0 0 0 0 1 0 1 0 1 25 5
```

**hyperbola**

Syntax:

```plaintext
hyperbola name x y z [dx dy dz] [ux uy uz]
  firstradius secondradius
```

Creates a 2d or 3d conic. The first arguments define the center. The axis system is given by $firstradius$, the major radius, and $secondradius$, the minor radius. Note that the hyperbola has only one branch, that in the X direction.

The Draw hyperbola is parameterized as follows:

$$P(U) = 0 + firstradius*Cosh(U)*XDir + secondradius*Sinh(U)*YDir$$

where:
- \( P \) is the point of parameter \( U \),
- \( O, XDir \) and \( YDir \) are respectively the origin, \( X Direction \) and \( YDirection \) of its local coordinate system.

**Example:**

```plaintext
# default 2d hyperbola, with asymptotes 1,1 -1,1
hyperbola h1 0 0 30 30

# 2d hyperbola at angle 60 degrees
hyperbola h2 0 0 1 2 20 20

# 3d hyperbola, in the XY plane
hyperbola h3 0 0 0 50 50
```

**parabola**

Syntax:

```
parabola name x y [z [dx dy dz]] [ux uy [uz]]
FocalLength
```

Creates a 2d or 3d parabola. in the axis system defined by the first arguments. The origin is the apex of the parabola.

The \textit{Geom_Parabola} is parameterized as follows:

\[
P(u) = 0 + u^2/(4.*F)*XDir + u*YDir
\]

where:

- \( P \) is the point of parameter \( u \),
- \( O, XDir \) and \( YDir \) are respectively the origin, \( X Direction \) and \( Y Direction \) of its local coordinate system,
- \( F \) is the focal length of the parabola.

**Example:**

```plaintext
# 2d parabola
parabola p1 0 0 50
```
# 2d parabola with convexity +Y
text:
parabola p2 0 0 0 1 50

text:

# 3d parabola in the Y-Z plane, convexity +Z
text:
parabola p3 0 0 0 1 0 0 0 0 1 50

beziercurve, 2dbeziercurve

Syntax:

beziercurve name nbpole pole, [weight]
2dbeziercurve name nbpole pole, [weight]

Creates a 3d rational or non-rational Bezier curve. Give the number of poles (control points,) and the coordinates of the poles *(x1 y1 z1 [w1] x2 y2 z2 [w2])* The degree will be nbpoles-1. To create a rational curve, give weights with the poles. You must give weights for all poles or for none. If the weights of all the poles are equal, the curve is polynomial, and therefore non-rational.

Example:

# a rational 2d bezier curve (arc of circle)
2dbeziercurve ci 3 0 0 1 10 0 sqrt(2.)/2. 10 10 1

# a 3d bezier curve, not rational
beziercurve cc 4 0 0 0 10 0 0 10 0 10 10 10 10

bsplinecurve, 2dbsplinecurve, pbsplinecurve,
2dpbsplinecurve

Syntax:

bsplinecurve name degree nbknots knot, umult pole, weight
2dbsplinecurve name degree nbknots knot, umult pole, weight
pbsplinecurve  name degree nbknots knot, umult pole, weight (periodic)
2dpbsplinecurve name degree nbknots knot, umult pole, weight (periodic)

Creates 2d or 3d bspline curves; the **pbsplinecurve** and **2dpbsplinecurve** commands create periodic bspline curves.

A bspline curve is defined by its degree, its periodic or non-periodic nature, a table of knots and a table of poles (i.e. control points). Consequently, specify the degree, the number of knots, and for each knot, the multiplicity, for each pole, the weight. In the syntax above, the commas link the adjacent arguments which they fall between: knot and multiplicities, pole and weight.

The table of knots is an increasing sequence of reals without repetition. Multiplicities must be lower or equal to the degree of the curve. For non-periodic curves, the first and last multiplicities can be equal to degree+1. For a periodic curve, the first and last multiplicities must be equal.

The poles must be given with their weights, use weights of 1 for a non rational curve, the number of poles must be:

- For a non periodic curve: Sum of multiplicities - degree + 1
- For a periodic curve: Sum of multiplicities - last multiplicity

**Example:**

```plaintext
# a bspline curve with 4 poles and 3 knots
bsplinecurve bc 2 3 0 3 1 1 2 3 \
10 0 7 1 7 0 7 1 3 0 8 1 0 0 7 1
# a 2d periodic circle (parameter from 0 to 2*pi !!)
dset h sqrt(3)/2
2dpbsplinecurve c 2 \
4 0 2 pi/1.5 2 pi/0.75 2 2*pi 2 \
0 -h/3 1 \
0.5 -h/3 0.5 \
0.25 h/6 1 \
0 2*h/3 0.5 \
```


Note that you can create the NURBS subset of bspline curves and surfaces by trimming analytical curves and surfaces and executing the command convert.

uiSO, viso

Syntax:

uiSO name surface u
viso name surface u

Creates a U or V isoparametric curve from a surface.

Example:

# create a cylinder and extract iso curves

cylinder c 10
uiSO c1 c pi/6
viso c2 c

Note that this cannot be done from offset surfaces.

to3d, to2d

Syntax:

to3d name curve2d [plane]
to2d name curve3d [plane]

Create respectively a 3d curve from a 2d curve and a 2d curve from a 3d curve. The transformation uses a planar surface to define the XY plane in 3d (by default this plane is the default OXYplane). to3d always gives a correct result, but as to2d is not a projection, it may surprise you. It is always correct if the curve is planar and parallel to the plane of
projection. The points defining the curve are projected on the plane. A circle, however, will remain a circle and will not be changed to an ellipse.

Example:

```plaintext
# the following commands
circle c 0 0 5
plane p -2 1 0 1 2 3
to3d c c p

# will create the same circle as
circle c -2 1 0 1 2 3 5
```

See also: project

**project**

Syntax:

```plaintext
project name curve3d surface
```

Computes a 2d curve in the parametric space of a surface corresponding to a 3d curve. This can only be used on analytical surfaces.

If we, for example, intersect a cylinder and a plane and project the resulting ellipse on the cylinder, this will create a 2d sinusoid-like bspline.

```plaintext
cylinder c 5
plane p 0 0 0 0 1 1
intersect i c p
project i2d i c
```
Surface creation

The following types of surfaces exist:

- Analytical surfaces: plane, cylinder, cone, sphere, torus;
- Polar surfaces: bezier surfaces, bspline surfaces;
- Trimmed and Offset surfaces;
- Surfaces produced by Revolution and Extrusion, created from curves with the `revsurf` and `extsurf`;
- NURBS surfaces.

Surfaces are displayed with isoparametric lines. To show the parameterization, a small parametric line with a length 1/10 of V is displayed at 1/10 of U.

**plane**

Syntax:

```
plane name [x y z [dx dy dz [ux uy uz]]]
```

Creates an infinite plane.

A plane is the same as a 3d coordinate system, \(x, y, z\) is the origin, \(dx, dy, dz\) is the Z direction and \(ux, uy, uz\) is the X direction.

The plane is perpendicular to Z and X is the U parameter. \(dx, dy, dz\) and \(ux, uy, uz\) must not be null or collinear. \(ux, uy, uz\) will be modified to be orthogonal to \(dx, dy, dz\).

There are default values for the coordinate system. If no arguments are given, the global system \((0,0,0), (0,0,1), (1,0,0)\). If only the origin is given, the axes are those given by default\((0,0,1), (1,0,0)\). If the origin and the Z axis are given, the X axis is generated perpendicular to the Z axis.

Note that this definition will be used for all analytical surfaces.

**Example:**
# a plane through the point 10,0,0 perpendicular to X
# with U direction on Y
plane p1 10 0 0 1 0 0 0 1 0

# an horizontal plane with origin 10, -20, -5
plane p2 10 -20 -5

cylinder

Syntax:

cylinder name [x y z [dx dy dz [ux uy uz]]] radius

A cylinder is defined by a coordinate system, and a radius. The surface generated is an infinite cylinder with the Z axis as the axis. The U parameter is the angle starting from X going in the Y direction.

Example:

# a cylinder on the default Z axis, radius 10
cylinder c1 10

# a cylinder, also along the Z axis but with origin 5,
# 10, -3
cylinder c2 5 10 -3 10

# a cylinder through the origin and on a diagonal
# with longitude pi/3 and latitude pi/4 (euler angles)
dset lo pi/3. la pi/4.
cylinder c3 0 0 0 cos(la)*cos(lo) cos(la)*sin(lo) sin(la) 10

cone

Syntax:
cone name [x y z [dx dy dz [ux uy uz]]] semi-angle radius

Creates a cone in the infinite coordinate system along the Z-axis. The radius is that of the circle at the intersection of the cone and the XY plane. The semi-angle is the angle formed by the cone relative to the axis; it should be between -90 and 90. If the radius is 0, the vertex is the origin.

Example:

# a cone at 45 degrees at the origin on Z
cone c1 45 0

# a cone on axis Z with radius r1 at z1 and r2 at z2
cone c2 0 0 z1 180.*atan2(r2-r1,z2-z1)/pi r1

sphere

Syntax:

sphere name [x y z [dx dy dz [ux uy uz]]] radius

Creates a sphere in the local coordinate system defined in the plane command. The sphere is centered at the origin.

To parameterize the sphere, u is the angle from X to Y, between 0 and 2*pi. v is the angle in the half-circle at angle u in the plane containing the Z axis. v is between -pi/2 and pi/2. The poles are the points Z = +/-radius; their parameters are u, +/-pi/2 for any u in 0,2*pi.

Example:

# a sphere at the origin
sphere s1 10
# a sphere at 10 10 10, with poles on the axis 1,1,1
sphere s2 10 10 10 1 1 1 10

torus
Syntax:

torus name [x y z [dx dy dz [ux uy uz]]] major minor

Creates a torus in the local coordinate system with the given major and minor radii. \( Z \) is the axis for the major radius. The major radius may be lower in value than the minor radius.

To parameterize a torus, \( u \) is the angle from \( X \) to \( Y \); \( v \) is the angle in the plane at angle \( u \) from the \( XY \) plane to \( Z \). \( u \) and \( v \) are in \( 0,2\pi \).

Example:

```plaintext
# a torus at the origin
torus t1 20 5

# a torus in another coordinate system
torus t2 10 5 -2 2 1 0 20 5
```

**beziersurf**

Syntax:

beziersurf name nbupoles nbvolpes pole, [weight]

Use this command to create a bezier surface, rational or non-rational. First give the numbers of poles in the \( u \) and \( v \) directions.

Then give the poles in the following order: pole(1, 1), pole(nbupoles, 1), pole(1, nbvpoles) and pole(nbupoles, nbvpoles).

Weights may be omitted, but if you give one weight you must give all of them.

Example:

```plaintext
# a non-rational degree 2,3 surface
beziersurf s 3 4 \ 0 0 0 10 0 5 20 0 0 \ 0 10 2 10 10 3 20 10 2 \
```
bsplinesurf, upbsplinesurf, vpbsplinesurf, uvpbsplinesurf

Syntax:

```plaintext
bsplinesurf name udegree nbuknots uknot umult ...  
             nbvknout vknout  
vmult ... x y z w ...  
upbsplinesurf ...  
vpbsplinesurf ...  
uvpbsplinesurf ...
```

- `bsplinesurf` generates bspline surfaces;
- `upbsplinesurf` creates a bspline surface periodic in u;
- `vpbsplinesurf` creates one periodic in v;
- `uvpbsplinesurf` creates one periodic in uv.

The syntax is similar to the `bsplinecurve` command. First give the degree in u and the knots in u with their multiplicities, then do the same in v. The poles follow. The number of poles is the product of the number in u and the number in v.

See `bsplinecurve` to compute the number of poles, the poles are first given in U as in the `beziersurf` command. You must give weights if the surface is rational.

Example:

```plaintext
# create a bspline surface of degree 1 2  
# with two knots in U and three in V  
bsplinesurf s \  
1 2 0 2 1 2 \  
2 3 0 3 1 1 2 3 \  
0 0 0 1 10 0 5 1 \  
0 10 2 1 10 10 3 1 \  
0 20 10 1 10 20 20 1 \  
0 30 0 1 10 30 0 1
```
**trim, trimu, trimv**

Syntax:

```plaintext
trim newname name [u1 u2 [v1 v2]]
trimu newname name
trimv newname name
```

The **trim** commands create trimmed curves or trimmed surfaces. Note that trimmed curves and surfaces are classes of the `Geom` package.

- **trim** creates either a new trimmed curve from a curve or a new trimmed surface in u and v from a surface.
- **trimu** creates a u-trimmed surface,
- **trimv** creates a v-trimmed surface.

After an initial trim, a second execution with no parameters given recreates the basis curve. The curves can be either 2d or 3d. If the trimming parameters decrease and if the curve or surface is not periodic, the direction is reversed.

**Note** that a trimmed curve or surface contains a copy of the basis geometry: modifying that will not modify the trimmed geometry.Trimming trimmed geometry will not create multiple levels of trimming. The basis geometry will be used.

**Example:**

```plaintext
# create a 3d circle
circle c 0 0 0 10

# trim it, use the same variable, the original is deleted
trim c c 0 pi/2

# the original can be recovered!
trim orc c

# trim again
```
trim c c pi/4 pi/2

# the original is not the trimmed curve but the basis
trim orc c

# as the circle is periodic, the two following
   commands
are identical
trim cc c pi/2 0
trim cc c pi/2 2*pi

# trim an infinite cylinder
cylinder cy 10
trimv cy cy 0 50

**offset**

Syntax:

```
offset name basename distance [dx dy dz]
```

Creates offset curves or surfaces at a given distance from a basis curve or surface. Offset curves and surfaces are classes from the *Geom* package.

The curve can be a 2d or a 3d curve. To compute the offsets for a 3d curve, you must also give a vector *dx,dy,dz*. For a planar curve, this vector is usually the normal to the plane containing the curve.

The offset curve or surface copies the basic geometry, which can be modified later.

**Example:**

```
# graphic demonstration that the outline of a torus
# is the offset of an ellipse
smallview +X+Y
dset angle pi/6
torus t 0 0 0 0 cos(angle) sin(angle) 50 20
```
fit
ellipse e 0 0 0 50 50*\text{sin}(\text{angle})

\# note that the distance can be negative
offset l1 e 20 0 0 1

\textbf{revsurf}

Syntax:

\begin{verbatim}
revsurf name curvename x y z dx dy dz
\end{verbatim}

Creates a surface of revolution from a 3d curve.

A surface of revolution or revolved surface is obtained by rotating a curve (called the \textit{meridian}) through a complete revolution about an axis (referred to as the \textit{axis of revolution}). The curve and the axis must be in the same plane (the \textit{reference plane} of the surface). Give the point of origin x,y,z and the vector dx,dy,dz to define the axis of revolution.

To parameterize a surface of revolution: \(u\) is the angle of rotation around the axis. Its origin is given by the position of the meridian on the surface. \(v\) is the parameter of the meridian.

\textbf{Example:}

\begin{verbatim}
\# another way of creating a torus like surface
circle c 50 0 0 20
revsurf s c 0 0 0 0 1 0
\end{verbatim}

\textbf{extsurf}

Syntax:

\begin{verbatim}
extsurf newname curvename dx dy dz
\end{verbatim}

Creates a surface of linear extrusion from a 3d curve. The basis curve is swept in a given direction, the \textit{direction of extrusion} defined by a vector.

In the syntax, \(dx,dy,dz\) gives the direction of extrusion.
To parameterize a surface of extrusion: \( u \) is the parameter along the extruded curve; the \( v \) parameter is along the direction of extrusion.

**Example:**

```plaintext
# an elliptic cylinder
elipse e 0 0 0 10 5
extsurf s e 0 0 1
# to make it finite
trimv s s 0 10
```

**convert**

Syntax:

```plaintext
convert newname name
```

Creates a 2d or 3d NURBS curve or a NURBS surface from any 2d curve, 3d curve or surface. In other words, conics, beziers and bsplines are turned into NURBS. Offsets are not processed.

**Example:**

```plaintext
# turn a 2d arc of a circle into a 2d NURBS
circle c 0 0 5
trim c c 0 pi/3
convert c1 c

# an easy way to make a planar bspline surface
plane p
trim p p -1 1 -1 1
convert p1 p
```

**Note** that offset curves and surfaces are not processed by this command.
Curve and surface modifications

Draw provides commands to modify curves and surfaces, some of them are general, others restricted to bezier curves or bsplines.

General modifications:

- Reversing the parametrization: `reverse, ureverse, vreverse`

Modifications for both bezier curves and bsplines:

- Exchanging U and V on a surface: `exchuv`
- Segmentation: `segment, segsur`
- Increasing the degree: `incdeg, incudeg, incvdeg`
- Moving poles: `cmovep, movep, movecolp, moverowp`

Modifications for bezier curves:

- Adding and removing poles: `insertpole, rempole, remcolpole, remrowpole`

Modifications for bspline:

- Inserting and removing knots: `insertknot, remknot, insertuknot, remuknot, insetvknot, remv knot`
- Modifying periodic curves and surfaces: `setperiodic, setnotperiodic, setorigin, setupperiodic, setunotperiodic, setuorigin, setvperiodic, setvnotperiodic, setvorigin`

`reverse, ureverse, vreverse`

Syntax:

```
reverse curvename
ureverse surfacename
vreverse surfacename
```

The `reverse` command reverses the parameterization and inverses the orientation of a 2d or 3d curve. Note that the geometry is modified. To
keep the curve or the surface, you must copy it before modification.

**ureverse** or **vreverse** reverse the u or v parameter of a surface. Note that the new parameters of the curve may change according to the type of curve. For instance, they will change sign on a line or stay 0,1 on a bezier.

Reversing a parameter on an analytical surface may create an indirect coordinate system.

**Example:**

```plaintext
# reverse a trimmed 2d circle
circle c 0 0 5
trim c c pi/4 pi/2
reverse c

# dumping c will show that it is now trimmed between
# 3*pi/2 and 7*pi/4 i.e. 2*pi-pi/2 and 2*pi-pi/4
```

**exchuv**

Syntax:

```
exchuv surfacename
```

For a bezier or bspline surface this command exchanges the u and v parameters.

**Example:**

```plaintext
# exchanging u and v on a spline (made from a cylinder)
cylinder c 5
trimv c c 0 10
convert c1 c
exchuv c1
```

**segment, segsur**
Syntax:

```
segment curve Ufirst Ulast
segsur surface Ufirst Ulast Vfirst Vlast
```

**segment** and **segsur** segment a bezier curve and a bspline curve or surface respectively.

These commands modify the curve to restrict it between the new parameters: **Ufirst**, the starting point of the modified curve, and **Ulast**, the end point. **Ufirst** is less than **Ulast**.

This command must not be confused with **trim** which creates a new geometry.

**Example:**

```
# segment a bezier curve in half
beziercurve c 3 0 0 0 10 0 0 10 10 0
segment c ufirst ulast
```

**iincudeg, incvdeg**

Syntax:

```
incudeg surfacename newdegree
incvdeg surfacename newdegree
```

**incudeg** and **incvdeg** increase the degree in the U or V parameter of a bezier or bspline surface.

**Example:**

```
# make a planar bspline and increase the degree to 2
plane p
trim p p -1 1 -1 1
convert p1 p
incudeg p1 2
```
Note that the geometry is modified.

**cmovep, movep, movecolp, moverowp**

Syntax:

- `cmovep curve index dx dy [dz]`
- `movep surface uindex vindex dx dy dz`
- `movecolp surface uindex dx dy dz`
- `moverowp surface vindex dx dy dz`

**move** methods translate poles of a bezier curve, a bspline curve or a bspline surface.

- **cmovep** and **movep** translate one pole with a given index.
- **movecolp** and **moverowp** translate a whole column (expressed by the `uindex`) or row (expressed by the `vindex`) of poles.

**Example:**

```bash
# start with a plane
# transform to bspline, raise degree and add relief
plane p
trim p p -10 10 -10 10
convert p1 p
incud p1 2
incvd p1 2
movecolp p1 2 0 0 5
moverowp p1 2 0 0 5
movep p1 2 2 0 0 5
```

**insertpole, rempole, remcolpole, remrowpole**

Syntax:

- `insertpole curvename index x y [z] [weight]`
- `rempole curvename index`
insertpole inserts a new pole into a 2d or 3d bezier curve. You may add a weight for the pole. The default value for the weight is 1. The pole is added at the position after that of the index pole. Use an index 0 to insert the new pole before the first one already existing in your drawing.

rempole removes a pole from a 2d or 3d bezier curve. Leave at least two poles in the curves.

remcolpole and remrowpole remove a column or a row of poles from a bezier surface. A column is in the v direction and a row in the u direction. The resulting degree must be at least 1; i.e. there will be two rows and two columns left.

Example:

```
# start with a segment, insert a pole at end
# then remove the central pole
beziercurve c 2 0 0 0 10 0 0
insertpole c 2 10 10 0
rempole c 2
```

insertknot, insertuknot, insertvknot

Syntax:

```
insertknot name knot [mult = 1] [knot mult ...]
insertuknot surfacename knot mult
insertvknot surfacename knot mult
```

insertknot inserts knots in the knot sequence of a bspline curve. You must give a knot value and a target multiplicity. The default multiplicity is 1. If there is already a knot with the given value and a multiplicity lower than the target one, its multiplicity will be raised.

insertuknot and insertvknot insert knots in a surface.

Example:
# create a cylindrical surface and insert a knot

cylinder c 10
trim c c 0 pi/2 0 10
convert c1 c
insertuknot c1 pi/4 1

remknot, remuknot, remvknot

Syntax:

```
remknot index [mult] [tol]
remuknot index [mult] [tol]
remvknot index [mult] [tol]
```

remknot removes a knot from the knot sequence of a curve or a surface. Give the index of the knot and optionally, the target multiplicity. If the target multiplicity is not 0, the multiplicity of the knot will be lowered. As the curve may be modified, you are allowed to set a tolerance to control the process. If the tolerance is low, the knot will only be removed if the curve will not be modified.

By default, if no tolerance is given, the knot will always be removed.

Example:

```
# bspline circle, remove a knot
circle c 0 0 5
convert c1 c
incd c1 5
remknot c1 2
```

Note that Curves or Surfaces may be modified.

setperiodic, setnotperiodic, setuperiodic, setunotperiodic, setvperiodic, setvnotperiodic

Syntax:

```
setperiodic curve
```
setnotperiodic curve
setuperiodic surface
setunotperiodic surface
setvperiodic surface
setvnotperiodic surface

**setperiodic** turns a bspline curve into a periodic bspline curve; the knot vector stays the same and excess poles are truncated. The curve may be modified if it has not been closed. **setnotperiodic** removes the periodicity of a periodic curve. The pole table may be modified. Note that knots are added at the beginning and the end of the knot vector and the multiplicities are knots set to degree+1 at the start and the end.

**setuperiodic** and **setvperiodic** make the u or the v parameter of bspline surfaces periodic; **setunotperiodic**, and **setvnotperiodic** remove periodicity from the u or the v parameter of bspline surfaces.

**Example:**

```
# a circle deperioedicized
circle c 0 0 5
convert c1 c
setnotperiodic c1
```

**setorigin, setuorigin, setvorigin**

Syntax:

```
setorigin curvename index
setuorigin surfacename index
setvorigin surfacename index
```

These commands change the origin of the parameters on periodic curves or surfaces. The new origin must be an existing knot. To set an origin other than an existing knot, you must first insert one with the **insertknot** command.

**Example:**
# a torus with new U and V origins
torus t 20 5
call t1 t
setuorigin t1 2
setvorigin t1 2
Transformations

Draw provides commands to apply linear transformations to geometric objects: they include translation, rotation, mirroring and scaling.

**translate, dtranslate**

Syntax:

```
translate name [names ...] dx dy dz
2dtranslate name [names ...] dx dy
```

The **Translate** command translates 3d points, curves and surfaces along a vector \(dx,dy,dz\). You can translate more than one object with the same command.

For 2d points or curves, use the **2dtranslate** command.

**Example:**

```
# 3d translation
point p 10 20 30
circle c 10 20 30 5
torus t 10 20 30 5 2
translate p c t 0 0 15
```

*NOTE Objects are modified by this command.*

**rotate, 2drotate**

Syntax:

```
rotate name [name ...] x y z dx dy dz angle
2drotate name [name ...] x y angle
```

The **rotate** command rotates a 3d point curve or surface. You must give an axis of rotation with a point \(x,y,z\), a vector \(dx,dy,dz\) and an angle in degrees.
For a 2d rotation, you need only give the center point and the angle. In 2d or 3d, the angle can be negative.

**Example:**

```bash
# make a helix of circles. create a script file with
this code and execute it using **source**.
circle c0 10 0 0 3
for {set i 1} {$i <= 10} {incr i} {
    copy c[expr $i-1] c$i
    translate c$i 0 0 3
    rotate c$i 0 0 0 0 0 1 36
}
```

**pmirror, lmirror, smirror, dpmirror, dlmirror**

Syntax:

- `pmirror name [names ...] x y z`
- `lmirror name [names ...] x y z dx dy dz`
- `smirror name [names ...] x y z dx dy dz`
- `2dpmirror name [names ...] x y`
- `2dlmirror name [names ...] x y dx dy`

The mirror commands perform a mirror transformation of 2d or 3d geometry.

- **pmirror** is the point mirror, mirroring 3d curves and surfaces about a point of symmetry.
- **lmirror** is the line mirror command, mirroring 3d curves and surfaces about an axis of symmetry.
- **smirror** is the surface mirror, mirroring 3d curves and surfaces about a plane of symmetry. In the last case, the plane of symmetry is perpendicular to dx,dy,dz.
- **2dpmirror** is the point mirror in 2D.
- **2dlmirror** is the axis symmetry mirror in 2D.

**Example:**
# build 3 images of a torus
torus t 10 10 10 1 2 3 5 1
copy t t1
pmirror t1 0 0 0
copy t t2
lmirror t2 0 0 0 1 0 0
copy t t3
smirror t3 0 0 0 1 0 0

**pscale, dpscale**

Syntax:

```
pscale name [name ...] x y z s
2dpscale name [name ...] x y s
```

The *pscale* and 2dpscale commands transform an object by point scaling. You must give the center and the scaling factor. Because other scalings modify the type of the object, they are not provided. For example, a sphere may be transformed into an ellipsoid. Using a scaling factor of -1 is similar to using **pmirror**.

**Example:**

```
# double the size of a sphere
sphere s 0 0 0 10
pscale s 0 0 0 2
```
Curve and surface analysis

**Draw** provides methods to compute information about curves and surfaces:

- **coord** to find the coordinates of a point.
- **cvalue** and **2dcvalue** to compute points and derivatives on curves.
- **svalue** to compute points and derivatives on a surface.
- **localprop** and **minmaxcurandif** to compute the curvature on a curve.
- **parameters** to compute (u,v) values for a point on a surface.
- **proj** and **2dproj** to project a point on a curve or a surface.
- **surface_radius** to compute the curvature on a surface.

**coord**

Syntax:

```
coord P x y [z]
```

Sets the x, y (and optionally z) coordinates of the point P.

**Example:**

```
# translate a point
point p 10 5 5
translate p 5 0 0
coord p x y z
# x value is 15
```

**cvalue, 2dcvalue**

Syntax:

```
cvalue curve U x y z [d1x d1y d1z [d2x d2y d2z]]
2dcvalue curve U x y [d1x d1y [d2x d2y]]
```

For a curve at a given parameter, and depending on the number of
arguments, \textbf{cvalue} computes the coordinates in \(x,y,z\), the first derivative in \(d_{1x},d_{1y},d_{1z}\) and the second derivative in \(d_{2x},d_{2y},d_{2z}\).

**Example:**

Let on a bezier curve at parameter 0 the point is the first pole; the first derivative is the vector to the second pole multiplied by the degree; the second derivative is the difference first to the second pole, second to the third pole multiplied by \textit{degree-1}:

\[
\begin{align*}
2\text{dbeziercurve} & \quad c \quad 4 \quad 0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 1 \quad 3 \quad 0 \\
2\text{dcvalue} & \quad c \quad 0 \quad x \quad y \quad d_{1x} \quad d_{1y} \quad d_{2x} \quad d_{2y} \\
# \quad \text{values of} \quad x \quad y \quad d_{1x} \quad d_{1y} \quad d_{2x} \quad d_{2y} \\
# \quad \text{are} \quad 0 \quad 0 \quad 3 \quad 3 \quad 0 \quad -6
\end{align*}
\]

\textbf{svalue}

Syntax:

\[
\text{svalue surfname U v x y z [dux duy duz dvx dvy dvz [d2ux d2uy d2uz d2vx d2vy d2vz d2uvx d2uvy d2uvz]]}
\]

Computes points and derivatives on a surface for a pair of parameter values. The result depends on the number of arguments. You can compute the first and the second derivatives.

**Example:**

\[
\begin{align*}
# \quad \text{display points on a sphere} \\
\text{sphere} \quad s \quad 10 \\
\text{for} \quad \{\text{dset} \quad t \quad 0\} \quad \{\text{[dval t] <= 1} \} \quad \{\text{dset} \quad t \quad t+0.01\} \quad \{
\text{svalue} \quad s \quad t*2*\pi \quad t*\pi-\pi/2 \quad x \quad y \quad z \\
\text{point} \quad . \quad x \quad y \quad z \\
\}
\end{align*}
\]

\textbf{localprop, minmaxcurandinf}
Syntax:

localprop curvename U
minmaxcurandinf curve

**localprop** computes the curvature of a curve. **minmaxcurandinf** computes and prints the parameters of the points where the curvature is minimum and maximum on a 2d curve.

**Example:**

```
# show curvature at the center of a bezier curve
beziercurve c 3 0 0 0 10 2 0 20 0 0
localprop c 0.5
== Curvature : 0.02
```

**parameters**

Syntax:

```
parameters surf/curve x y z U [V]
```

Returns the parameters on the surface of the 3d point x,y,z in variables u and v. This command may only be used on analytical surfaces: plane, cylinder, cone, sphere and torus.

**Example:**

```
# Compute parameters on a plane
plane p 0 0 10 1 1 0
parameters p 5 5 5 u v
# the values of u and v are : 0 5
```

**proj, 2dproj**

Syntax:

```
proj name x y z
2dproj name xy
```
Use `proj` to project a point on a 3d curve or a surface and `2dproj` for a 2d curve.

The command will compute and display all points in the projection. The lines joining the point to the projections are created with the names *ext_1, ext_2, ...*

**Example:**

Let us project a point on a torus

```
torus t 20 5
proj t 30 10 7
== ext_1 ext_2 ext_3 ext_4
```

**surface_radius**

Syntax:

```
surface_radius surface u v [c1 c2]
```

Computes the main curvatures of a surface at parameters *(u,v)*. If there are extra arguments, their curvatures are stored in variables `c1` and `c2`.

**Example:**

Let us compute curvatures of a cylinder:

```
cylinder c 5
surface_radius c pi 3 c1 c2
== Min Radius of Curvature : -5
== Min Radius of Curvature : infinite
```
Intersections

- **intersect** computes intersections of surfaces;
- **2dintersect** computes intersections of 2d curves.

**intersect**

Syntax:

```plaintext
intersect name surface1 surface2
```

Intersects two surfaces; if there is one intersection curve it will be named `name`, if there are more than one they will be named `name_1, name_2, ...`

**Example:**

```plaintext
# create an ellipse
cone c 45 0
plane p 0 0 40 0 1 5
intersect e c p
```

**2dintersect**

Syntax:

```plaintext
2dintersect curve1 curve2
```

Displays the intersection points between two 2d curves.

**Example:**

```plaintext
# intersect two 2d ellipses
ellipse e1 0 0 5 2
ellipse e2 0 0 0 1 5 2
2dintersect e1 e2
```
Approximations

Draw provides command to create curves and surfaces by approximation.

- **2dapprox** fits a curve through 2d points;
- **appro** fits a curve through 3d points;
- **surfapp** and **grilapp** fit a surface through 3d points;
- **2dinterpolate** interpolates a curve.

**appro, dapprox**

Syntax:

```
appro result nbpoint [curve]
2dapprox result nbpoint [curve / x1 y1 x2 y2]
```

These commands fit a curve through a set of points. First give the number of points, then choose one of the three ways available to get the points. If you have no arguments, click on the points. If you have a curve argument or a list of points, the command launches computation of the points on the curve.

**Example:**

Let us pick points and they will be fitted

```
2dapprox c 10
```

**surfapp, grilapp**

Syntax:

```
surfapp name nbupoints nbvpoints x y z ....
grilapp name nbupoints nbvpoints xo dx yo dy z11 z12 ...
```

- **surfapp** fits a surface through an array of u and v points,
nbupoints*nbvpoints.
- **grilapp** has the same function, but the x,y coordinates of the points are on a grid starting at x0,y0 with steps dx,dy.

**Example:**

```plaintext
# a surface using the same data as in the beziersurf example sect 4.4
surfapp s 3 4
0 0 0 10 0 5 20 0 0 \
0 10 2 10 10 3 20 10 2 \
0 20 10 10 20 20 20 10 \
0 30 0 10 30 0 20 30 0
```
Projections

Draw provides commands to project points/curves on curves/surfaces.

- **proj** projects point on the curve/surface (see *proj command description*);
- **project** projects 3D curve on the surface (see *project command description*);
- **projponf** projects point on the face.

**projponf**

Syntax:

```
projponf face pnt [extrema flag: -min/-max/-minmax]
                     [extrema algo: -g(grad)/-t(tree)]
```

**projponf** projects point *pnt* on the face *face*. You can change the Extrema options:

- To change the Extrema search algorithm use the following options:
  - *g* - for Grad algorithm;
  - *t* - for Tree algorithm;
- To change the Extrema search solutions use the following options:
  - *min* - to look for Min solutions;
  - *max* - to look for Max solutions;
  - *minmax* - to look for MinMax solutions.

**Example**

```
plane p 0 0 0 0 0 1
mkface f p
point pnt 5 5 10

projponf f pnt
  # proj dist = 10
  # uvproj = 5 5
  # pproj = 5 5 0
```
Constraints

- **cirtang** constructs 2d circles tangent to curves;
- **lintan** constructs 2d lines tangent to curves.

**cirtang**

Syntax:

```
cirtang cname curve/point/radius curve/point/radius curve/point/radius
```

Builds all circles satisfying the three constraints which are either a curve (the circle must be tangent to that curve), a point (the circle must pass through that point), or a radius for the circle. Only one constraint can be a radius. The solutions will be stored in variables `name_1`, `name_2`, etc.

**Example:**

```
# a point, a line and a radius. 2 solutions
point p 0 0
line l 10 0 -1 1
cirtang c p 1 4
== c_1 c_2
```

**lintan**

Syntax:

```
lintan name curve curve [angle]
```

Builds all 2d lines tangent to two curves. If the third angle argument is given the second curve must be a line and **lintan** will build all lines tangent to the first curve and forming the given angle with the line. The angle is given in degrees. The solutions are named `name_1`, `name_2`, etc.

**Example:**
# lines tangent to 2 circles, 4 solutions
circle c1 -10 0 10
circle c2 10 0 5
lintan 1 c1 c2

# lines at 15 degrees tangent to a circle and a line, 2 solutions: l1_1 l1_2
circle c1 -10 0 1
line l 2 0 1 1
lintan l1 c1 l 15
Display

Draw provides commands to control the display of geometric objects. Some display parameters are used for all objects, others are valid for surfaces only, some for bezier and bspline only, and others for bspline only.

On curves and surfaces, you can control the mode of representation with the dmode command. You can control the parameters for the mode with the defle command and the discr command, which control deflection and discretization respectively.

On surfaces, you can control the number of isoparametric curves displayed on the surface with the nbiso command.

On bezier and bspline curve and surface you can toggle the display of the control points with the clpoles and shpoles commands.

On bspline curves and surfaces you can toggle the display of the knots with the shknots and clknots commands.

**dmod, discr, defle**

Syntax:

```
dmode name [name ...] u/d
discr name [name ...] nbintervals
defle name [name ...] deflection
```

**dmod** command allows choosing the display mode for a curve or a surface.

In mode *u*, or *uniform deflection*, the points are computed to keep the polygon at a distance lower than the deflection of the geometry. The deflection is set with the defle command. This mode involves intensive use of computational power.

In *d*, or discretization mode, a fixed number of points is computed. This number is set with the discr command. This is the default mode. On a
bspline, the fixed number of points is computed for each span of the curve. (A span is the interval between two knots).

If the curve or the isolines seem to present too many angles, you can either increase the discretization or lower the deflection, depending on the mode. This will increase the number of points.

Example:

```plaintext
# increment the number of points on a big circle
circle c 0 0 50 50
discr 100

# change the mode
dmode c u
```

**nbiso**

Syntax:

```plaintext
nbiso name [names...] nuiso nviso
```

Changes the number of isoparametric curves displayed on a surface in the U and V directions. On a bspline surface, isoparametric curves are displayed by default at knot values. Use *nbiso* to turn this feature off.

Example:

Let us display 35 meridians and 15 parallels on a sphere:

```plaintext
sphere s 20
nbiso s 35 15
```

**clpoles, shpoles**

Syntax:

```plaintext
clpoles name
shpoles name
```
On bezier and bspline curves and surfaces, the control polygon is displayed by default: clpoles erases it and shpoles restores it.

Example:

Let us make a bezier curve and erase the poles

```
beziercurve c 3 0 0 0 10 0 0 10 10 0
clpoles c
```

**clknots, shknots**

Syntax:

```
clknuts name
shknuts name
```

By default, knots on a bspline curve or surface are displayed with markers at the points with parametric value equal to the knots. clknuts removes them and shknuts restores them.

Example:

```
# hide the knots on a bspline curve
bsplinecurve bc 2 3 0 3 1 1 2 3 \
10 0 7 1 7 0 7 1 3 0 8 1 0 0 7 1
clknuts bc
```
Topology commands

Draw provides a set of commands to test OCCT Topology libraries. The Draw commands are found in the DRAWEXE executable or in any executable including the BRepTest commands.

Topology defines the relationship between simple geometric entities, which can thus be linked together to represent complex shapes. The type of variable used by Topology in Draw is the shape variable.

The different topological shapes include:

- **COMPOUND**: A group of any type of topological object.
- **COMPSOLID**: A set of solids connected by their faces. This expands the notions of WIRE and SHELL to solids.
- **SOLID**: A part of space limited by shells. It is three dimensional.
- **SHELL**: A set of faces connected by their edges. A shell can be open or closed.
- **FACE**: In 2d, a plane; in 3d, part of a surface. Its geometry is constrained (trimmed) by contours. It is two dimensional.
- **WIRE**: A set of edges connected by their vertices. It can be open or closed depending on whether the edges are linked or not.
- **EDGE**: A topological element corresponding to a restrained curve. An edge is generally limited by vertices. It has one dimension.
- **VERTEX**: A topological element corresponding to a point. It has a zero dimension.

Shapes are usually shared. **copy** will create a new shape which shares its representation with the original. Nonetheless, two shapes sharing the same topology can be moved independently (see the section on **transformation**).

The following topics are covered in the eight sections of this chapter:

- Basic shape commands to handle the structure of shapes and control the display.
- Curve and surface topology, or methods to create topology from geometry and vice versa.
- Primitive construction commands: box, cylinder, wedge etc.
• Sweeping of shapes.
• Transformations of shapes: translation, copy, etc.
• Topological operations, or booleans.
• Drafting and blending.
• Analysis of shapes.
Basic topology

The set of basic commands allows simple operations on shapes, or step-by-step construction of objects. These commands are useful for analysis of shape structure and include:

- **isos** and **discretisation** to control display of shape faces by isoparametric curves.
- **orientation**, **complement** and **invert** to modify topological attributes such as orientation.
- **explode**, **exwire** and **nbshapes** to analyze the structure of a shape.
- **emptycopy**, **add**, **compound** to create shapes by stepwise construction.

In Draw, shapes are displayed using isoparametric curves. There is color coding for the edges:

- a red edge is an isolated edge, which belongs to no faces.
- a green edge is a free boundary edge, which belongs to one face,
- a yellow edge is a shared edge, which belongs to at least two faces.

isos, discretisation

Syntax:

```
isos [name ...][nbisos]
discretisation nbpoints
```

Determines or changes the number of isoparametric curves on shapes.

The same number is used for the u and v directions. With no arguments, **isos** prints the current default value. To determine, the number of isos for a shape, give it name as the first argument.

**discretisation** changes the default number of points used to display the curves. The default value is 30.

Example:
Warning: don’t confuse isos and discretisation with the geometric commands nbisos and discr.

**orientation, complement, invert, normals, range**

Syntax:

```
orientation name [name ...] F/R/E/I
complement name [name ...]
invert name
normals s (length = 10), disp normals
range name value value
```

- **orientation** – assigns the orientation of simple and complex shapes to one of the following four values: FORWARD, REVERSED, INTERNAL, EXTERNAL.
- **complement** – changes the current orientation of shapes to its complement: FORWARD to REVERSED and INTERNAL to EXTERNAL.
- **invert** – creates a copy of the original shape with a reversed orientation of all subshapes. For example, it may be useful to reverse the normals of a solid.
- *normals** – returns the assignment of colors to orientation values.
- **range** – defines the length of a selected edge by defining the values of a starting point and an end point.

**Example:**

```
# to invert normals of a box
box b 10 20 30
normals b 5
invert b
normals b 5

# to assign a value to an edge
box b1 10 20 30
```
# to define the box as edges
explode b1 e
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_10 b_11 b_12
# to define as an edge
makedge e 1
# to define the length of the edge as starting from 0 and finishing at 1
range e 0 1

**explode, exwire, nbshapes**

Syntax:

```
explode name [C/So/Sh/F/W/E/V]
exwire name
nbshapes name
```

**explode** extracts subshapes from an entity. The subshapes will be named `name_1, name_2, ...` Note that they are not copied but shared with the original.

With name only, **explode** will extract the first sublevel of shapes: the shells of a solid or the edges of a wire, for example. With one argument, **explode** will extract all subshapes of that type: `C` for compounds, `So` for solids, `Sh` for shells, `F` for faces, `W` for wires, `E` for edges, `V` for vertices.

**exwire** is a special case of **explode** for wires, which extracts the edges in an ordered way, if possible. Each edge, for example, is connected to the following one by a vertex.

**nbshapes** counts the number of shapes of each type in an entity.

**Example:**

```
# on a box
box b 10 20 30

# whatis returns the type and various information
whatis b
```
= b is a shape SOLID FORWARD Free Modified

# make one shell
explode b
whatis b_1
= b_1 is a shape SHELL FORWARD Modified Orientable Closed

# extract the edges b_1, ... , b_12
explode b e
e
==b_1 ... b_12

# count subshapes
nbshapes b
==
Number of shapes in b
VERTEX : 8
EDGE : 12
WIRE : 6
FACE : 6
SHELL : 1
SOLID : 1
COMPSOLID : 0
COMPOUND : 0
SHAPE : 34

emptycopy, add, compound

Syntax:

emptycopy [newname] name
add name toname
compound [name ...] compoundname

emptycopy returns an empty shape with the same orientation, location, and geometry as the target shape, but with no sub-shapes. If the newname argument is not given, the new shape is stored with the same name. This command is used to modify a frozen shape. A frozen shape is
a shape used by another one. To modify it, you must emptycopy it. Its subshape may be reinserted with the add command.

add inserts shape C into shape S. Verify that C and S reference compatible types of objects:

- Any Shape can be added to a Compound.
- Only a Solid can be added to a CompSolid.
- Only a Shell can Edge or a Vertex can be added into a Solid.
- Only a Face can be added to a Shell.
- Only a Wire and Vertex can be added in a Solid.
- Only an Edge can be added to a Wire.
- Only a Vertex can be added to an Edge.
- Nothing can be added to a Vertex.

emptycopy and add should be used with care.

On the other hand, compound is a safe way to achieve a similar result. It creates a compound from shapes. If no shapes are given, the compound is empty.

Example:

```
# a compound with three boxes
box b1 0 0 0 1 1 1
box b2 3 0 0 1 1 1
box b3 6 0 0 1 1 1
compound b1 b2 b3 c
```

compare

Syntax:

```
compare shape1 shape2
```

compare compares the two shapes shape1 and shape2 using the methods TopoDS_Shape::IsSame() and TopoDS_Shape::IsEqual().

Example
box b1 1 1 1
copy b1 b2
compare b1 b2
# same shapes
# equal shapes

orientation b2 R
compare b1 b2
# same shapes

box b2 1 1 1
compare b1 b2
# shapes are not same

issubshape

Syntax:

issubshape subshape shape

issubshape checks if the shape subshape is sub-shape of the shape shape and gets its index in the shape.

Example

box b 1 1 1
explode b f
issubshape b_2 b
# b_2 is sub-shape of b. Index in the shape: 2.
Curve and surface topology

This group of commands is used to create topology from shapes and to extract shapes from geometry.

- To create vertices, use the `vertex` command.
- To create edges use, the `edge`, `mkedge` commands.
- To create wires, use the `wire`, `polyline`, `polyvertex` commands.
- To create faces, use the `mkplane`, `mkface` commands.
- To extract the geometry from edges or faces, use the `mkcurve` and `mkface` commands.
- To extract the 2d curves from edges or faces, use the `pcurve` command.

**vertex**

Syntax:

```
vertex name [x y z / p edge]
```

Creates a vertex at either a 3d location x,y,z or the point at parameter p on an edge.

**Example:**

```
vertex v1 10 20 30
```

**mkpoint**

Syntax:

```
mkpoint name vertex
```

Creates a point from the coordinates of a given vertex.

**Example:**

```
mkpoint p v1
```
**edge, mkedge, uiisoedge, visoedge**

Syntax:

```plaintext
edge name vertex1 vertex2
mkedge edge curve [surface] [pfirst plast] [vfirst
   [pfirst] vlast [plast]]
uiisoedge edge face u v1 v2
visoedge edge face v u1 u2
```

- **edge** creates a straight line edge between two vertices.
- **mkedge** generates edges from curves. Two parameters can be given for the vertices: the first and last parameters of the curve are given by default. Vertices can also be given with their parameters, this option allows blocking the creation of new vertices. If the parameters of the vertices are not given, they are computed by projection on the curve. Instead of a 3d curve, a 2d curve and a surface can be given.

**Example:**

```
# straight line edge
vertex v1 10 0 0
vertex v2 10 10 0
edge e1 v1 v2

# make a circular edge
circle c 0 0 0 5
mkedge e2 c 0 pi/2

# A similar result may be achieved by trimming the curve
# The trimming is removed by mkedge
trim c c 0 pi/2
mkedge e2 c
```

- **visoedge** and **uiisoedge** are commands that generate a **uiiso** parameter edge or a **viso** parameter edge.
Example:

```
# to create an edge between v1 and v2 at point u
# to create the example plane
plane p
trim p p 0 1 0 1
convert p p
incudeg p 3
incvdeg p 3
movep p 2 2 0 0 1
movep p 3 3 0 0 0.5
mkface p p
# to create the edge in the plane at the u axis point 0.5, and between the v axis points v=0.2 and v =0.8
uiisoedge e p 0.5 0.20 0.8
```

**wire, polyline, polyvertex**

**Syntax:**

```
wire wirename e1/w1 [e2/w2 ...]
polyline name x1 y1 z1 x2 y2 z2 ...
polyvertex name v1 v2 ...
```

**wire** creates a wire from edges or wires. The order of the elements should ensure that the wire is connected, and vertex locations will be compared to detect connection. If the vertices are different, new edges will be created to ensure topological connectivity. The original edge may be copied in the new one.

**polyline** creates a polygonal wire from point coordinates. To make a closed wire, you should give the first point again at the end of the argument list.

**polyvertex** creates a polygonal wire from vertices.

**Example:**

```
# create two polygonal wires
```
profile

Syntax

profile name [code values] [code values] ...

profile builds a profile in a plane using a moving point and direction. By default, the profile is closed and a face is created. The original point is 0 0, and direction is 1 0 situated in the XY plane.

<table>
<thead>
<tr>
<th>Code</th>
<th>Values **</th>
<th>**Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>X Y Z</td>
<td>Sets the origin of the plane</td>
</tr>
<tr>
<td>P</td>
<td>DX DY DZ UX UY UZ</td>
<td>Sets the normal and X of the plane</td>
</tr>
<tr>
<td>F</td>
<td>X Y</td>
<td>Sets the first point</td>
</tr>
<tr>
<td>X</td>
<td>DX</td>
<td>Translates a point along X</td>
</tr>
<tr>
<td>Y</td>
<td>DY</td>
<td>Translates a point along Y</td>
</tr>
<tr>
<td>L</td>
<td>DL</td>
<td>Translates a point along direction</td>
</tr>
<tr>
<td>XX</td>
<td>X</td>
<td>Sets point X coordinate</td>
</tr>
<tr>
<td>YY</td>
<td>Y</td>
<td>Sets point Y coordinate</td>
</tr>
<tr>
<td>T</td>
<td>DX DY</td>
<td>Translates a point</td>
</tr>
<tr>
<td>TT</td>
<td>X Y</td>
<td>Sets a point</td>
</tr>
<tr>
<td>R</td>
<td>Angle</td>
<td>Rotates direction</td>
</tr>
<tr>
<td>RR</td>
<td>Angle</td>
<td>Sets direction</td>
</tr>
<tr>
<td>D</td>
<td>DX DY</td>
<td>Sets direction</td>
</tr>
<tr>
<td>IX</td>
<td>X</td>
<td>Intersects with vertical</td>
</tr>
<tr>
<td>IY</td>
<td>Y</td>
<td>Intersects with horizontal</td>
</tr>
<tr>
<td>C</td>
<td>Radius Angle</td>
<td>Arc of circle tangent to direction</td>
</tr>
</tbody>
</table>

Codes and values are used to define the next point or change the direction. When the profile changes from a straight line to a curve, a
tangent is created. All angles are in degrees and can be negative.

The point [code values] can be repeated any number of times and in any order to create the profile contour.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No suffix</td>
<td>Makes a closed face</td>
</tr>
<tr>
<td>W</td>
<td>Make a closed wire</td>
</tr>
<tr>
<td>WW</td>
<td>Make an open wire</td>
</tr>
</tbody>
</table>

The profile shape definition is the suffix; no suffix produces a face, w is a closed wire, ww is an open wire.

Code letters are not case-sensitive.

**Example:**

```
# to create a triangular plane using a vertex at the origin, in the xy plane
profile p 0 0 0 0 x 1 y 0 x 1 y 1
```

**Example:**

```
# to create a contour using the different code possibilities

# two vertices in the xy plane
profile p F 1 0 x 2 y 1 ww

# to view from a point normal to the plane top

# add a circular element of 45 degrees
profile p F 1 0 x 2 y 1 c 1 45 ww

# add a tangential segment with a length value 1
profile p F 1 0 x 2 y 1 c 1 45 1 1 ww
```
# add a vertex with xy values of 1.5 and 1.5
profile p F 1 0 x 2 y 1 c 1 45 l 1 tt 1.5 1.5 ww

# add a vertex with the x value 0.2, y value is constant
profile p F 1 0 x 2 y 1 c 1 45 l 1 tt 1.5 1.5 xx 0.2 ww

# add a vertex with the y value 2 x value is constant
profile p F 1 0 x 2 y 1 c 1 45 l 1 tt 1.5 1.5 yy 2ww

# add a circular element with a radius value of 1 and a circular value of 290 degrees
profile p F 1 0 x 2 y 1 c 1 45 l 1 tt 1.5 1.5 xx 0.2 yy 2 c 1 290

# wire continues at a tangent to the intersection x = 0
profile p F 1 0 x 2 y 1 c 1 45 l 1 tt 1.5 1.5 xx 0.2 yy 2 c 1 290 ix 0 ww

# continue the wire at an angle of 90 degrees until it intersects the y axis at y= -0.3
profile p F 1 0 x 2 y 1 c 1 45 l 1 tt 1.5 1.5 xx 0.2 yy 2 c 1 290 ix 0 r 90 ix -0.3 ww

# close the wire
profile p F 1 0 x 2 y 1 c 1 45 l 1 tt 1.5 1.5 xx 0.2 yy 2 c 1 290 ix 0 r 90 ix -0.3 w

# to create the plane with the same contour
profile p F 1 0 x 2 y 1 c 1 45 l 1 tt 1.5 1.5 xx 0.2 yy 2 c 1 290 ix 0 r 90 ix -0.3

bsplineprof

Syntax:
bsplineprof name [S face] [W Ww]

- for an edge: <digitizes> ... <mouse button 2>
- to end profile: <mouse button 3>

Builds a profile in the XY plane from digitizes. By default the profile is closed and a face is built.

**bsplineprof** creates a 2d profile from bspline curves using the mouse as the input. *MB1* creates the points, *MB2* finishes the current curve and starts the next curve, *MB3* closes the profile.

The profile shape definition is the suffix; no suffix produces a face, **w** is a closed wire, **ww** is an open wire.

**Example:**

```
# to view the xy plane
top
# to create a 2d curve with the mouse
bsplineprof res
# click mb1 to start the curve
# click mb1 to create the second vertex
# click mb1 to create a curve
==
# click mb2 to finish the curve and start a new curve
==
# click mb1 to create the second curve
# click mb3 to create the face
```

**mkoffset**

**mkoffset** creates a parallel wire in the same plane using a face or an existing continuous set of wires as a reference. The number of occurrences is not limited. The offset distance defines the spacing and the positioning of the occurrences.

**Syntax:**
mkoffset result shape nboffset stepoffset
  [jointype(a/i) [alt]]

where:

- **result** - the base name for the resulting wires. The index of the occurrence (starting with 1) will be added to this name, so the resulting wires will have the names - result_1, result_2 ...;
- **shape** - input shape (face or compound of wires);
- **nboffset** - the number of the parallel occurrences;
- **stepoffset** - offset distance between occurrences;
- **jointype(a/i)** - join type (a for arc (default) and i for intersection);
- **alt** - altitude from the plane of the input face in relation to the normal to the face.

**Example:**

```plaintext
# Create a box and select a face
box b 1 2 3
explode b f
# Create three exterior parallel contours with an offset value of 2
mkoffset r b_1 3 2
# wires r_1, r_2 and r_3 are created

# Create three exterior parallel contours with an offset value of 2 without round corners
mkoffset r b_1 3 2 i
# wires r_1, r_2 and r_3 are created

# Create one interior parallel contour with an offset value of 0.4
mkoffset r b_1 1 -0.4
```

**Note** that on a concave input contour for an interior step mkoffset command may produce several wires which will be contained in a single compound.

**Example:**
# to create the example contour
profile p F 0 0 x 2 y 4 tt 1 1 tt 0 4 w
# creates an incoherent interior offset
mkoffset r p 1 -0.50

# creates two incoherent wires
mkoffset r p 1 -0.55
# r_1 is a compound of two wires

**mkplane, mkface**

Syntax:

```
mkplane name wire
mkface name surface [ufirst ulast vfirst vlast]
```

**mkplane** generates a face from a planar wire. The planar surface will be constructed with an orientation which keeps the face inside the wire.

**mkface** generates a face from a surface. Parameter values can be given to trim a rectangular area. The default boundaries are those of the surface.

**Example:**

```
# make a polygonal face
polyline f 0 0 0 20 0 0 20 10 0 10 10 0 10 20 0 0 20
       0 0 0 0
mkplane f f

# make a cylindrical face
cylinder g 10
trim g g -pi/3 pi/2 0 15
mkface g g
```

**mkcurve, mksurface**

Syntax:
**mkcurve** creates a 3d curve from an edge. The curve will be trimmed to the edge boundaries.

**mksurface** creates a surface from a face. The surface will not be trimmed.

**Example:**

```
# make a line
vertex v1 0 0 0
vertex v2 10 0 0
edge e v1 v2
```

**pcurve**

Syntax:

```
pcurve [name edgename] facename
```

Extracts the 2d curve of an edge on a face. If only the face is specified, the command extracts all the curves and colors them according to their orientation. This is useful in checking to see if the edges in a face are correctly oriented, i.e. they turn counter-clockwise. To make curves visible, use a fitted 2d view.

**Example:**

```
# view the pcurves of a face
plane p
trim p p -1 1 -1 1
mkface p p
av2d; # a 2d view
pcurve p
2dfit
```

**chfi2d**
Syntax:

\[
\text{chfi2d result face [edge1 edge2 (F radius/CDD d1
d2/CDA d ang) ....}
\]

Creates chamfers and fillets on 2D objects. Select two adjacent edges and:

- a radius value
- two respective distance values
- a distance value and an angle

The radius value produces a fillet between the two faces.

The distance is the length value from the edge between the two selected faces in a normal direction.

**Example:**

Let us create a 2d fillet:

```
top
profile p x 2 y 2 x -2
chfi2d cfr p . . F 0.3
==Pick an object
#select an edge
==Pick an object
#select an edge
```

Let us create a 2d chamfer using two distances:

```
profile p x 2 y 2 x -2
chfi2d cfr p . . CDD 0.3 0.6
==Pick an object
#select an edge
==Pick an object
#select an edge
```

Let us create a 2d chamfer using a defined distance and angle
nproject

Syntax:

nproject pj e1 e2 e3 ... surf -g -d [dmax] [Tol [continuity [maxdeg [maxseg]]]]

Creates a shape projection which is normal to the target surface.

Example:

# create a curved surface
line l 0 0 0 1 0 0
trim l l 0 2
convert l l

incdeg l 3
cmovep l 1 0 0.5 0
cmovep l 3 0 0.5 0
copy l l l
translate ll 2 -0.5 0
mkedge e1 l
mkedge e2 ll
wire w e1 e2
prism p w 0 0 3
don1 p
#display in four views
mu4
fit
# create the example shape
circle c 1.8 -0.5 1 0 1 0 1 0 0 0.4
mkedge e c
donly p e
# create the normal projection of the shape(circle)
nproject r e p
Primitives

Primitive commands make it possible to create simple shapes. They include:

- **box** and **wedge** commands.
- **pcylinder**, **pcone**, **psphere**, **ptorus** commands.
- **halfspace** command

**box, wedge**

Syntax:

```
box name [x y z] dx dy dz
wedge name dx dy dz ltx / xmin zmin xmax zmax
```

**box** creates a box parallel to the axes with dimensions \(dx,dy,dz\). \(x,y,z\) is the corner of the box. It is the default origin.

**wedge** creates a box with five faces called a wedge. One face is in the OXZ plane, and has dimensions \(dx,dz\) while the other face is in the plane \(y = dy\). This face either has dimensions \(ltx,dz\) or is bounded by \(xmin,zmin,xmax,zmax\).

The other faces are defined between these faces. The face in the \(y = yd\) plane may be degenerated into a line if \(ltx = 0\), or a point if \(xmin = xmax\) and \(ymin = ymax\). In these cases, the line and the point both have 5 faces each. To position the wedge use the **translate** and **trotate** commands.

**Example:**

```
# a box at the origin
box b1 10 20 30

# another box
box b2 30 30 40 10 20 30
```
# a wedge
wedge w1 10 20 30 5

# a wedge with a sharp edge (5 faces)
wedge w2 10 20 30 0

# a pyramid
wedge w3 20 20 20 10 10 10 10

**pcylinder, pcone, psphere, ptorus**

Syntax:

pcylinder name [plane] radius height [angle]
pcone name [plane] radius1 radius2 height [angle]
psphere name [plane] radius1 [angle1 angle2] [angle]
ptorus name [plane] radius1 radius2 [angle1 angle2]

All these commands create solid blocks in the default coordinate system, using the Z axis as the axis of revolution and the X axis as the origin of the angles. To use another system, translate and rotate the resulting solid or use a plane as first argument to specify a coordinate system. All primitives have an optional last argument which is an angle expressed in degrees and located on the Z axis, starting from the X axis. The default angle is 360.

**pcylinder** creates a cylindrical block with the given radius and height.

**pcone** creates a truncated cone of the given height with radius1 in the plane \( z = 0 \) and radius2 in the plane \( z = \text{height} \). Neither radius can be negative, but one of them can be null.

**psphere** creates a solid sphere centered on the origin. If two angles, \( angle1 \) and \( angle2 \), are given, the solid will be limited by two planes at latitude \( angle1 \) and \( angle2 \). The angles must be increasing and in the range -90,90.
**ptorus** creates a solid torus with the given radii, centered on the origin, which is a point along the z axis. If two angles increasing in degree in the range 0 – 360 are given, the solid will be bounded by two planar surfaces at those positions on the circle.

**Example:**

```plaintext
# a can shape
pcylinder cy 5 10

# a quarter of a truncated cone
pcone co 15 10 10 90

# three-quarters of sphere
psphere sp 10 270

# half torus
ptorus to 20 5 0 90
```

**halfspace**

Syntax:

```
halfspace result face/shell x y z
```

**halfspace** creates an infinite solid volume based on a face in a defined direction. This volume can be used to perform the boolean operation of cutting a solid by a face or plane.

**Example:**

```plaintext
box b 0 0 0 1 2 3
explode b f
==b_1 b_2 b_3 b_4 b_5 b_6
halfspace hr b_3 0.5 0.5 0.5
```
Sweeping

Sweeping creates shapes by sweeping out a shape along a defined path:

- **prism** – sweeps along a direction.
- **revol** – sweeps around an axis.
- **pipe** – sweeps along a wire.
- **mksweep** and **buildsweep** – to create sweeps by defining the arguments and algorithms.
- **thursections** – creates a sweep from wire in different planes.

**prism**

Syntax:

```
prism result base dx dy dz [Copy | Inf | SemiInf]
```

Creates a new shape by sweeping a shape in a direction. Any shape can be swept: a vertex gives an edge; an edge gives a face; and a face gives a solid.

The shape is swept along the vector $dx \ dy \ dz$. The original shape will be shared in the result unless `Copy` is specified. If `Inf` is specified the prism is infinite in both directions. If `SemiInf` is specified the prism is infinite in the $dx,dy,dz$ direction, and the length of the vector has no importance.

**Example:**

```
# sweep a planar face to make a solid
polyline f 0 0 0 10 0 0 10 5 0 5 5 0 5 15 0 0 15 0 0
0 0
mkplane f f
```

**revol**

Syntax:

```
revol result base x y z dx dy dz angle [Copy]
```
Creates a new shape by sweeping a base shape through an angle along the axis \( x,y,z \) \( dx,dy,dz \). As with the prism command, the shape can be of any type and is not shared if \( \text{Copy} \) is specified.

**Example:**

```
# shell by wire rotation
polyline w 0 0 0 10 0 0 10 5 0 5 5 0 5 15 0 0 15 0
revol s w 20 0 0 0 1 0 90
```

**pipe**

Syntax:

```
pipe name wire_spine Profile
```

Creates a new shape by sweeping a shape known as the profile along a wire known as the spine.

**Example:**

```
# sweep a circle along a bezier curve to make a solid pipe

beziercurve spine 4 0 0 0 10 0 0 10 10 0 20 10 0
mkedge spine spine
wire spine spine
circle profile 0 0 0 1 0 0 2
mkedge profile profile
wire profile profile
mkplane profile profile
pipe p spine spine profile
```

**mksweep, addsweep, setsweep, deletesweep, buildsweep, simulsweep**

Syntax:

```
mksweep wire
```
addsweep wire[vertex][-M][-C] [auxiliaryshape]
deletesweep wire
setsweep options [arg1 [arg2 [...]]]
simulsweep r [n] [option]
buildsweep [r] [option] [Tol]

options are:

- **-FR**: Tangent and Normal are defined by a Frenet trihedron
- **-CF**: Tangent is given by Frenet, the Normal is computed to minimize the torsion
- **-DX Surf**: Tangent and Normal are given by Darboux trihedron, surf must be a shell or a face
- **-CN dx dy dz**: BiNormal is given by $dx$ $dy$ $dz$
- **-FX Tx Ty TZ [Nx Ny Nz]**: Tangent and Normal are fixed
- **-G guide**

These commands are used to create a shape from wires. One wire is designated as the contour that defines the direction; it is called the spine. At least one other wire is used to define the sweep profile.

- **mksweep** – initializes the sweep creation and defines the wire to be used as the spine.
- **addsweep** – defines the wire to be used as the profile.
- **deletesweep** – cancels the choice of profile wire, without leaving the mksweep mode. You can re-select a profile wire.
- **setsweep** – commands the algorithms used for the construction of the sweep.
- **simulsweep** – can be used to create a preview of the shape. [n] is the number of sections that are used to simulate the sweep.
- **buildsweep** – creates the sweep using the arguments defined by all the commands.

**Example:**

# create a sweep based on a semi-circular wire using the Frenet algorithm
# create a circular figure
circle c2 0 0 0 1 0 0 10
trim c2 c2 -pi/2 pi/2
mksweep w
# to display all the options for a sweep
setsweep
# to create a sweep using the Frenet algorithm where
the
# normal is computed to minimise the torsion
setsweep -CF
addsweep w -R
# to simulate the sweep with a visual approximation
simulsweep w 3

thrusections

Syntax:

thrusections [-N] result issolid isruled wire1 wire2 [...wire...]

thrusections creates a shape using wires that are positioned in different planes. Each wire selected must have the same number of edges and vertices. A bezier curve is generated between the vertices of each wire. The option [-N] means that no check is made on wires for direction.

Example:

# create three wires in three planes
polyline w1 0 0 0 5 0 0 5 5 0 2 3 0
polyline w2 0 1 3 4 1 3 4 4 3 1 3 3
polyline w3 0 0 5 5 0 5 5 5 5 2 3 5
# create the shape
thrusections th issolid isruled w1 w2 w3
==thrusections th issolid isruled w1 w2 w3
Tolerances obtenues -- 3d : 0
- 2d : 0
Topological transformation

Transformations are applications of matrices. When the transformation is nondeforming, such as translation or rotation, the object is not copied. The topology localcoordinate system feature is used. The copy can be enforced with the `tcopy` command.

- **tcp**y – makes a copy of the structure of a shape.
- **ttranslate**, **trotate**, **tmove** and **reset** – move a shape.
- **tmirror** and **tscale** – always modify the shape.

**tcp**

Syntax:

```
tcopy name toname [name toname ...]
```

Copies the structure of one shape, including the geometry, into another, newer shape.

**Example:**

```plaintext
# create an edge from a curve and copy it
beziercurve c 3 0 0 0 10 0 0 20 10 0
mkedge e1 c
ttranslate e1 0 5 0
tcopy e1 e2
ttranslate e2 0 5 0
# now modify the curve, only e1 and e2 will be modified
```

**tmove**, **treset**

Syntax:

```
tmove name [name ...] shape
reset name [name ...]
```
**tmove** and **reset** modify the location, or the local coordinate system of a shape.

**tmove** applies the location of a given shape to other shapes. **reset** restores one or several shapes it to its or their original coordinate system(s).

**Example:**

```tcl
# create two boxes
box b1 10 10 10
box b2 20 0 0 10 10 10
# translate the first box
ttranslate b1 0 10 0
# and apply the same location to b2
tmove b2 b1
# return to original positions
reset b1 b2
```

**ttranslate, trotate**

Syntax:

```tcl
 ttranslate [name ...] dx dy dz
trotate [name ...] x y z dx dy dz angle
```

**ttranslate** translates a set of shapes by a given vector, and **trotate** rotates them by a given angle around an axis. Both commands only modify the location of the shape. When creating multiple shapes, the same location is used for all the shapes. (See **toto.tcl** example below. Note that the code of this file can also be directly executed in interactive mode.)

Locations are very economic in the data structure because multiple occurrences of an object share the topological description.

**Example:**

```tcl
# make rotated copies of a sphere in between two
```
cylinders
# create a file source toto.tcl
# toto.tcl code:
for {set i 0} {$i < 360} {incr i 20} {
    copy s s$i
trotate s$i 0 0 0 0 0 1 $i
}

# create two cylinders
pcylinder c1 30 5
copy c1 c2
ttranslate c2 0 0 20

#create a sphere
psphere s 3
ttranslate s 25 0 12.5

# call the source file for multiple copies
source toto.tcl

tmirror, tscale

Syntax:

```tmirror name x y z dx dy dz
tscale name x y z scale```

- **tmirror** makes a mirror copy of a shape about a plane x,y,z dx,dy,dz.
- **Tsacle** applies a central homotopic mapping to a shape.

Example:

```# mirror a portion of cylinder about the YZ plane
pcylinder c1 10 10 270
copy c1 c2
tmirror c2 15 0 0 1 0 0
# and scale it
tscale c1 0 0 0 0.5```
Old Topological operations

- **fuse**, **cut**, **common** are boolean operations.
- **section**, **psection** compute sections.
- **sewing** joins two or more shapes.

### fuse, cut, common

**Syntax:**

```
fuse name shape1 shape2
cut name shape1 shape2
common name shape1 shape2
```

**fuse** creates a new shape by a boolean operation on two existing shapes. The new shape contains both originals intact.

**cut** creates a new shape which contains all parts of the second shape but only the first shape without the intersection of the two shapes.

**common** creates a new shape which contains only what is in common between the two original shapes in their intersection.

**Example:**

```
# all four boolean operations on a box and a cylinder
box b 0 -10 5 20 20 10
pcylinder c 5 20

fuse s1 b c
ttranslate s1 40 0 0

cut s2 b c
ttranslate s2 -40 0 0

cut s3 c b
```
section, psection

Syntax:

section result shape1 shape2
psection name shape plane

section creates a compound object consisting of the edges for the intersection curves on the faces of two shapes.

psection creates a planar section consisting of the edges for the intersection curves on the faces of a shape and a plane.

Example:

# section line between a cylinder and a box
pcylinder c 10 20
box b 0 0 5 15 15 15
trotate b 0 0 0 1 1 1 20
section s b c

# planar section of a cone
pcone c 10 30 30
plane p 0 0 15 1 1 2
psection s c p

sewing

Syntax:

sewing result [tolerance] shape1 shape2 ...

Sewing joins shapes by connecting their adjacent or near adjacent edges. Adjacency can be redefined by modifying the tolerance value.
Example:

```plaintext
# create two adjacent boxes
box b 0 0 0 1 2 3
box b2 0 2 0 1 2 3
sewing sr b b2
whatis sr
sr is a shape COMPOUND FORWARD Free Modified
```
New Topological operations

The new algorithm of Boolean operations avoids a large number of weak points and limitations presented in the old boolean operation algorithm.

**bparallelmode**

- **bparallelmode** enable or disable parallel mode for boolean operations. Sequential computing is used by default.

Syntax:

```
bparallelmode [1/0]
```

Without arguments, bparallelmode shows current state of parallel mode for boolean operations.

- 0 Disable parallel mode,
- 1 Enable parallel mode

Example:

```
# Enable parallel mode for boolean operations.
bparallelmode 1

# Show state of parallel mode for boolean operations.
bparallelmode
```

**bop, bopfuse, bopcut, boptuc, bopcommon**

- **bop** defines *shape1* and *shape2* subject to ulterior Boolean operations
- **bopfuse** creates a new shape by a boolean operation on two existing shapes. The new shape contains both originals intact.
- **bopcut** creates a new shape which contains all parts of the second shape but only the first shape without the intersection of the two shapes.
- **boptuc** is a reverced **bopcut**.
- **bopcommon** creates a new shape which contains only whatever is in common between the two original shapes in their intersection.

**Syntax:**

```
bop shape1 shape2
bopcommon result
bopfuse result
bopcut result
boptuc result
```

These commands have short variants:

```
bcommon result shape1 shape2
bfuse result shape1 shape2
bcut result shape1 shape2
```

**bop** fills data structure (DS) of boolean operation for `shape1` and `shape2`. **bopcommon, bopfuse, bopcut, boptuc** commands are used after **bop** command. After one **bop** command it is possible to call several commands from the list above. For example:

```
bop S1 S2
bopfuse R
```

**Example:**

Let us produce all four boolean operations on a box and a cylinder:

```
box b 0 -10 5 20 20 10
pcylinder c 5 20

# fills data structure
bop b c

bopfuse s1
ttranslate s1 40 0 0

bopcut s2
```
Now use short variants of the commands:

```
bfuse s11 b c
ttranslate s11 40 0 100

bcut s12 b c
ttranslate s12 -40 0 100

bcommon s14 b c
ttranslate s14 0 -40 100
```

**bopsection**

Syntax:

```
bop shape1 shape2
bopsection result
```

- **bopsection** – creates a compound object consisting of the edges for the intersection curves on the faces of two shapes.
- **bop** – fills data structure (DS) of boolean operation for *shape1* and *shape2*.
- **bopsection** – is used after **bop** command.

Short variant syntax:

```
bsection result shape1 shape2 [-2d/-2d1/-2s2] [-a]
```

- `-2d` – PCurves are computed on both parts.
- `-2d1` – PCurves are computed on first part.
- `-2d2` – PCurves are computed on second part.
- -a – built geometries are approximated.

Example:

Let us build a section line between a cylinder and a box

\[
\begin{align*}
\text{pcylinder} & \ c 10 \ 20 \\
\text{box} & \ b 0 \ 0 \ 5 \ 15 \ 15 \ 15 \\
\text{trotate} & \ b 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 20 \\
\text{bop} & \ b \ c \\
\text{bopsection} & \ s \\
\# \ Short \ variant: \\
\text{bsection} & \ s2 \ b \ c
\end{align*}
\]

bopcheck, bopargshape

Syntax:

\[
\begin{align*}
\text{bopcheck} & \ \text{shape} \\
\text{bopargcheck} & \ \text{shape1} \ [[\text{shape2}] \ [-F/O/C/T/S/U]] \\
& \ [[/R|F|T|V|E|I|P]] \ [#BF]
\end{align*}
\]

bopcheck checks a shape for self-interference.

bopargcheck checks the validity of argument(s) for boolean operations.

- Boolean Operation – (by default a section is made) :
  - F (fuse)
  - O (common)
  - C (cut)
  - T (cut21)
  - S (section)
  - U (unknown)
- Test Options – (by default all options are enabled) :
  - R (disable small edges (shrink range) test)
  - F (disable faces verification test)
  - T (disable tangent faces searching test)
  - V (disable test possibility to merge vertices)
  - E (disable test possibility to merge edges)
  - I (disable self-interference test)
- **P** (disable shape type test)
- **Additional Test Options:**
  - **B** (stop test on first faulty found) – by default it is off;
  - **F** (full output for faulty shapes) – by default the output is made in a short format.

**Note** that Boolean Operation and Test Options are used only for a couple of argument shapes, except for **I** and **P** options that are always used to test a couple of shapes as well as a single shape.

**Example:**

```plaintext
# checks a shape on self-interference
box b1 0 0 0 1 1 1
bopcheck b1

# checks the validity of argument for boolean cut operations
box b2 0 0 0 10 10 10
bopargcheck b1 b2 -C
```
Drafting and blending

Drafting is creation of a new shape by tilting faces through an angle.

Blending is the creation of a new shape by rounding edges to create a fillet.

- Use the depouille command for drafting.
- Use the chamf command to add a chamfer to an edge
- Use the blend command for simple blending.
- Use bfuseblend for a fusion + blending operation.
- Use bcutblend for a cut + blending operation.
- Use buildvol, mkevol, updatevol to realize varying radius blending.

depouille

Syntax:

```
dep result shape dirx diry dirz face angle x y x dx dy dz [face angle...]
```

Creates a new shape by drafting one or more faces of a shape.

Identify the shape(s) to be drafted, the drafting direction, and the face(s) with an angle and an axis of rotation for each face. You can use dot syntax to identify the faces.

Example:

```
# draft a face of a box
box b 10 10 10
explode b f
== b_1 b_2 b_3 b_4 b_5 b_6

dep a b 0 0 1 b_2 10 0 10 0 1 0 5
```

chamf
Syntax:

```
chamf newname shape edge face S dist
chamf newname shape edge face dist1 dist2
chamf newname shape edge face A dist angle
```

Creates a chamfer along the edge between faces using:

- a equal distances from the edge
- the edge, a face and distance, a second distance
- the edge, a reference face and an angle

Use the dot syntax to select the faces and edges.

**Examples:**

Let us create a chamfer based on equal distances from the edge (45 degree angle):

```
# create a box
box b 1 2 3
chamf ch b . . S 0.5
==Pick an object
# select an edge
==Pick an object
# select an adjacent face
```

Let us create a chamfer based on different distances from the selected edge:

```
box b 1 2 3
chamf ch b . . 0.3 0.4
==Pick an object
# select an edge
==Pick an object
# select an adjacent face
```

Let us create a chamfer based on a distance from the edge and an angle:

```
box b 1 2 3
```
chamf ch b . . A 0.4 30
==Pick an object
# select an edge
==Pick an object
# select an adjacent face

blend

Syntax:

blend result object rad1 ed1 rad2 ed2 ... [R/Q/P]

Creates a new shape by filleting the edges of an existing shape. The edge must be inside the shape. You may use the dot syntax. Note that the blend is propagated to the edges of tangential planar, cylindrical or conical faces.

Example:

# blend a box, click on an edge
box b 20 20 20
blend b b 2 .
==tolerance ang : 0.01
==tolerance 3d : 0.0001
==tolerance 2d : 1e-05
==fleche : 0.001
==tolblend 0.01 0.0001 1e-05 0.001
==Pick an object
# click on the edge you want ot fillet

==COMPUTE: temps total 0.1s dont :
==- Init + ExtentAnalyse 0s
==- PerformSetOfSurf 0.02s
==- PerformFilletOnVertex 0.02s
==- FilDS 0s
==- Reconstruction 0.06s
==- SetRegul 0s
bfuseblend

Syntax:

```
bfuseblend name shape1 shape2 radius [-d]
```

Creates a boolean fusion of two shapes and then blends (fillets) the intersection edges using the given radius. Option [-d] enables the Debugging mode in which the error messages, if any, will be printed.

Example:

```
# fuse-blend two boxes
box b1 20 20 5
copy b1 b2
ttranslate b2 -10 10 3
bfuseblend a b1 b2 1
```

bcutblend

Syntax:

```
bcutblend name shape1 shape2 radius [-d]
```

Creates a boolean cut of two shapes and then blends (fillets) the intersection edges using the given radius. Option [-d] enables the Debugging mode in which the error messages, if any, will be printed.

Example:

```
# cut-blend two boxes
box b1 20 20 5
copy b1 b2
ttranslate b2 -10 10 3
bcutblend a b1 b2 1
```

mkevol, updatevol, buildevol

Syntax:
mkevol result object (then use updatevol) [R/Q/P]
updatevol edge u1 radius1 [u2 radius2 ...]
buildevol

These three commands work together to create fillets with evolving radii.

- **mkevol** allows specifying the shape and the name of the result. It returns the tolerances of the fillet.
- **updatevol** allows describing the filleted edges you want to create. For each edge, you give a set of coordinates: parameter and radius and the command prompts you to pick the edge of the shape which you want to modify. The parameters will be calculated along the edges and the radius function applied to the whole edge.
- **buildevol** produces the result described previously in **mkevol** and **updatevol**.

Example:

```
# makes an evolved radius on a box
box b 10 10 10
mkevol b b
==tolerance ang : 0.01
==tolerance 3d : 0.0001
==tolerance 2d : 1e-05
==fleche : 0.001
==tolblend 0.01 0.0001 1e-05 0.001

# click an edge
updatevol . 0 1 1 3 2 2
==Pick an object

buildevol
==Dump of SweepApproximation
==Error 3d = 1.28548881203818e-14
==Error 2d = 1.3468326936926e-14 ,
==1.2029229999388e-14
==2 Segment(s) of degree 3
```
==COMPUTE: temps total 0.91s dont :
  ==-  Init + ExtentAnalyse 0s
  ==-  PerformSetOfSurf 0.33s
  ==-  PerformFilletOnVertex 0.53s
  ==-  FilDS 0.01s
  ==-  Reconstruction 0.04s
  ==-  SetRegul 0s
Analysis of topology and geometry

Analysis of shapes includes commands to compute length, area, volumes and inertial properties, as well as to compute some aspects impacting shape validity.

- Use `lprops`, `sprops`, `vprops` to compute integral properties.
- Use `bounding` to display the bounding box of a shape.
- Use `distmini` to calculate the minimum distance between two shapes.
- Use `xdistef`, `xdistcs`, `xdistcc`, `xdistc2dc2dss`, `xdistcc2ds` to check the distance between two objects on even grid.
- Use `checkshape` to check validity of the shape.
- Use `tolsphere` to see the tolerance spheres of all vertices in the shape.
- Use `validrange` to check range of an edge not covered by vertices.

**lprops, sprops, vprops**

Syntax:

```
lprops shape
sprops shape
vprops shape
```

- `lprops` computes the mass properties of all edges in the shape with a linear density of 1;
- `sprops` of all faces with a surface density of 1;
- `vprops` of all solids with a density of 1.

All three commands print the mass, the coordinates of the center of gravity, the matrix of inertia and the moments. Mass is either the length, the area or the volume. The center and the main axis of inertia are displayed.

**Example:**

```
# volume of a cylinder
```
pcylinder c 10 20
vprops c
== results
Mass : 6283.18529981086

Center of gravity :
X = 4.1004749224903e-06
Y = -2.03392858349861e-16
Z = 9.9999999941362

Matrix of Inertia :
366519.141445068
   5.71451850691484e-12
0.257640437382627
5.71451850691484e-12       366519.141444962
2.26823064169991e-10       0.257640437382627
2.26823064169991e-10       314159.265358863

Moments :
IX = 366519.141446336
IY = 366519.141444962
I.Z = 314159.265357595

bounding

Syntax:

bounding shape

Displays the bounding box of a shape. The bounding box is a cuboid created with faces parallel to the x, y, and z planes. The command returns the dimension values of the box, xmin ymin zmin xmax ymax zmax.

Example:

# bounding box of a torus
ptorus t 20 5
distmini

Syntax:

```
distmini name Shape1 Shape2
```

Calculates the minimum distance between two shapes. The calculation returns the number of solutions, if more than one solution exists. The options are displayed in the viewer (red) and the results are listed in the shell window. The `distmini` lines are considered as shapes which have a value v.

Example:

```
box b 0 0 0 10 20 30
box b2 30 30 0 10 20 30
distmini d1 b b2
==the distance value is : 22.36067977499797
==the number of solutions is : 2

==solution number 1
==the type of the solution on the first shape is 0
==the type of the solution on the second shape is 0
==the coordinates of the point on the first shape are:
==X=10 Y=20 Z=30
==the coordinates of the point on the second shape are:
==X=30 Y=30 Z=30

==solution number 2:
```
==the type of the solution on the first shape is 0
==the type of the solution on the second shape is 0
==the coordinates of the point on the first shape are:
==X=10 Y=20 Z=0
==the coordinates of the point on the second shape are:
==X=30 Y=30 Z=0
==d1_val d1 d12

**xdistef, xdistcs, xdistcc, xdistc2dc2dss, xdistcc2ds**

Syntax:

```
xdistef edge face
xdistcs curve surface firstParam lastParam
    [NumberOfSamplePoints]
xdistcc curve1 curve2 startParam finishParam
    [NumberOfSamplePoints]
xdistcc2ds c curve2d surf startParam finishParam
    [NumberOfSamplePoints]
xdistc2dc2dss curve2d_1 curve2d_2 surface_1 surface_2
    startParam finishParam [NumberOfSamplePoints]
```

It is assumed that curves have the same parametrization range and
*startParam* is less than *finishParam*.

Commands with prefix *xdist* allow checking the distance between two
objects on even grid:

- **xdistef** – distance between edge and face;
- **xdistcs** – distance between curve and surface. This means that the
  projection of each sample point to the surface is computed;
- **xdistcc** – distance between two 3D curves;
- **xdistcc2ds** – distance between 3d curve and 2d curve on surface;
- **xdistc2dc2dss** – distance between two 2d curves on surface.

**Examples**
checkshape

Syntax:

```plaintext
checkshape [-top] shape [result] [-short]
```

Where:

- `top` – optional parameter, which allows checking only topological validity of a shape.
- `shape` – the only required parameter which represents the name of the shape to check.
- `result` – optional parameter which is the prefix of the output shape names.
- `short` – a short description of the check.

**checkshape** examines the selected object for topological and geometric coherence. The object should be a three dimensional shape.

**Example:**

```plaintext
# checkshape returns a comment valid or invalid
box b1 0 0 0 1 1 1
checkshape b1
# returns the comment
this shape seems to be valid
```

tolsphere

Syntax:

```plaintext
tolsphere shape
```
Where:

- *shape* – the name of the shape to process.

**tolsphere** shows vertex tolerances by drawing spheres around each vertex in the shape. Each sphere is assigned a name of the shape with suffix "_vXXX", where XXX is the number of the vertex in the shape.

**Example:**

```plaintext
# tolsphere returns all names of created spheres.
box b1 0 0 0 1 1 1
settolerance b1 0.05
tolsphere b1
# creates spheres and returns the names
b1_v1 b1_v2 b1_v3 b1_v4 b1_v5 b1_v6 b1_v7 b1_v8
```

**validrange**

Syntax:

```plaintext
validrange edge [(out) u1 u2]
```

Where:

- *edge* – the name of the edge to analyze.
- *u1, u2* – optional names of variables to put the range into.

**validrange** computes valid range of the edge. If *u1* and *u2* are not given it returns first and last parameters. Otherwise, it sets the variables *u1* and *u2*.

**Example:**

```plaintext
circle c 0 0 0 10
mkedge e c
mkedge e c 0 pi
validrange e
# returns the range
1.9884375000000002e-008 3.141592637054181
```
validrange e u1 u2
dval u1
1.9884375000000002e-008
dval u2
3.1415926337054181
Surface creation

Surface creation commands include surfaces created from boundaries and from spaces between shapes.

- **gplate** creates a surface from a boundary definition.
- **filling** creates a surface from a group of surfaces.

**gplate,**

Syntax:

```
gplate result nbrcurfront nbrpntconst [SurfInit]
    [edge 0] [edge tang (1:G1;2:G2) surf]...[point]
    [u v tang (1:G1;2:G2) surf] ...
```

Creates a surface from a defined boundary. The boundary can be defined using edges, points, or other surfaces.

**Example:**

```
plane p
trim p p -1 3 -1 3
mkface p p

bezicurve c1 3 0 0 0 1 0 1 2 0 0
mkedge e1 c1
tcopy e1 e2
tcopy e1 e3

ttranslate e2 0 2 0
trotate e3 0 0 0 0 0 1 90
tcopy e3 e4
ttranslate e4 2 0 0
# create the surface
gplate r1 4 0 p e1 0 e2 0 e3 0 e4 0
==
```
Results

DistMax=8.50014503228635e-16
* GEOMPLATE END*
Calculation time: 0.33
Loop number: 1
Approximation results
Approximation error : 2.06274907619957e-13
Criterium error : 4.97600631215754e-14

# to create a surface defined by edges and passing through a point
# to define the border edges and the point plane p
trim p p -1 3 -1 3
mkface p p

beziercurve c1 3 0 0 0 1 0 1 2 0 0
mkedge e1 c1
tcopy e1 e2
tcopy e1 e3
ttranslate e2 0 2 0
trotate e3 0 0 0 0 0 1 90
tcopy e3 e4
ttranslate e4 2 0 0
# to create a point
point pp 1 1 0
# to create the surface
gplate r2 4 1 p e1 0 e2 0 e3 0 e4 0 pp
=

Results

DistMax=3.65622157610934e-06
* GEOMPLATE END*
Calculation time: 0.27
Loop number: 1
Approximation results
Approximation error : 0.000422195884750181
filling, fillingparam

Syntax:

```
filling result nbB nbC nbP [SurfInit] [edge] [face]order...
edge[face]order... point/u v face order...
```

Creates a surface between borders. This command uses the gplate algorithm but creates a surface that is tangential to the adjacent surfaces. The result is a smooth continuous surface based on the G1 criterion.

To define the surface border:

- enter the number of edges, constraints, and points
- enumerate the edges, constraints and points

The surface can pass through other points. These are defined after the border definition.

You can use the fillingparam command to access the filling parameters.

The options are:

- `-l`: to list current values
- `-i`: to set default values
- `-rdeg nbPonC nbIt anis`: to set filling options
- `-c t2d t3d tang tcur`: to set tolerances
- `-a maxdeg maxseg`: Approximation option

Example:

```
# to create four curved survaces and a point plane p
trim p p -1 3 -1 3
mkface p p
beziercurve c1 3 0 0 0 1 0 1 2 0 0
```
mkedge e1 c1
tcopy e1 e2
tcopy e1 e3
ttranslate e2 0 2 0
trotate e3 0 0 0 0 0 1 90
tcopy e3 e4
ttranslate e4 2 0 0
point pp 1 1 0

prism f1 e1 0 -1 0
prism f2 e2 0 1 0
prism f3 e3 -1 0 0
prism f4 e4 1 0 0

# to create a tangential surface
filling r1 4 0 0 p e1 f1 1 e2 f2 1 e3 f3 1 e4 f4 1
# to create a tangential surface passing through
  point pp
filling r2 4 0 1 p e1 f1 1 e2 f2 1 e3 f3 1 e4 f4 1
  pp#
# to visualise the surface in detail
isos r2 40
# to display the current filling parameters
fillingparam -1
==
Degree = 3
NbPtsOnCur = 10
NbIter = 3
Anisotropie = 0
Tol2d = 1e-05
Tol3d = 0.0001
TolAng = 0.01
TolCurv = 0.1
MaxDeg = 8
MaxSegments = 9
Complex Topology

Complex topology is the group of commands that modify the topology of shapes. This includes feature modeling.

offsetshape, offsetcompshape

Syntax:

```
offsetshape r shape offset [tol] [face ...]
offsetcompshape r shape offset [face ...]
```

**offsetshape** and **offsetcompshape** assign a thickness to the edges of a shape. The *offset* value can be negative or positive. This value defines the thickness and direction of the resulting shape. Each face can be removed to create a hollow object.

The resulting shape is based on a calculation of intersections. In case of simple shapes such as a box, only the adjacent intersections are required and you can use the **offsetshape** command.

In case of complex shapes, where intersections can occur from non-adjacent edges and faces, use the **offsetcompshape** command. **comp** indicates complete and requires more time to calculate the result.

The opening between the object interior and exterior is defined by the argument face or faces.

**Example:**

```
box b1 10 20 30
explode b1 f
== b1_1 b1_2 b1_3 b1_4 b1_5 b1_6
offsetcompshape r b1 -1 b1_3
```

**featprism, featdprism, featrevol, featlf, featrf**

Syntax:
**featprism** loads the arguments for a prism with contiguous sides normal to the face.

**featdprism** loads the arguments for a prism which is created in a direction normal to the face and includes a draft angle.

**featrevol** loads the arguments for a prism with a circular evolution.

**featlf** loads the arguments for a linear rib or slot. This feature uses planar faces and a wire as a guideline.

**featrf** loads the arguments for a rib or slot with a curved surface. This feature uses a circular face and a wire as a guideline.

**featperform** loads the arguments to create the feature.

**featperformval** uses the defined arguments to create a feature with a limiting value.

All the features are created from a set of arguments which are defined when you initialize the feature context. Negative values can be used to create depressions.

**Examples:**

Let us create a feature prism with a draft angle and a normal direction :
# create a box with a wire contour on the upper face
box b 1 1 1
profil f 0 0 0 1 F 0.25 0.25 x 0.5 y 0.5 x -0.5
explode b f
# loads the feature arguments defining the draft angle
featdprism b f b_6 5 1 0
# create the feature
featperformval dprism r 1
==BRepFeat_MakeDPrism::Perform(Height)
BRepFeat_Form::GlobalPerform ()
Gluer
still Gluer
Gluer result

Let us create a feature prism with circular direction:

# create a box with a wire contour on the upper face
box b 1 1 1
profil f 0 0 0 1 F 0.25 0.25 x 0.5 y 0.5 x -0.5
explode b f
# loads the feature arguments defining a rotation axis
featrevol b f b_6 1 0 1 0 1 0 1 0
featperformval revol r 45
==BRepFeat_MakeRevol::Perform(Angle)
BRepFeat_Form::GlobalPerform ()
Gluer
still Gluer
Gluer result

Let us create a slot using the linear feature:

# create the base model using the multi viewer
mu4
profile p x 5 y 1 x -3 y -0.5 x -1.5 y 0.5 x 0.5 y 4
   x -1 y -5
prism pr p 0 0 1
# create the contour for the linear feature
vertex v1 -0.2 4 0.3
vertex v2 0.2 4 0.3
vertex v3 0.2 0.2 0.3
vertex v4 4 0.2 0.3
vertex v5 4 -0.2 0.3
draw e1 v1 v2
draw e2 v2 v3
draw e3 v3 v4
draw e4 v4 v5
draw w e1 e2 e3 e4

# define a plane
plane pl 0.2 0.2 0.3 0 0 1

# loads the linear feature arguments
featlf pr w pl 0 0 0.3 0 0 0 0 1
featperform lf result

Let us create a rib using the revolution feature:

# create the base model using the multi viewer
mu4
pcylinder c1 3 5

# create the contour for the revolution feature
profile w c 1 190 WW
trotate w 0 0 0 1 0 0 90
ttranslate w -3 0 1
trotate w -3 0 1.5 0 0 1 180
plane pl -3 0 1.5 0 1 0

# loads the revolution feature arguments
feattrf c1 w pl 0 0 0 0 0 1 0.3 0.3 1 1
featperform rf result

draft

Syntax:

draft result shape dirx diry dirz angle
shape/surf/length [-IN/-OUT] [Ri/Ro] [-Internal]
Computes a draft angle surface from a wire. The surface is determined by the draft direction, the inclination of the draft surface, a draft angle, and a limiting distance.

- The draft angle is measured in radians.
- The draft direction is determined by the argument -INTERNAL.
- The argument Ri/Ro determines whether the corner edges of the draft surfaces are angular or rounded.
- Arguments that can be used to define the surface distance are:
  - length, a defined distance
  - shape, until the surface contacts a shape
  - surface, until the surface contacts a surface.

**Note** that the original aim of adding a draft angle to a shape is to produce a shape which can be removed easily from a mould. The Examples below use larger angles than are used normally and the calculation results returned are not indicated.

**Example:**

```plaintext
# to create a simple profile
profile p F 0 0 x 2 y 4 tt 0 4 w

# creates a draft with rounded angles
draft res p 0 0 1 3 1 -Ro

# to create a profile with an internal angle
profile p F 0 0 x 2 y 4 tt 1 1.5 tt 0 4 w

# creates a draft with rounded external angles
draft res p 0 0 1 3 1 -Ro
```

**deform**

Syntax:

```plaintext
deform newname name CoeffX CoeffY CoeffZ
```

Modifies the shape using the x, y, and z coefficients. You can reduce or magnify the shape in the x,y, and z directions.

**Example:**
pcylinder c 20 20
deform a c 1 3 5
# the conversion to bspline is followed by the deformation

nurbsconvert

Syntax:

nurbsconvert result name [result name]

Changes the NURBS curve definition of a shape to a Bspline curve definition. This conversion is required for assymetric deformation and prepares the arguments for other commands such as deform. The conversion can be necessary when transferring shape data to other applications.

edgestofaces

edgestofaces - The command allows building planar faces from the planar edges randomly located in 3D space.

It has the following syntax:

edgestofaces r_faces edges [-a AngTol -s Shared(0/1)]

Options:

- -a AngTol - angular tolerance used for distinguishing the planar faces;
- -s Shared(0/1) - boolean flag which defines whether the input edges are already shared or have to be intersected.
Texture Mapping to a Shape

Texture mapping allows you to map textures on a shape. Textures are texture image files and several are predefined. You can control the number of occurrences of the texture on a face, the position of a texture and the scale factor of the texture.

vttexture

Syntax:

```
vttexture NameOfShape TextureFile
vttexture NameOfShape
vttexture NameOfShape ?
vttexture NameOfShape IdOfTexture
```

**TextureFile** identifies the file containing the texture you want. The same syntax without **TextureFile** disables texture mapping. The question-mark ? lists available textures. **IdOfTexture** allows applying predefined textures.

vtexscale

Syntax:

```
vtexscale NameOfShape ScaleU ScaleV
vtexscale NameOfShape ScaleUV
vtexscale NameOfShape
```

**ScaleU** and **Scale V** allow scaling the texture according to the U and V parameters individually, while **ScaleUV** applies the same scale to both parameters.

The syntax without **ScaleU**, **ScaleV** or **ScaleUV** disables texture scaling.

vtexorigin

Syntax:
vtexorigin NameOfShape UOrigin VOrigin
vtexorigin NameOfShape UVOrigin
vtexorigin NameOfShape

*UOrigin* and *VOrigin* allow placing the texture according to the U and V parameters individually, while *UVOrigin* applies the same position value to both parameters.

The syntax without *UOrigin*, *VOrigin* or *UVOrigin* disables origin positioning.

**vtexrepeat**

Syntax:

vtexrepeat NameOfShape URepeat VRepeat
vtexrepeat NameOfShape UVRepeat
vtexrepeat NameOfShape

*URepeat* and *VRepeat* allow repeating the texture along the U and V parameters individually, while *UVRepeat* applies the same number of repetitions for both parameters.

The same syntax without *URepeat*, *VRepeat* or *UVRepeat* disables texture repetition.

**vtexdefault**

Syntax:

vtexdefault NameOfShape

*Vtexdefault* sets or resets the texture mapping default parameters.

The defaults are:

- *URepeat* = *VRepeat* = 1 no repetition
- *UOrigin* = *VOrigin* = 1 origin set at (0,0)
- *UScale* = *VScale* = 1 texture covers 100% of the face
General Fuse Algorithm commands

This chapter describes existing commands of Open CASCADE Draw Test Harness that are used for debugging of General Fuse Algorithm (GFA). It is also applicable for all General Fuse based algorithms such as Boolean Operations Algorithm (BOA), Splitter Algorithm (SPA), Cells Builder Algorithm etc.

See Boolean operations user's guide for the description of these algorithms.
Definitions

The following terms and definitions are used in this document:

- **Objects** – list of shapes that are arguments of the algorithm.
- **Tools** – list of shapes that are arguments of the algorithm. Difference between Objects and Tools is defined by specific requirements of the operations (Boolean Operations, Splitting Operation).
- **DS** – internal data structure used by the algorithm (**BOPDS_DS** object).
- **PaveFiller** – intersection part of the algorithm (**BOPAlgo_PaveFiller** object).
- **Builder** – builder part of the algorithm (**BOPAlgo_Builder** object).
- **IDS Index** – the index of the vector *myLines*. 
General commands

- **bclearobjects** – clears the list of Objects;
- **bcleartools** – clears the list of Tools;
- **baddobjects S1 S2...Sn** – adds shapes S1, S2, ... Sn as Objects;
- **baddtools S1 S2...Sn** – adds shapes S1, S2, ... Sn as Tools;
- **bfillds** – performs the Intersection Part of the Algorithm;
- **bbuild r** – performs the Building Part of the Algorithm (General Fuse operation); r is the resulting shape;
- **bsplit r** – performs the Splitting operation; r is the resulting shape;
- **bbop r iOp** – performs the Boolean operation; r is the resulting shape; iOp - type of the operation (0 - COMMON; 1 - FUSE; 2 - CUT; 3 - CUT21; 4 - SECTION);
- **bcbuild rx** – performs initialization of the Cells Builder algorithm (see Usage of the Cells Builder algorithm for more details).
Commands for Intersection Part

All commands listed below are available when the Intersection Part of the algorithm is done (i.e. after the command bfillds).

**bopds**

Syntax:

```
bopds -v [e, f]
```

Displays:

- all BRep shapes of arguments that are in the DS [default];
- `-v` : only vertices of arguments that are in the DS;
- `-e` : only edges of arguments that are in the DS;
- `-f` : only faces of arguments that are in the DS.

**bopdsdump**

Prints contents of the DS.

Example:

```
Draw[28]> bopdsdump
*** DS ***
Ranges:2 number of ranges
 range: 0 33 indices for range 1
 range: 34 67 indices for range 2
Shapes:68 total number of source shapes
 0 : SOLID { 1 }
 1 : SHELL { 2 12 22 26 30 32 }
 2 : FACE { 4 5 6 7 8 9 10 11 }
 3 : WIRE { 4 7 9 11 }
 4 : EDGE { 5 6 }
 5 : VERTEX { }
 6 : VERTEX { }
```
has the following meaning:

- 0 – index in the DS;
- SOLID – type of the shape;
- \{1\} – a DS index of the successors.

**bopindex**

Syntax:

```
bopindex S
```

Prints DS index of shape S.

**bopiterator**

Syntax:

```
bopiterator [t1 t2]
```

Prints pairs of DS indices of source shapes that are intersected in terms of bounding boxes.

\[t1 \ t2\] are types of the shapes:

- 7 – vertex;
- 6 – edge;
- 4 – face.

Example:

```
Draw[104]> bopiterator 6 4
EF: ( z58 z12 )
EF: ( z17 z56 )
EF: ( z19 z64 )
```
• *bopiterator 6 4* prints pairs of indices for types: edge/face;
  • *z58 z12* – DS indices of intersecting edge and face.

### bopinterf

**Syntax:**

```
bopinterf t
```

Prints contents of *myInterfTB* for the type of interference *t*:

- *t=0* : vertex/vertex;
- *t=1* : vertex/edge;
- *t=2* : edge/edge;
- *t=3* : vertex/face;
- *t=4* : edge/face.

**Example:**

```
Draw[108]> bopinterf 4
EF: (58, 12, 68), (17, 56, 69), (19, 64, 70), (45, 26, 71), (29, 36, 72), (38, 32, 73), 6 EF found.
```

Here, record *(58, 12, 68)* means:

- *58* – a DS index of the edge;
- *12* – a DS index of the face;
- *68* – a DS index of the new vertex.

### bopsp

Displays split edges.

**Example:**

```
Draw[33]> bopsp
```
edge 58 : z58_74 z58_75  
edge 17 : z17_76 z17_77  
edge 19 : z19_78 z19_79  
edge 45 : z45_80 z45_81  
edge 29 : z29_82 z29_83  
edge 38 : z38_84 z38_85

- *edge 58* – 58 is a DS index of the original edge.  
- *z58_74 z58_75* – split edges, where 74, 75 are DS indices of the split edges.

**bopcb**

**Syntax:**

`bopcb [nE]`

Prints Common Blocks for:

- all source edges (by default);  
- the source edge with the specified index *nE*.

**Example:**

```
Draw[43]> bopcb 17  
-- CB:  
PB:{  E:71 orE:17  Pave1: {  68 3.000 }  Pave2: {  18 10.000 } }  
Faces: 36
```

This command dumps common blocks for the source edge with index 17.

- *PB* – information about the Pave Block;  
  - *71* – a DS index of the split edge  
  - *17* – a DS index of the original edge  
- *Pave1: { 68 3.000 }* – information about the Pave:  
  - *68* – a DS index of the vertex of the pave  
  - *3.000* – a parameter of vertex 68 on edge 17  
- *Faces: 36* – 36 is a DS index of the face the common block belongs to.
bopfin

Syntax:

```
bopfin nF
```

Prints Face Info about IN-parts for the face with DS index nF.

Example:

```
Draw[47]> bopfin 36
pave blocks In:
PB:{ E:71 orE:17 Pave1: { 68 3.000 } Pave2: { 18 10.000 } }
PB:{ E:75 orE:19 Pave1: { 69 3.000 } Pave2: { 18 10.000 } }
vrts In: 18
```

- PB:{E:71 orE:17 Pave1: { 68 3.000 } Pave2: { 18 10.000 } } – information about the Pave Block;
- vrts In ... 18 – a DS index of the vertex IN the face.

bopfon

Syntax:

```
bopfon nF
```

Print Face Info about ON-parts for the face with DS index nF.

Example:

```
Draw[58]> bopfon 36
pave blocks On:
PB:{ E:72 orE:38 Pave1: { 69 0.000 } Pave2: { 68 10.000 } }
PB:{ E:76 orE:45 Pave1: { 69 0.000 } Pave2: { 71 10.000 } }
```
• PB: { E: 78 or E: 43 Pave1: { 71 0.000 } Pave2: { 70 10.000 } } – information about the Pave Block;
• vrts On: ... 68 69 70 71 – DS indices of the vertices ON the face.

**bopwho**

Syntax:

```
bopwho nS
```

Prints the information about the shape with DS index nF.

Example:

```
Draw[116]> bopwho 5
rank: 0
```

• rank: 0 – means that shape 5 results from the Argument with index 0.

Example:

```
Draw[118]> bopwho 68
the shape is new
EF: (58, 12),
FF curves: (12, 56),
FF curves: (12, 64),
```

This means that shape 68 is a result of the following interferences:

• EF: (58, 12) – edge 58 / face 12
• FF curves: (12, 56) – edge from the intersection curve between
faces 12 and 56
• *FF curves: (12, 64)* – edge from the intersection curve between faces 12 and 64

**bopnews**

Syntax:

```
bopnews -v [-e]
```

• `-v` – displays all new vertices produced during the operation;
• `-e` – displays all new edges produced during the operation.
Commands for the Building Part

The commands listed below are available when the Building Part of the algorithm is done (i.e. after the command *bbuild*).

**bopim**

Syntax:

```
 bopim S
```

Shows the compound of shapes that are images of shape S from the argument.
Data Exchange commands

This chapter presents some general information about Data Exchange (DE) operations.

DE commands are intended for translation files of various formats (IGES,STEP) into OCCT shapes with their attributes (colors, layers etc.)

This files include a number of entities. Each entity has its own number in the file which we call label and denote as # for a STEP file and D for an IGES file. Each file has entities called roots (one or more). A full description of such entities is contained in the Users’ Guides

- for **STEP format** and
- for **IGES format**.

Each Draw session has an interface model, which is a structure for keeping various information.

The first step of translation is loading information from a file into a model. The second step is creation of an OpenCASCADE shape from this model.

Each entity from a file has its own number in the model (num). During the translation a map of correspondences between labels(from file) and numbers (from model) is created.

The model and the map are used for working with most of DE commands.
IGES commands

igesread

Syntax:

```
igesread <file_name> <result_shape_name>
    [<selection>]
```

Reads an IGES file to an OCCT shape. This command will interactively ask the user to select a set of entities to be converted.

<table>
<thead>
<tr>
<th>N</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>End</td>
<td>finish conversion and exit igesbrep</td>
</tr>
<tr>
<td>1</td>
<td>Visible roots</td>
<td>convert only visible roots</td>
</tr>
<tr>
<td>2</td>
<td>All roots</td>
<td>convert all roots</td>
</tr>
<tr>
<td>3</td>
<td>One entity</td>
<td>convert entity with number provided by the user</td>
</tr>
<tr>
<td>4</td>
<td>Selection</td>
<td>convert only entities contained in selection</td>
</tr>
</tbody>
</table>

After the selected set of entities is loaded the user will be asked how loaded entities should be converted into OCCT shapes (e.g., one shape per root or one shape for all the entities). It is also possible to save loaded shapes in files, and to cancel loading.

The second parameter of this command defines the name of the loaded shape. If several shapes are created, they will get indexed names. For instance, if the last parameter was s, they will be s_1, ... s_N.

<selection> specifies the scope of selected entities in the model, by default it is xst-transferrable-roots. If we use symbol * as <selection> all roots will be translated.

See also the detailed description of Selecting IGES entities.

Example:

```
# translation all roots from file
```
tgread /disk01/files/model.igs a *

**tplosttrim**

Syntax:

```
tplosttrim [<IGES_type>]
```

Sometimes the trimming contours of IGES faces (i.e., entity 141 for 143, 142 for 144) can be lost during translation due to fails. This command gives us a number of lost trims and the number of corresponding IGES entities. It outputs the rank and numbers of faces that lost their trims and their numbers for each type (143, 144, 510) and their total number. If a face lost several of its trims it is output only once. Optional parameter `<IGES_type>` can be `0TrimmedSurface`, `BoundedSurface` or `Face` to specify the only type of IGES faces.

**Example:**

```
tplosttrim TrimmedSurface
```

**brepigges**

Syntax:

```
brepigges <shape_name> <filename.igs>
```

Writes an OCCT shape to an IGES file.

**Example:**

```
# write shape with name aa to IGES file
brepigges aa /disk1/tmp/aaa.igs
== unit (write) : MM
== mode write : Faces
== To modify : command param
== 1 Shapes written, giving 345 Entities
== Now, to write a file, command : writeall filename
== Output on file : /disk1/tmp/aaa.igs
```
Write OK
STEP commands

These commands are used during the translation of STEP models.

stepread

Syntax:

```plaintext
stepread file_name result_shape_name [selection]
```

Read a STEP file to an OCCT shape. This command will interactively ask the user to select a set of entities to be converted:

<table>
<thead>
<tr>
<th>N</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>End</td>
<td>Finish transfer and exit stepread</td>
</tr>
<tr>
<td>1</td>
<td>root with rank 1</td>
<td>Transfer first root</td>
</tr>
<tr>
<td>2</td>
<td>root by its rank</td>
<td>Transfer root specified by its rank</td>
</tr>
<tr>
<td>3</td>
<td>One entity</td>
<td>Transfer entity with a number provided by the user</td>
</tr>
<tr>
<td>4</td>
<td>Selection</td>
<td>Transfer only entities contained in selection</td>
</tr>
</tbody>
</table>

After the selected set of entities is loaded the user will be asked how loaded entities should be converted into OCCT shapes. The second parameter of this command defines the name of the loaded shape. If several shapes are created, they will get indexed names. For instance, if the last parameter was `s`, they will be `s_1, ... s_N`. `<selection>` specifies the scope of selected entities in the model. If we use symbol `*` as `<selection>` all roots will be translated.

See also the detailed description of Selecting STEP entities.

Example:

```plaintext
# translation all roots from file
stepread /disk01/files/model.stp a  *
```

stepwrite
Syntax:

```plaintext
stepwrite mode shape_name file_name
```

Writes an OCCT shape to a STEP file.

The following modes are available:

- `a` – as is – the mode is selected automatically depending on the type & geometry of the shape;
- `m` – `manifold_solid_brep` or `brep_with_voids`
- `f` – `faceted_brep`
- `w` – `geometric_curve_set`
- `s` – `shell_based_surface_model`

For further information see [Writing a STEP file](#).

**Example:**

Let us write shape a to a STEP file in mode 0.

```plaintext
stepwrite 0 a /disk1/tmp/aaa.igs
```
General commands

These are auxiliary commands used for the analysis of result of translation of IGES and STEP files.

**count**

**Syntax:**

```
count <counter> [ <selection> ]
```

Calculates statistics on the entities in the model and outputs a count of entities.

The optional selection argument, if specified, defines a subset of entities, which are to be taken into account. The first argument should be one of the currently defined counters.

<table>
<thead>
<tr>
<th>Counter</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xst-types</td>
<td>Calculates how many entities of each OCCT type exist</td>
</tr>
<tr>
<td>step214-types</td>
<td>Calculates how many entities of each STEP type exist</td>
</tr>
</tbody>
</table>

**Example:**

```
count xst-types
```

**data**

**Syntax:**

```
data <symbol>
```

Obtains general statistics on the loaded data. The information printed by this command depends on the symbol specified.

**Example:**
# print full information about warnings and fails data c

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>Prints the information contained in the header of the file</td>
</tr>
<tr>
<td>c or f</td>
<td>Prints messages generated during the loading of the STEP file (when the procedure of the integrity of the loaded data check is performed) and the resulting statistics (f works only with fail messages while c with both fail and warning messages)</td>
</tr>
<tr>
<td>t</td>
<td>The same as c or f, with a list of failed or warned entities</td>
</tr>
<tr>
<td>m or l</td>
<td>The same as t but also prints a status for each entity</td>
</tr>
<tr>
<td>e</td>
<td>Lists all entities of the model with their numbers, types, validity status etc.</td>
</tr>
<tr>
<td>R</td>
<td>The same as e but lists only root entities</td>
</tr>
</tbody>
</table>

**elabel**

Syntax:

`elabel <num>`

Entities in the IGES and STEP files are numbered in the succeeding order. An entity can be identified either by its number or by its label. Label is the letter ‘#’ (for STEP, for IGES use ‘D’) followed by the rank. This command gives us a label for an entity with a known number.

**Example:**

`elabel 84`

**entity**

Syntax:

`entity <%(D)>_or_<num> <level_of_information>`

The content of an IGES or STEP entity can be obtained by using this
Entity can be determined by its number or label. `<level_of_information>` has range [0-6]. You can get more information about this level using this command without parameters.

**Example:**

```plaintext
# full information for STEP entity with label 84
entity #84 6
```

**enum**

Syntax:

```plaintext
enum <#(D)>
```

Prints a number for the entity with a given label.

**Example:**

```plaintext
# give a number for IGES entity with label 21
enum D21
```

**estatus**

Syntax:

```plaintext
estatus <#(D)>_or_<num>
```

The list of entities referenced by a given entity and the list of entities referencing to it can be obtained by this command.

**Example:**

```plaintext
estatus #315
```

**fromshape**

Syntax:
fromshape <shape_name>

Gives the number of an IGES or STEP entity corresponding to an OCCT shape. If no corresponding entity can be found and if OCCT shape is a compound the command explodes it to subshapes and try to find corresponding entities for them.

Example:

fromshape a_1_23

givecount

Syntax:

givecount <selection_name> [<selection_name>]

Prints a number of loaded entities defined by the selection argument. Possible values of <selection_name> you can find in the “IGES FORMAT Users’s Guide”.

Example:

givecount xst-model-roots

givelist

Syntax:

givelist <selection_name>

Prints a list of a subset of loaded entities defined by the selection argument:

<table>
<thead>
<tr>
<th>Selection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xst-model-all</td>
<td>all entities of the model</td>
</tr>
<tr>
<td>xst-model-roots</td>
<td>all roots</td>
</tr>
<tr>
<td>xst-pointed</td>
<td>(Interactively) pointed entities (not used in</td>
</tr>
<tr>
<td>DRAW)</td>
<td>all transferable (recognized) entities</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>xst-transferrable-all</td>
<td>all transferable (recognized) entities</td>
</tr>
<tr>
<td>xst-transferrable-roots</td>
<td>Transferable roots</td>
</tr>
</tbody>
</table>

**Example:**

```bash
# give a list of all entities of the model
givelist xst-model-all
```

**listcount**

Syntax: `listcount <counter> [<selection> ...]`

Prints a list of entities per each type matching the criteria defined by arguments. Optional `<selection>` argument, if specified, defines a subset of entities, which are to be taken into account. Argument `<counter>` should be one of the currently defined counters:

<table>
<thead>
<tr>
<th>Counter</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xst-types</td>
<td>Calculates how many entities of each OCCT type exist</td>
</tr>
<tr>
<td>iges-types</td>
<td>Calculates how many entities of each IGES type and form exist</td>
</tr>
<tr>
<td>iges-levels</td>
<td>Calculates how many entities lie in different IGES levels</td>
</tr>
</tbody>
</table>

**Example:**

```bash
listcount xst-types
```

**listitems**

Syntax:

```bash
listitems
```

This command prints a list of objects (counters, selections etc.) defined in the current session.
listtypes

Syntax:

listtypes [<selection_name> ...]

Gives a list of entity types which were encountered in the last loaded file (with a number of entities of each type). The list can be shown not for all entities but for a subset of them. This subset is defined by an optional selection argument.

newmodel

Syntax:

newmodel

Clears the current model.

param

Syntax:

param [<parameter>] [<value>]

This command is used to manage translation parameters. Command without arguments gives a full list of parameters with current values. Command with <parameter> (without ) gives us the current value of this parameter and all possible values for it. Command with

sets this new value to <parameter>.

Example:

Let us get the information about possible schemes for writing STEP file:

param write.step.schema
sumcount

Syntax:

```
sumcount <counter> [ <selection> ... ]
```

Prints only a number of entities per each type matching the criteria defined by arguments.

Example:

```
sumcount xst-types
```

tpclear

Syntax:

```
tpclear
```

Clears the map of correspondences between IGES or STEP entities and OCCT shapes.

tpdraw

Syntax:

```
tpdraw <#(D)>_or_<num>
```

Example:

```
tpdraw 57
```

tpent

Syntax:

```
tpent <#(D)>_or_<num>
```

Get information about the result of translation of the given IGES or STEP
Example:

```
tpent \#23
```

**tpstat**

Syntax:

```
.tpstat [*|?]<symbol> [<selection>]
```

Provides all statistics on the last transfer, including a list of transferred entities with mapping from IGES or STEP to OCCT types, as well as fail and warning messages. The parameter `<symbol>` defines what information will be printed:

- `g` – General statistics (a list of results and messages)
- `c` – Count of all warning and fail messages
- `C` – List of all warning and fail messages
- `f` – Count of all fail messages
- `F` – List of all fail messages
- `n` – List of all transferred roots
- `s` – The same, with types of source entity and the type of result
- `b` – The same, with messages
- `t` – Count of roots for geometrical types
- `r` – Count of roots for topological types
- `l` – The same, with the type of the source entity

The sign `*` before parameters `n, s, b, t, r` makes it work on all entities (not only on roots).

The sign `?` before `n, s, b, t` limits the scope of information to invalid entities.

Optional argument `<selection>` can limit the action of the command to the selection, not to all entities.

To get help, run this command without arguments.

Example:
xload

Syntax:

xload <file_name>

This command loads an IGES or STEP file into memory (i.e. to fill the model with data from the file) without creation of an OCCT shape.

Example:

xload /disk1/tmp/aaa.stp
Overview of XDE commands

These commands are used for translation of IGES and STEP files into an XCAF document (special document is inherited from CAF document and is intended for Extended Data Exchange (XDE) ) and working with it. XDE translation allows reading and writing of shapes with additional attributes – colors, layers etc. All commands can be divided into the following groups:

- XDE translation commands
- XDE general commands
- XDE shape's commands
- XDE color's commands
- XDE layer's commands
- XDE property's commands

Reminding: All operations of translation are performed with parameters managed by command `param`.

ReadIges

Syntax:

ReadIges document file_name

Reads information from an IGES file to an XCAF document.

Example:

ReadIges D /disk1/tmp/aaa.igs
==> Document saved with name D

ReadStep

Syntax:

ReadStep <document> <file_name>
Reads information from a STEP file to an XCAF document.

**Example:**

```
ReadStep D /disk1/tmp/aaa.stp
== Document saved with name D
```

**WriteIges**

_Syntax:

```
WriteIges <document> <file_name>
```

**Example:**

```
WriteIges D /disk1/tmp/aaa.igs
```

**WriteStep**

_Syntax:

```
WriteStep <document> <file_name>
```

Writes information from an XCAF document to a STEP file.

**Example:**

```
WriteStep D /disk1/tmp/aaa.stp
```

**XFileCur**

_Syntax:

```
XFileCur
```

Returns the name of file which is set as the current one in the Draw session.

**Example:**
XFileCur
== *as1-ct-203.stp*

**XFileList**

Syntax:

**XFileList**

Returns a list all files that were transferred by the last transfer. This command is meant (assigned) for the assemble step file.

**Example:**

**XFileList**

```plaintext
==> *as1-ct-Bolt.stp*
==> *as1-ct-L-Bracket.stp*
==> *as1-ct-LBA.stp*
==> *as1-ct-NBA.stp*
==> ...
```

**XFileSet**

Syntax:

**XFileSet** <filename>

Sets the current file taking it from the components list of the assemble file.

**Example:**

**XFileSet as1-ct-NBA.stp**

**XFromShape**

Syntax:

**XFromShape** <shape>
This command is similar to the command `fromshape`, but gives additional information about the file name. It is useful if a shape was translated from several files.

Example:

```
XFromShape a
==> Shape a: imported from entity 217:#26 in file as1-ct-Nut.stp
```
XDE general commands

XNewDoc

Syntax:

XNewDoc <document>

Creates a new XCAF document.

Example:

XNewDoc D

XShow

Syntax:

XShow <document> [ <label1> ... ]

Shows a shape from a given label in the 3D viewer. If the label is not given – shows all shapes from the document.

Example:

# show shape from label 0:1:1:4 from document D
XShow D 0:1:1:4

XStat

Syntax:

XStat <document>

Prints common information from an XCAF document.

Example:
XStat D

===>Statistics of shapes in the document:
===>level N 0 : 9
===>level N 1 : 18
===>level N 2 : 5
===>Total number of labels for shapes in the document = 32
===>Number of labels with name = 27
===>Number of labels with color link = 3
===>Number of labels with layer link = 0
===>Statistics of Props in the document:
===>Number of Centroid Props = 5
===>Number of Volume Props = 5
===>Number of Area Props = 5
===>Number of colors = 4
===>BLUE1 RED YELLOW BLUE2
===>Number of layers = 0

XWdump

Syntax:

XWdump <document> <filename>

Saves the contents of the viewer window as an image (XWD, png or BMP file). <filename> must have a corresponding extension.

Example:

XWdump D /disk1/tmp/image.png

Xdump

Syntax:

Xdump <document> [int deep {0|1}]

Prints information about the tree structure of the document. If parameter
1 is given, then the tree is printed with a link to shapes.

**Example:**

```
Xdump D 1
==>	ASSEMBLY 0:1:1:1 L-BRACKET(0xe8180448)
==>	ASSEMBLY 0:1:1:2 NUT(0xe82151e8)
==>	ASSEMBLY 0:1:1:3 BOLT(0xe829b000)
==>	ASSEMBLY 0:1:1:4 PLATE(0xe8387780)
==>	ASSEMBLY 0:1:1:5 ROD(0xe8475418)
==>	ASSEMBLY 0:1:1:6 AS1(0xe8476968)
==>		ASSEMBLY 0:1:1:7 L-BRACKET-
		ASSEMBLY(0xe8476230)
==>		ASSEMBLY 0:1:1:1 L-BRACKET(0xe8180448)
==>		ASSEMBLY 0:1:1:8 NUT-BOLT-
		ASSEMBLY(0xe8475ec0)
==>			ASSEMBLY 0:1:1:2 NUT(0xe82151e8)
==>			ASSEMBLY 0:1:1:3 BOLT(0xe829b000)
etc.
```
XDE shape commands

XAddComponent

Syntax:

```
XAddComponent <document> <label> <shape>
```

Adds a component shape to assembly.

Example:

Let us add shape b as component shape to assembly shape from label 0:1:1:1

```
XAddComponent D 0:1:1:1 b
```

XAddShape

Syntax:

```
XAddShape <document> <shape> [makeassembly=1]
```

Adds a shape (or an assembly) to a document. If this shape already exists in the document, then prints the label which points to it. By default, a new shape is added as an assembly (i.e. last parameter 1), otherwise it is necessary to pass 0 as the last parameter.

Example:

```
# add shape b to document D
XAddShape D b 0
== 0:1:1:10
# if pointed shape is compound and last parameter in
# XAddShape command is used by default (1), then for
# each subshapes new label is created
```

XFindComponent
**Syntax:**

```
XFindComponent <document> <shape>
```

Prints a sequence of labels of the assembly path.

**Example:**

```
XFindComponent D b
```

**XFindShape**

**Syntax:**

```
XFindShape <document> <shape>
```

Finds and prints a label with an indicated top-level shape.

**Example:**

```
XFindShape D a
```

**XGetFreeShapes**

**Syntax:**

```
XGetFreeShapes <document> [shape_prefix]
```

Print labels or create DRAW shapes for all free shapes in the document. If `shape_prefix` is absent – prints labels, else – creates DRAW shapes with names `shape_prefix_num` (i.e. for example: there are 3 free shapes and `shape_prefix = a` therefore shapes will be created with names `a_1`, `a_2` and `a_3`).

**Note:** a free shape is a shape to which no other shape refers to.

**Example:**

```
XGetFreeShapes D
== 0:1:1:6 0:1:1:10 0:1:1:12 0:1:1:13
```
XGetFreeShapes D sh  
== sh_1 sh_2 sh_3 sh_4

XGetOneShape

Syntax:

XGetOneShape <shape> <document>

Creates one DRAW shape for all free shapes from a document.

Example:

XGetOneShape a D

XGetReferredShape

Syntax:

XGetReferredShape <document> <label>

Prints a label that contains a top-level shape that corresponds to a shape at a given label.

Example:

XGetReferredShape D 0:1:1:1:1

XGetShape

Syntax:

XGetShape <result> <document> <label>

Puts a shape from the indicated label in document to result.

Example:
XGetShape b D 0:1:1:3

XGetTopLevelShapes
Syntax:
XGetTopLevelShapes <document>
Prints labels that contain top-level shapes.
Example:
XGetTopLevelShapes D

XLabelInfo
Syntax:
XLabelInfo <document> <label>
Prints information about a shape, stored at an indicated label.
Example:
XLabelInfo D 0:1:1:6
==> There are TopLevel shapes. There is an Assembly. This Shape is not used.

XNewShape
Syntax:
XNewShape <document>
Creates a new empty top-level shape.
Example:
XNewShape D

**XRemoveComponent**

Syntax:

XRemoveComponent <document> <label>

Removes a component from the components label.

Example:
XRemoveComponent D 0:1:1:1:1

**XRemoveShape**

Syntax:

XRemoveShape <document> <label>

Removes a shape from a document (by it's label).

Example:
XRemoveShape D 0:1:1:2

**XSetShape**

Syntax:

XSetShape <document> <label> <shape>

Sets a shape at the indicated label.

Example:
XSetShape D 0:1:1:3 b
XUpdateAssemblies

Syntax:

```
XUpdateAssemblies <document>
```

Updates all assembly compounds in the XDE document.

Example:

```
XUpdateAssemblies D
```
XDE color commands

XAddColor

Syntax:

```
XAddColor <document> <R> <G> <B>
```

Adds color in document to the color table. Parameters R,G,B are real.

Example:

```
XAddColor D 0.5 0.25 0.25
```

XFindColor

Syntax:

```
XFindColor <document> <R> <G> <B>
```

Finds a label where the indicated color is situated.

Example:

```
XFindColor D 0.25 0.25 0.5
==> 0:1:2:2
```

XGetAllColors

Syntax:

```
XGetAllColors <document>
```

Prints all colors that are defined in the document.

Example:

```
XGetAllColors D
```
XGetColor

Syntax:

XGetColor <document> <label>

Returns a color defined at the indicated label from the color table.

Example:

XGetColor D 0:1:2:3
== BLUE1

XGetObjVisibility

Syntax:

XGetObjVisibility <document> {<label>|<shape>}

Returns the visibility of a shape.

Example:

XGetObjVisibility D 0:1:1:4

XGetShapeColor

Syntax:

XGetShapeColor <document> <label> <colortype(s|c)>

Returns the color defined by label. If colortype='s' – returns surface color, else – returns curve color.

Example:

XGetShapeColor D 0:1:1:4 c
**XRemoveColor**

Syntax:

```plaintext
XRemoveColor <document> <label>
```

Removes a color from the color table in a document.

**Example:**

```plaintext
XRemoveColor D 0:1:2:1
```

**XSetColor**

Syntax:

```plaintext
XSetColor <document> {<label>|<shape>} <R> <G> <B>
```

Sets an RGB color to a shape given by label.

**Example:**

```plaintext
XSetColor D 0:1:1:4 0.5 0.5 0.
```

**XSetObjVisibility**

Syntax:

```plaintext
XSetObjVisibility <document> {<label>|<shape>} {0|1}
```

Sets the visibility of a shape.

**Example:**

```plaintext
# set shape from label 0:1:1:4 as invisible
XSetObjVisibility D 0:1:1:4 0
```

**XUnsetColor**
Syntax:

```
XUnsetColor <document> {<label>|<shape>} <colortype>
```

Unset a color given type (‘s’ or ‘c’) for the indicated shape.

**Example:**

```
XUnsetColor D 0:1:1:4 s
```
XDE layer commands

XAddLayer

Syntax:

```
XAddLayer <document> <layer>
```

Adds a new layer in an XCAF document.

Example:

```
XAddLayer D layer2
```

XFindLayer

Syntax:

```
XFindLayer <document> <layer>
```

Prints a label where a layer is situated.

Example:

```
XFindLayer D Bolt
== 0:1:3:2
```

XGetAllLayers

Syntax:

```
XGetAllLayers <document>
```

Prints all layers in an XCAF document.

Example:

```
XGetAllLayers D
```
XGetLayers

Syntax:

```plaintext
XGetLayers <document> {<shape>|<label>}
```

Returns names of layers, which are pointed to by links of an indicated shape.

**Example:**

```plaintext
XGetLayers D 0:1:1:3
== *bolt* *123*
```

XGetOneLayer

Syntax:

```plaintext
XGetOneLayer <document> <label>
```

Prints the name of a layer at a given label.

**Example:**

```plaintext
XGetOneLayer D 0:1:3:2
```

XIsVisible

Syntax:

```plaintext
XIsVisible <document> {<label>|<layer>}
```

Returns 1 if the indicated layer is visible, else returns 0.

**Example:**

```plaintext
XIsVisible D 0:1:3:1
```
**XRemoveAllLayers**

Syntax:

```python
XRemoveAllLayers <document>
```

Removes all layers from an XCAF document.

**Example:**

```plaintext
XRemoveAllLayers D
```

**XRemoveLayer**

Syntax:

```python
XRemoveLayer <document> {<label>|<layer>}
```

Removes the indicated layer from an XCAF document.

**Example:**

```plaintext
XRemoveLayer D layer2
```

**XSetLayer**

Syntax:

```python
XSetLayer <document> {<shape>|<label>}
layer [shape_in_one_layer {0|1}]
```

Sets a reference between a shape and a layer (adds a layer if it is necessary). Parameter `<shape_in_one_layer>` shows whether a shape could be in a number of layers or only in one (0 by default).

**Example:**

```plaintext
XSetLayer D 0:1:1:2 layer2
```
**XSetVisibility**

Syntax:

```
XSetVisibility <document> {<label>|<layer>}
  <isvisible {0|1}>
```

Sets the visibility of a layer.

**Example:**

```
# set layer at label 0:1:3:2 as invisible
XSetVisibility D 0:1:3:2 0
```

**XUnSetAllLayers**

Syntax:

```
XUnSetAllLayers <document> {<label>|<shape>}
```

Unsets a shape from all layers.

**Example:**

```
XUnSetAllLayers D 0:1:1:2
```

**XUnSetLayer**

Syntax:

```
XUnSetLayer <document> {<label>|<shape>} <layer>
```

Unsets a shape from the indicated layer.

**Example:**

```
XUnSetLayer D 0:1:1:2 layer1
```
XDE property commands

XCheckProps

Syntax:

```
XCheckProps <document> [ {0|deflection} [<shape>|<label>] ]
```

Gets properties for a given shape (volume, area and centroid) and compares them with the results after internal calculations. If the second parameter is 0, the standard OCCT tool is used for the computation of properties. If the second parameter is not 0, it is processed as a deflection. If the deflection is positive the computation is done by triangulations, if it is negative – meshing is forced.

Example:

```
# check properties for shapes at label 0:1:1:1 from document using standard Open CASCADE Technology tools
XCheckProps D 0 0:1:1:1
== Label 0:1:1:1 ;L-BRACKET*
== Area defect: -0.0 ( 0%)
== Volume defect: 0.0 ( 0%)
== CG defect: dX=-0.000, dY=0.000, dZ=0.000
```

XGetArea

Syntax:

```
XGetArea <document> [<shape>|<label>]
```

Returns the area of a given shape.

Example:

```
XGetArea D 0:1:1:1
```
### XGetCentroid

**Syntax:**

```
XGetCentroid <document> {<shape>|<label>}
```

Returns the center of gravity coordinates of a given shape.

**Example:**

```
XGetCentroid D 0:1:1:1
```

### XGetVolume

**Syntax:**

```
XGetVolume <document> {<shape>|<label>}
```

Returns the volume of a given shape.

**Example:**

```
XGetVolume D 0:1:1:1
```

### XSetArea

**Syntax:**

```
XSetArea <document> {<shape>|<label>} <area>
```

Sets new area to attribute list ??? given shape.

**Example:**

```
XSetArea D 0:1:1:1 2233.99
```

### XSetCentroid
Syntax:

XSetCentroid  <document>  {<shape>|<label>}  <x>  <y>  <z>

Sets new center of gravity to the attribute list given shape.

Example:

XSetCentroid  D 0:1:1:1 0. 0. 100.

XSetMaterial

Syntax:

XSetMaterial  <document>  {<shape>|<label>}  <name>
                <density(g/cu sm)>

Adds a new label with material into the material table in a document, and adds a link to this material to the attribute list of a given shape or a given label. The last parameter sets the density of a pointed material.

Example:

XSetMaterial  D 0:1:1:1 Titanium 8899.77

XSetVolume

Syntax:

XSetVolume  <document>  {<shape>|<label>}  <volume>

Sets new volume to the attribute list given shape.

Example:

XSetVolume  D 0:1:1:1 444555.33

XShapeMassProps

Syntax:
XShapeMassProps <document> [ <deflection> [{<shape>| <label>}] ]

Computes and returns real mass and real center of gravity for a given shape or for all shapes in a document. The second parameter is used for calculation of the volume and CG(center of gravity). If it is 0, then the standard CASCADE tool (geometry) is used for computation, otherwise – by triangulations with a given deflection.

Example:

XShapeMassProps D
== Shape from label : 0:1:1:1
== Mass = 193.71681469282299
== CenterOfGravity X = 14.594564763807696, Y = 20.20271885211281, Z = 49.999999385313245
== Shape from label : 0:1:1:2 not have a mass etc.

XShapeVolume

Syntax:

XShapeVolume <shape> <deflection>

Calculates the real volume of a pointed shape with a given deflection.

Example:

XShapeVolume a 0
Shape Healing commands

General commands

bsplres

Syntax:

```
bsplres <result> <shape> <tol3d> <tol2d> <reqdegree>
   <reqnbsegments> <continuity3d> <continuity2d>
   <PriorDeg> <RationalConvert>
```

Performs approximations of a given shape (BSpline curves and surfaces or other surfaces) to BSpline with given required parameters. The specified continuity can be reduced if the approximation with a specified continuity was not done successfully. Results are put into the shape, which is given as a parameter result. For a more detailed description see the ShapeHealing User’s Guide (operator: **BSplineRestriction**).

checkfclass2d

Syntax:

```
checkfclass2d <face> <ucoord> <vcoord>
```

Shows where a point which is given by coordinates is located in relation to a given face – outbound, inside or at the bounds.

**Example:**

```
checkfclass2d f 10.5 1.1
== Point is OUT
```

checkoverlapedges

Syntax:
checkoverlapedges <edge1> <edge2> [<toler> <domaindist>]

Checks the overlapping of two given edges. If the distance between two edges is less than the given value of tolerance then edges are overlapped. Parameter <domaindist> sets length of part of edges on which edges are overlapped.

Example:

checkoverlapedges e1 e2

comtol

Syntax:

comptol <shape> [nbpoints] [prefix]

Compares the real value of tolerance on curves with the value calculated by standard (using 23 points). The maximal value of deviation of 3d curve from pcurve at given simple points is taken as a real value (371 is by default). Command returns the maximal, minimal and average value of tolerance for all edges and difference between real values and set values. Edges with the maximal value of tolerance and relation will be saved if the ‘prefix’ parameter is given.

Example:

comptol h 871 t

=> Edges tolerance computed by 871 points:
=> MAX=8.0001130696523449e-008
        AVG=6.349346868091096e-009  MIN=0
=> Relation real tolerance / tolerance set in edge
=> MAX=0.80001130696523448  AVG=0.06349345591805905
        MIN=0
=> Edge with max tolerance saved to t_edge_tol
=> Concerned faces saved to shapes t_1, t_2
**convtorevol**

Syntax:

```
convtorevol <result> <shape>
```

Converts all elementary surfaces of a given shape into surfaces of revolution. Results are put into the shape, which is given as the `<result>` parameter.

**Example:**

```
convtorevol r a
```

**directfaces**

Syntax:

```
directfaces <result> <shape>
```

Converts indirect surfaces and returns the results into the shape, which is given as the result parameter.

**Example:**

```
directfaces r a
```

**expshape**

Syntax:

```
expshape <shape> <maxdegree> <maxseg>
```

Gives statistics for a given shape. This test command is working with Bezier and BSpline entities.

**Example:**

```
expshape a 10 10
==>
Number of Rational Bspline curves 128
```
fixsmall

Syntax:

fixsmall <result> <shape> [<toler>=1.]

Fixes small edges in given shape by merging adjacent edges with a given tolerance. Results are put into the shape, which is given as the result parameter.

Example:

fixsmall r a 0.1

fixsmalledges

Syntax:

fixsmalledges <result> <shape> [<toler> <mode> <maxangle>]

Searches at least one small edge at a given shape. If such edges have been found, then small edges are merged with a given tolerance. If parameter <mode> is equal to Standard_True (can be given any values, except 2), then small edges, which can not be merged, are removed, otherwise they are to be kept (Standard_False is used by default). Parameter <maxangle> sets a maximum possible angle for merging two adjacent edges, by default no limit angle is applied (-1). Results are put into the shape, which is given as parameter result.

Example:

fixsmalledges r a 0.1 1

fixshape

Syntax:
Performs fixes of all sub-shapes (such as Solids, Shells, Faces, Wires and Edges) of a given shape. Parameter `<preci>` sets a basic precision value, `<maxpreci>` sets the maximal allowed tolerance. Results are put into the shape, which is given as parameter result. `{switches}` allows to tune parameters of ShapeFix.

The following syntax is used:

- `<symbol>` may be
  - "-" to set parameter off,
  - "+" to set on or
  - "*" to set default
- `<parameter>` is identified by letters:
  - l – FixLackingMode
  - o – FixOrientationMode
  - h – FixShiftedMode
  - m – FixMissingSeamMode
  - d – FixDegeneratedMode
  - s – FixSmallMode
  - i – FixSelfIntersectionMode
  - n – FixNotchedEdgesMode

For enhanced message output, use switch '+?'

**Example:**

```
fixshape r a 0.001
```

**fixwgaps**

**Syntax:**

```
fixwgaps <result> <shape> [toler=0]
```

Fixes gaps between ends of curves of adjacent edges (both 3d and pcurves) in wires in a given shape with a given tolerance. Results are put into the shape, which is given as parameter result.
Example:

```
fixwgaps r a
```

**offsetcurve, offset2dcurve**

Syntax:

```
offsetcurve <result> <curve> <offset> <direction(as point)>
offset2dcurve <result> <curve> <offset>
```

**offsetcurve** works with the curve in 3d space, **offset2dcurve** in 2d space.

Both commands are intended to create a new offset curve by copying the given curve to distance, given by parameter `<offset>`. Parameter `<direction>` defines direction of the offset curve. It is created as a point. For correct work of these commands the direction of normal of the offset curve must be perpendicular to the plane, the basis curve is located there. Results are put into the curve, which is given as parameter `<result>`.

Example:

```
point pp 10 10 10
offsetcurve r c 20 pp
```

**projcurve**

Syntax:

```
projcurve <edge>|<curve3d>|<curve3d first last> <X> <Y> <Z>
```

**projcurve** returns the projection of a given point on a given curve. The curve may be defined by three ways: by giving the edge name, giving the 3D curve and by giving the unlimited curve and limiting it by pointing its start and finish values.
Example:

projcurve k_1 0 1 5
==Edge k_1 Params from 0 to 1.3
==Precision (BRepBuilderAPI) : 9.9999999999999995e-008  ==Projection : 0 1 5
==Result : 0 1.1000000000000001 0
==Param = -0.20000000000000001  Gap = 5.0009999000199947

projpcurve

Syntax:

projpcurve <edge> <face> <Tol> <X> <Y> <Z> [<start_param>]

projpcurve returns the projection of a given point on a given curve on surface. The curve on surface is defined by giving the edge and face names. Edge must have curve 2D repesentation on the face. Optional parameter <start_param> is any parameter of pcurve, which is used by algoritm as start point for searching projection of given point with help of local Extrema algorithm. If this parameter is not set, algorithm uses whole parametric interval of pcurve for searching projection.

Example:

# Using global searching
projpcurve f_1 f 1.e-7 0.877 0 0.479
==Point: 0.87762772831890712 0 0.47934285239977856
==Dist: 0.0007152557954264938

# Using starting parameter on edge
projpcurve f_1 f 1.e-7 0.877 0 0.479 .6
==Point: 0.87762772831890712 0 0.47934285239977856
==Dist: 0.0007152557954264938
projface

Syntax:

projface <face> <X> <Y> [<Z>]

Returns the projection of a given point to a given face in 2d or 3d space. If two coordinates (2d space) are given then returns coordinates projection of this point in 3d space and vice versa.

Example:

projface a_1 10.0 0.0
== Point UV U = 10 V = 0
== proj X = -116 Y = -45 Z = 0

scaleshape

Syntax:

scaleshape <result> <shape> <scale>

Returns a new shape, which is the result of scaling of a given shape with a coefficient equal to the parameter <scale>. Tolerance is calculated for the new shape as well.

Example:

scaleshape r a_1 0.8

settolerance

Syntax:

settolerance <shape> [<mode>=v-e-w-f-a] <val>(fix value) or <tolmin> <tolmax>

Sets new values of tolerance for a given shape. If the second parameter
mode is given, then the tolerance value is set only for these sub shapes.

**Example:**

```plaintext
settolerance a 0.001
```

**splitface**

Syntax:

```plaintext
splitface <result> <face> [u usplit1 usplit2...] [v vsplit1 vsplit2 ...]
```

Splits a given face in parametric space and puts the result into the given parameter `<result>`. Returns the status of split face.

**Example:**

```plaintext
# split face f by parameter u = 5
splitface r f u 5
==> Splitting by  U: 5
==> Status: DONE1
```

**statshape**

Syntax:

```plaintext
statshape <shape> [particul]
```

Returns the number of sub-shapes, which compose the given shape. For example, the number of solids, number of faces etc. It also returns the number of geometrical objects or sub-shapes with a specified type, example, number of free faces, number of C0 surfaces. The last parameter becomes out of date.

**Example:**

```plaintext
statshape a
==> Count Item
```
tolerance

Syntax:

```plaintext
tolerance <shape> [<mode>:D v e f c] [<tolmin> <tolmax>:real]
```

Returns tolerance (maximal, avg and minimal values) of all given shapes and tolerance of their Faces, Edges and Vertices. If parameter `<tolmin>` or `<tolmax>` or both of them are given, then sub-shapes are returned as a result of analysis of this shape, which satisfy the given tolerances. If a particular value of entity ((D)all shapes (v) vertices (e) edges (f) faces (c) combined (faces)) is given as the second parameter then only this group will be analyzed for tolerance.

Example:

```plaintext
tolerance a
>>> Tolerance MAX=0.31512672416608001
       AVG=0.14901359484722074 MIN=9.9999999999999995e-08
>>> FACE    : MAX=9.9999999999999995e-08
       AVG=9.9999999999999995e-08 MIN=9.9999999999999995e-08
>>> EDGE    : MAX=0.31512672416608001
       AVG=0.098691334511810405 MIN=9.9999999999999995e-08
```
tolerance a v 0.1 0.001

Analysing Vertices gives 6 Shapes between tol1=0.1000000000000001 and tol2=0.001, named tol_1 to tol_6
Conversion commands

DT_ClosedSplit

Syntax:

DT_ClosedSplit <result> <shape>

Divides all closed faces in the shape (for example cone) and returns result of given shape into shape, which is given as parameter result. Number of faces in resulting shapes will be increased. Note: A closed face is a face with one or more seam.

Example:

DT_ClosedSplit r a

DT_ShapeConvert, DT_ShapeConvertRev

Syntax:

DT_ShapeConvert <result> <shape> <convert2d> <convert3d>
DT_ShapeConvertRev <result> <shape> <convert2d> <convert3d>

Both commands are intended for the conversion of 3D, 2D curves to Bezier curves and surfaces to Bezier based surfaces. Parameters convert2d and convert3d take on a value 0 or 1. If the given value is 1, then the conversion will be performed, otherwise it will not be performed. The results are put into the shape, which is given as parameter Result. Command DT_ShapeConvertRev differs from DT_ShapeConvert by converting all elementary surfaces into surfaces of revolution first.

Example:

DT_ShapeConvert r a 1 1
== Status: DONE1
**DT_ShapeDivide**

Syntax:

```
DT_ShapeDivide <result> <shape> <tol>
```

Divides the shape with C1 criterion and returns the result of geometry conversion of a given shape into the shape, which is given as parameter result. This command illustrates how class `ShapeUpgrade_ShapeDivideContinuity` works. This class allows to convert geometry with a continuity less than the specified continuity to geometry with target continuity. If conversion is not possible then the geometrical object is split into several ones, which satisfy the given tolerance. It also returns the status shape splitting:

- **OK**: no splitting was done
- **Done1**: Some edges were split
- **Done2**: Surface was split
- **Fail1**: Some errors occurred

**Example:**

```
DT_ShapeDivide r a 0.001
== Status: OK
```

**DT_SplitAngle**

Syntax:

```
DT_SplitAngle <result> <shape> [MaxAngle=95]
```

Works with all revolved surfaces, like cylinders, surfaces of revolution, etc. This command divides given revolved surfaces into segments so that each resulting segment covers not more than the given `MaxAngle` degrees and puts the result of splitting into the shape, which is given as parameter result. Values of returned status are given above. This command illustrates how class `ShapeUpgrade_ShapeDivideAngle` works.

**Example:**
DT_SplitAngle r a
== Status: DONE2

**DT_SplitCurve**

Syntax:

```
DT_SplitCurve <curve> <tol> <split(0|1)>
```

Divides the 3d curve with C1 criterion and returns the result of splitting of the given curve into a new curve. If the curve had been divided by segments, then each segment is put to an individual result. This command can correct a given curve at a knot with the given tolerance, if it is impossible, then the given surface is split at that knot. If the last parameter is 1, then 5 knots are added at the given curve, and its surface is split by segments, but this will be performed not for all parametric spaces.

**Example:**

```
DT_SplitCurve r c
```

**DT_SplitCurve2d**

Syntax:

```
DT_SplitCurve2d Curve Tol Split(0/1)
```

Works just as **DT_SplitCurve** (see above), only with 2d curve.

**Example:**

```
DT_SplitCurve2d r c
```

**DT_SplitSurface**

Syntax:

```
DT_SplitSurface <result> <Surface|GridSurf> <tol>
```
<split(0|1)>

Divides surface with C1 criterion and returns the result of splitting of a given surface into surface, which is given as parameter result. If the surface has been divided into segments, then each segment is put to an individual result. This command can correct a given C0 surface at a knot with a given tolerance, if it is impossible, then the given surface is split at that knot. If the last parameter is 1, then 5 knots are added to the given surface, and its surface is split by segments, but this will be performed not for all parametric spaces.

Example:
split surface with name "su"

DT_SplitSurface res su 0.1 1 ==> single surf ==> appel a
SplitSurface::Init ==> appel a SplitSurface::Build ==> appel a
SplitSurface::GlobalU/VKnots ==> nb GlobalU;nb GlobalV=7 2 0 1 2 3 4
5 6.2831853072 0 1 ==> appel a Surfaces ==> transfert resultat ==>
res1_1_1 res1_2_1 res1_3_1 res1_4_1 res1_5_1 res1_6_1

DT_ToBspl

Syntax:

```
DT_ToBspl <result> <shape>
```

Converts a surface of linear extrusion, revolution and offset surfaces into
BSpline surfaces. Returns the result into the shape, which is given as
parameter result.

Example:

```
DT_ToBspl res sh
== error = 5.20375663162094e-08 spans = 10
== Surface is aproximated with continuity 2
```
Performance evaluation commands

VDrawSphere

Syntax:

```
vdrawsphere shapeName Fineness [X=0.0 Y=0.0 Z=0.0] [Radius=100.0] [ToEnableVBO=1] [NumberOfViewerUpdate=1] [ToShowEdges=0]
```

Calculates and displays in a given number of steps a sphere with given coordinates, radius and fineness. Returns the information about the properties of the sphere, the time and the amount of memory required to build it.

This command can be used for visualization performance evaluation instead of the outdated Visualization Performance Meter.

Example:

```
vdrawsphere s 200 1 1 1 500 1
== Compute Triangulation...
== NumberOfPoints: 39602
== NumberOfTriangles: 79200
== Amount of memory required for PolyTriangulation without Normals: 2 Mb
== Amount of memory for colors: 0 Mb
== Amount of memory for PolyConnect: 1 Mb
== Amount of graphic card memory required: 2 Mb
== Number of scene redrawings: 1
== CPU user time: 15.600099999998950 msec
== CPU system time: 0.0000000000000000 msec
== CPU average time of scene redrawing: 15.600099999998950 msec
```
Simple vector algebra and measurements

This section contains description of auxiliary commands that can be useful for simple calculations and manipulations needed when analyzing complex models.
Vector algebra commands

This section describes commands providing simple calculations with 2D and 3D vectors. The vector is represented by a TCL list of double values (coordinates). The commands get input vector coordinates from the command line as distinct values. So, if you have a vector stored in a variable you need to use eval command as a prefix, for example, to compute the magnitude of cross products of two vectors given by 3 points the following commands can be used:

```
0 30 0
4 -7 -10
Draw[14]> set cross [eval cross $vec1 $vec2]
-300 0 -120
Draw[15]> eval module $cross
323.10988842807024
```

**vec**

Syntax:

```
vec <x1> <y1> <z1> <x2> <y2> <z2>
```

Returns coordinates of vector between two 3D points.

Example:

```
vec 1 2 3 6 5 4
```

**2dvec**

Syntax:

```
2dvec <x1> <y1> <x2> <y2>
```
Returns coordinates of vector between two 2D points.

Example:

```
2dvec 1 2 4 3
```

**pln**

Syntax:

```
pln <x1> <y1> <z1> <x2> <y2> <z2> <x3> <y3> <z3>
```

Returns plane built on three points. A plane is represented by 6 double values: coordinates of the origin point and the normal directoin.

Example:

```
pln 1 2 3 6 5 4 9 8 7
```

**module**

Syntax:

```
module <x> <y> <z>
```

Returns module of a vector.

Example:

```
module 1 2 3
```

**2dmodule**

Syntax:

```
2dmodule <x> <y>
```

Returns module of a 2D vector.

Example:
2dmodule 1 2

**norm**

Syntax:

```plaintext
norm <x> <y> <z>
```

Returns unified vector from a given 3D vector.

Example:

```plaintext
norm 1 2 3
```

**2dnorm**

Syntax:

```plaintext
2dnorm <x> <y>
```

Returns unified vector from a given 2D vector.

Example:

```plaintext
2dnorm 1 2
```

**inverse**

Syntax:

```plaintext
inverse <x> <y> <z>
```

Returns inversed 3D vector.

Example:

```plaintext
inverse 1 2 3
```

**2dinverse**
Syntax:

2dinverse <x> <y>

Returns inversed 2D vector.

Example:

2dinverse 1 2

2dort

Syntax:

2dort <x> <y>

Returns 2D vector rotated on 90 degrees.

Example:

2dort 1 2

distpp

Syntax:

distpp <x1> <y1> <z1> <x2> <y2> <z2>

Returns distance between two 3D points.

Example:

distpp 1 2 3 4 5 6

2ddistpp

Syntax:

2ddistpp <x1> <y1> <x2> <y2>
Returns distance between two 2D points.

Example:

```
2ddistpp 1 2 3 4
```

**distplp**

Syntax:

```
distplp <x0> <y0> <z0> <nx> <ny> <nz> <xp> <yp> <zp>
```

Returns distance between plane defined by point and normal direction and another point.

Example:

```
distplp 0 0 0 0 0 1 5 6 7
```

**distlp**

Syntax:

```
distlp <x0> <y0> <z0> <dx> <dy> <dz> <xp> <yp> <zp>
```

Returns distance between 3D line defined by point and direction and another point.

Example:

```
distlp 0 0 0 1 0 0 5 6 7
```

**2ddistlp**

Syntax:

```
2ddistlp <x0> <y0> <dx> <dy> <xp> <yp>
```

Returns distance between 2D line defined by point and direction and another point.
Example:

2ddistlp 0 0 1 0 5 6

**distppp**

Syntax:

```
distppp <x1> <y1> <z1> <x2> <y2> <z2> <x3> <y3> <z3>
```

Returns deviation of point (x2,y2,z2) from segment defined by points (x1,y1,z1) and (x3,y3,z3).

Example:

```
distppp 0 0 0 1 1 0 2 0 0
```

**2ddistppp**

Syntax:

```
2ddistppp <x1> <y1> <x2> <y2> <x3> <y3>
```

Returns deviation of point (x2,y2) from segment defined by points (x1,y1) and (x3,y3). The result is a signed value. It is positive if the point (x2,y2) is on the left side of the segment, and negative otherwise.

Example:

```
2ddistppp 0 0 1 -1 2 0
```

**barycen**

Syntax:

```
barycen <x1> <y1> <z1> <x2> <y2> <z2> <par>
```

Returns point of a given parameter between two 3D points.

Example:
**2dbarycen**

Syntax:

```
2dbarycen <x1> <y1> <x2> <y2> <par>
```

Returns point of a given parameter between two 2D points.

Example:

```
2dbarycen 0 0 1 1 0.3
```

**cross**

Syntax:

```
cross <x1> <y1> <z1> <x2> <y2> <z2>
```

Returns cross product of two 3D vectors.

Example:

```
cross 1 0 0 0 1 0
```

**2dcross**

Syntax:

```
2dcross <x1> <y1> <x2> <y2>
```

Returns cross product of two 2D vectors.

Example:

```
2dcross 1 0 0 1
```

**dot**
Syntax:

```plaintext
dot <x1> <y1> <z1> <x2> <y2> <z2>
```

Returns scalar product of two 3D vectors.

Example:

```
dot 1 0 0 0 1 0
```

2ddot

Syntax:

```plaintext
2ddot <x1> <y1> <x2> <y2>
```

Returns scalar product of two 2D vectors.

Example:

```
2ddot 1 0 0 1
```

scale

Syntax:

```plaintext
scale <x> <y> <z> <factor>
```

Returns 3D vector multiplied by scalar.

Example:

```
scale 1 0 0 5
```

2dscale

Syntax:

```plaintext
2dscale <x> <y> <factor>
```
Returns 2D vector multiplied by scalar.

Example:

```
2dscale 1 0 5
```
Measurements commands

This section describes commands that make possible to provide measurements on a model.

**pnt**

Syntax:

```
pnt <object>
```

Returns coordinates of point in the given Draw variable. Object can be of type point or vertex. Actually this command is built up from the commands `mkpoint` and `coord`.

Example:

```
vertex v 0 1 0
pnt v
```

**pntc**

Syntax:

```
pntc <curv> <par>
```

Returns coordinates of point on 3D curve with given parameter. Actually this command is based on the command `cvalue`.

Example:

```
circle c 0 0 0 10
pntc c [dval pi/2]
```

**2dpntc**

Syntax:
2dpntc <curv2d> <par>

Returns coordinates of point on 2D curve with given parameter. Actually this command is based on the command 2dcvalue.

Example:

circle c 0 0 10
2dpntc c [dval pi/2]

pntsu

Syntax:

pntsu <surf> <u> <v>

Returns coordinates of point on surface with given parameters. Actually this command is based on the command svalue.

Example:

cylinder s 10
pntsu s [dval pi/2] 5

pntcons

Syntax:

pntcons <curv2d> <surf> <par>

Returns coordinates of point on surface defined by point on 2D curve with given parameter. Actually this command is based on the commands 2dcvalue and svalue.

Example:

line c 0 0 1 0
cylinder s 10
pntcons c s [dval pi/2]
**drseg**

Syntax:

```
drseg <name> <x1> <y1> <z1> <x2> <y2> <z2>
```

Creates a linear segment between two 3D points. The new object is given the *name*. The object is drawn in the axonometric view.

Example:

```
drseg s 0 0 0 1 0 0
```

**2ddrseg**

Syntax:

```
2ddrseg <name> <x1> <y1> <x2> <y2>
```

Creates a linear segment between two 2D points. The new object is given the *name*. The object is drawn in the 2D view.

Example:

```
2ddrseg s 0 0 1 0
```

**mpick**

Syntax:

```
mpick
```

Prints in the console the coordinates of a point clicked by mouse in a view (axonometric or 2D). This command will wait for mouse click event in a view.

Example:

```
mpick
```
mdist

Syntax:

mdist

Prints in the console the distance between two points clicked by mouse in a view (axonometric or 2D). This command will wait for two mouse click events in a view.

Example:

mdist
Inspector commands

This section describes commands that make possible to use Inspector.
tinspector

Syntax:

```
tinspector [-plugins {name1 ... [nameN] | all}]
[-activate name]
[-shape object [name1] ... [nameN]]
[-open file_name [name1] ... [nameN]]
[-update]
[-select {object | name1 ... [nameN]}]
[-show {0|1} = 1]
```

Starts tool of inspection. Options:

- **plugins** enters plugins that should be added in the inspector. Available names are: dfbrowser, vinspector and shapeview. Plugins order will be the same as defined in arguments. 'all' adds all available plugins in the order: DFBrowser, VInspector and ShapeView. If at the first call this option is not used, 'all' option is applied;
- **activate** activates the plugin in the tool view. If at the first call this option is not used, the first plugin is activated;
- **shape** initializes plugin/s by the shape object. If 'name' is empty, initializes all plugins;
- **open** gives the file to the plugin/s. If the plugin is active, after open, update content will be done;
- **update** updates content of the active plugin;
- **select** sets the parameter that should be selected in an active tool view. Depending on active tool the parameter is: ShapeView: 'object' is an instance of TopoDS_Shape TShape, DFBrowser: 'name' is an entry of TDF_Label and name2(optionaly) for TDF_Attribute type name, VInspector: 'object' is an instance of AIS_InteractiveObject;
- **show** sets Inspector view visible or hidden. The first call of this command will show it.

**Example:**

```
pload DCAF INSPECTOR
```
NewDocument Doc BinOcaf

set aSetAttr1 100  
set aLabel 0:2  
SetInteger Doc ${aLabel} ${aSetAttr1} 

tinspector -plugins dfbrowser -select 0:2  
   TDataStd_Integer

Example:

pload ALL INSPECTOR

box b1  200 100 120
box b2 100 200 220 100 120 100

tinspector -plugins shapeview -shape b1 -shape b2 -
  select b1

Example:

pload ALL INSPECTOR

tinspector -plugins vinspector  

vinit
box box_1 100 100 100  
vdisplay box_1

box box_2 180 120 200 150 150 150  
vdisplay box_2

vfit
vselmode box_1 1 1
vselmode box_1 3 1

tinspector -update -select box_1
Extending Test Harness with custom commands

The following chapters explain how to extend Test Harness with custom commands and how to activate them using a plug-in mechanism.
Custom command implementation

Custom command implementation has not undergone any changes since the introduction of the plug-in mechanism. The syntax of every command should still be like in the following example.

Example:

```cpp
static Standard_Integer myadvcurve(Draw_Interpretor& di, Standard_Integer n, char** a)
{
  ...
}
```

For examples of existing commands refer to Open CASCADE Technology (e.g. GeomliteTest.cxx).
Registration of commands in Test Harness

To become available in the Test Harness the custom command must be registered in it. This should be done as follows.

Example:

```cpp
void MyPack::CurveCommands(Draw_Interpretor& theCommands)
{
  ...
  char* g = "Advanced curves creation";

  theCommands.Add ( "myadvcurve", "myadvcurve name p1 p2 p3 - Creates my advanced curve from points", __FILE__, myadvcurve, g );
  ...
}
```
Creating a toolkit (library) as a plug-in

All custom commands are compiled and linked into a dynamic library (.dll on Windows, or .so on Unix/Linux). To make Test Harness recognize it as a plug-in it must respect certain conventions. Namely, it must export function `PLUGINFACTORY()` accepting the Test Harness interpreter object (`Draw_Interpretor`). This function will be called when the library is dynamically loaded during the Test Harness session.

This exported function `PLUGINFACTORY()` must be implemented only once per library.

For convenience the `DPLUGIN` macro (defined in the `Draw_PluginMacro.hxx` file) has been provided. It implements the `PLUGINFACTORY()` function as a call to the `Package::Factory()` method and accepts `Package` as an argument. Respectively, this `Package::Factory()` method must be implemented in the library and activate all implemented commands.

Example:

```c++
#include <Draw_PluginMacro.hxx>

void MyPack::Factory(Draw_Interpretor& theDI) {
    ...
    // MyPack::CurveCommands(theDI);
    ...
}

// Declare entry point PLUGINFACTORY
DPLUGIN(MyPack)
```
Creation of the plug-in resource file

As mentioned above, the plug-in resource file must be compliant with Open CASCADE Technology requirements (see Resource_Manager.hxx file for details). In particular, it should contain keys separated from their values by a colon (;;). For every created plug-in there must be a key. For better readability and comprehension it is recommended to have some meaningful name. Thus, the resource file must contain a line mapping this name (key) to the library name. The latter should be without file extension (.dll on Windows, .so on Unix/Linux) and without the ;lib; prefix on Unix/Linux. For several plug-ins one resource file can be created. In such case, keys denoting plug-ins can be combined into groups, these groups – into their groups and so on (thereby creating some hierarchy). Any new parent key must have its value as a sequence of child keys separated by spaces, tabs or commas. Keys should form a tree without cyclic dependencies.

Examples (file MyDrawPlugin):

```
! Hierarchy of plug-ins
ALL : ADVMODELING, MESHING
DEFAULT : MESHING
ADVMODELING : ADVSURF, ADVCURV

! Mapping from naming to toolkits (libraries)
ADVSURF : TKMyAdvSurf
ADVCURV : TKMyAdvCurv
MESHING : TKMyMesh
```

For other examples of the plug-in resource file refer to the Plug-in resource file chapter above or to the $CASROOT/src/DrawPlugin file shipped with Open CASCADE Technology.
Dynamic loading and activation

Loading a plug-in and activating its commands is described in the Activation of the commands implemented in the plug-in chapter.

The procedure consists in defining the system variables and using the `pload` commands in the Test Harness session.

**Example:**

```
Draw[]> set env(CSF_MyDrawPluginDefaults) /users/test
Draw[]> pload -MyDrawPlugin ALL
```
Inspector

Table of Contents

- Introduction
  - Overview
  - Getting started
- Inspector
  - Overview
- DFBrowser Plugin
  - Overview
  - Elements
  - Elements cooperation
  - TopoDS_Shape export
- VInspector Plugin
  - Overview
  - Elements
  - Elements cooperation
- ShapeView Plugin
  - Overview
  - Elements
  - Elements cooperation
- Common controls
  - 3D View
    - Overview
    - Elements
- TlnspectorEXE sample
- Launch in DRAW Test
Introduction

This manual explains how to use Inspector.
Overview

Inspector is a Qt-based library that provides functionality to interactively inspect low-level content of the OCAF data model, OCCT viewer and Modelisation Data. This component is aimed to assist the developers of OCCT-based applications to debug the problematic situations that occur in their applications.

Inspector has a plugin-oriented architecture. The current release contains the following plugins:

<table>
<thead>
<tr>
<th>Plugin</th>
<th>OCCT component</th>
<th>Root class of OCCT investigated component</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFBrowser</td>
<td>OCAF</td>
<td>TDocStd_Application</td>
</tr>
<tr>
<td>VInspector</td>
<td>Visualization</td>
<td>AIS_InteractiveContext</td>
</tr>
<tr>
<td>ShapeView</td>
<td>Modelisation Data</td>
<td>TopoDS_Shape</td>
</tr>
</tbody>
</table>

Each plugin implements logic of a corresponding OCCT component.

Each of the listed plugins is embedded in the common framework. The user is able to manage which plugins should be loaded by Inspector. Also he can extend number of plugins by implementing a new plugin.
Getting started

There are two launch modes:

1. Launch TInspectorEXE executable sample. For more details see TInspectorEXE section;
2. Launch DRAW, load plugin INSPECTOR, and use tinspector command. For more details see Launch in DRAW Test Harness section.

Note. If you have no Inspector library in your build directory, please make sure that OCCT is compiled with BUILD_Inspector option ON. For more details see Build procedure.
Inspector

Overview

Inspector consists of the following components:

- **buttons** to activate the corresponding plugin;
- **view area** to visualize the plugin content.

Plugins placement in Inspector
DFBrowser Plugin

Overview

This plugin visualizes content of TDocStd_Application in a tree view. It shows documents of the application, hierarchy of TDF_Labels, content of TDF_Attributes and interconnection between attributes (e.g. references). Additionally it has 3D view to visualize TopoDS_Shape elements stored in the document.

Elements
OCAF tree view

Each OCAF element has own tree view item:

<table>
<thead>
<tr>
<th>Type</th>
<th>Tree item</th>
<th>Text</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDocStd_Application</td>
<td>Application</td>
<td>TDocStd_Application</td>
<td>It is the root of tree view. Children are documents.</td>
</tr>
<tr>
<td>TDocStd_Document</td>
<td>Document</td>
<td>entry : name</td>
<td>It is a child of Application item. Children are Labels and Attributes item. Text view is an entry of the root label and the value of TDataStd_Name.</td>
</tr>
<tr>
<td>TDF_Label</td>
<td>Label</td>
<td>entry : name</td>
<td>It is a child of a Document or another Label item. Children and text view are the same as for Document item.</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>--------------</td>
<td>----------------------------------------------------------------</td>
</tr>
<tr>
<td>TDF_Attribute</td>
<td>Attribute</td>
<td>attribute type [additional information]</td>
<td>It is a child of a Label. It has no children. Text view is the attribute type (DynamicType().Name() of TDF_Attribute) and additional information (a combination of attribute value</td>
</tr>
</tbody>
</table>

Additional information of TDF_Attributes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDocStd_Owner</td>
<td>[storage format]</td>
</tr>
<tr>
<td>TDataStd_AsciiString, TDataStd_Name, TDataStd_Real, other Simple types</td>
<td>[value]</td>
</tr>
<tr>
<td>TDataStd_BooleanList, TDataStd_EXTStringList, other List types</td>
<td>[value_1 ... value_n]</td>
</tr>
<tr>
<td>TDataStd_BooleanArray, TDataStd_BYTEArray, other Array types</td>
<td>[value_1 ... value_n]</td>
</tr>
<tr>
<td></td>
<td>[tree node ID ==&gt; Father()-&gt;Label()] (if it</td>
</tr>
</tbody>
</table>
TDataStd_TreeNode

has father) or [tree node ID <= First()->Label()] (if it has NO father)

TDataStd_TreeNode(XDE)

[XDE tree node ID ==> Father()->Label()] (if it has father),
[XDE tree Node ID <= label_1, ..., label_n] (if it has NO father)

TNaming_NamedShape

[shape type : evolution]

TNaming_UsedShapes

[map extent]

Custom color of items:

<table>
<thead>
<tr>
<th>OCAF element Type</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDF_Label</td>
<td><strong>dark green</strong>, if the label has TDataStd_Name attribute, <strong>light grey</strong> if the label is empty (has no attributes on all levels of hierarchy), <strong>black</strong> otherwise</td>
</tr>
<tr>
<td>TNaming_NamedShape</td>
<td><strong>dark gray</strong> for TopAbs_FORWARD orientation of TopoDS_Shape, <strong>gray</strong> for TopAbs_REVERSED orientation of TopoDS_Shape, <strong>black</strong> for other orientation</td>
</tr>
</tbody>
</table>

Context popup menu:

<table>
<thead>
<tr>
<th>Action</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expand</td>
<td>Expands the next two levels under the selected item</td>
</tr>
<tr>
<td>Expand All</td>
<td>Expands the whole tree of the selected item</td>
</tr>
<tr>
<td>Collapse All</td>
<td>Collapses the whole tree of the selected item</td>
</tr>
</tbody>
</table>

**Property Panel**

Property panel is used to display content of Label or Attribute tree view items. This control is used for content of Label or Attribute tree view items or Search result view. Information is usually shown in one or several
TDF_Attribute has the following content in Property Panel:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDF_Label</td>
<td>a table of [entry or attribute name, value]</td>
<td></td>
</tr>
<tr>
<td>TDocStd_Owner, <strong>Simple types</strong>, <strong>List types</strong></td>
<td>a table of [method name, value]</td>
<td></td>
</tr>
<tr>
<td>TDataStd_BooleanArray, TDataStd_ByteArray, <strong>other Array types</strong></td>
<td>2 controls: * a table of [array bound, value], * table of [method name, value]</td>
<td></td>
</tr>
<tr>
<td>TDataStd_TreeNode</td>
<td>2 controls: * a table of [Tree ID, value] (visible only if Tree ID() != ID()), * a tree view of tree nodes starting from Root() of the tree node. The current tree node has dark blue text.</td>
<td></td>
</tr>
<tr>
<td>TDataStd_NamedData</td>
<td>tab bar of attribute elements, each tab has a table of [name, value]</td>
<td></td>
</tr>
<tr>
<td>TNaming_UsedShapes</td>
<td>a table of all the shapes handled by the framework</td>
<td></td>
</tr>
<tr>
<td>TNaming_NamedShape</td>
<td>2 controls: * a table of [method name, value] including CurrentShape/OriginalShape methods result of TNaming_Tools, * an evolution table. Tables contain buttons for</td>
<td></td>
</tr>
</tbody>
</table>
### TopoDS_Shape export.

| TNaming_Naming | 2 controls:  
|                | * a table of TNaming_Name values,  
|                | * a table of [method name, value] |

### Dump view

**Dump of TDF_Attribute**

Dump view shows result of `TDF_Attribute::Dump()` or `TDF_Label::Dump()` of selected tree view item.

### 3D view

3D View visualizes TopoDS_Shape elements of OCAF attribute via AIS facilities.

DFBrowser creates two kinds presentations depending on the selection place:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Source object</th>
<th>Visualization properties</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main presentation</td>
<td>Tree view item: TPrsStd_AISPresentation, TNaming_NamedShape, TNaming_Naming</td>
<td>Color: a default color for shape type of the current TopoDS_Shape</td>
<td><img src="image.png" alt="Main presentation" /></td>
</tr>
<tr>
<td>Additional</td>
<td>References in Property panel</td>
<td>Color: white</td>
<td><img src="image.png" alt="Additional presentation" /></td>
</tr>
</tbody>
</table>
Tree Navigation

Tree Navigation shows a path to the item selected in the tree view. The path is a sequence of label entries and attribute type name. Each element in the path is selectable - the user can click on it to select the corresponding tree view item.

Navigation control has buttons to go to the previous and the next selected tree view items.

Update Button

Update button synchronizes content of tree view to the current content of OCAF document that could be modified outside.

Search

The user can search OCAF element by typing:

- TDF__Label entry,
- TDF_Attribute name,
- TDataStd_Name and TDataStd_Comment attributes value.
As soon as the user confirms the typed criteria, the Property panel is filled by all satisfied values. The user can click a value to highlight the corresponding tree view item. By double click the item will be selected.

Elements cooperation

Tree item selection

Selection of tree view item updates content of the following controls:

- Navigation line
- Property Panel
- 3D View (if it is possible to create an interactive presentation)
- Dump View

Property Panel item selection

If property panel shows content of TDF_Label:

- selection of the table row highlights the corresponding item in tree view,
- double click on the table row selects this item in tree view.

If property panel shows content of TDF_Attribute that has reference to
another attribute, selection of this reference:

- highlights the referenced item in TreeView,
- displays additional presentation in 3D view if it can be created.

<table>
<thead>
<tr>
<th>Type</th>
<th>Reference</th>
<th>Additional presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDF_Reference</td>
<td>TDF_Label</td>
<td></td>
</tr>
<tr>
<td>TDataStd_ReferenceArray, TDataStd_ReferenceList, TNaming_Naming</td>
<td>one or several TDF_Label in a container</td>
<td></td>
</tr>
<tr>
<td>TDataStd_TreeNode</td>
<td>TDF_Label</td>
<td></td>
</tr>
<tr>
<td>TNaming_NamedShape</td>
<td>TDF_Label in Evolution table</td>
<td>selected TopoDS_Shapes in property panel table</td>
</tr>
<tr>
<td>TNaming_UsedShapes</td>
<td>one or several TNaming_NamedShape</td>
<td></td>
</tr>
</tbody>
</table>
TopoDS_Shape export

Property panel of TNaming_NamedShape attribute has controls to export TopoDS_Shape to:

- BREP. The save file dialog is started to enter the result file name,
- ShapeView plugin. Dialog about exporting element to ShapeView is shown with a possibility to activate this plugin immediately.
VInspector Plugin

Overview

VInspector

It visualizes interactive objects displayed in AIS_InteractiveContext in a tree view with computed selection components for each presentation. It shows the selected elements in the context and allows to select these elements.

Elements
Presentations tree view

It shows presentations and selection computed of them. Also, the view has columns with information about state of visualization elements.

VInspector tree items.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIS_InteractiveContext</td>
<td>It is the root of tree view. Children are interactive objects obtained by DisplayedObjects and ErasedObjects methods.</td>
</tr>
<tr>
<td>AIS_InteractiveObject</td>
<td>It is a child of AIS_InteractiveContext item. Children are SelectMgr_Selection obtained by iteration on CurrentSelection</td>
</tr>
<tr>
<td>SelectMgr_Selection</td>
<td>It is a child of AIS_InteractiveObject. Children are SelectMgr_SensitiveEntity obtaining by iteration on Sensitive</td>
</tr>
<tr>
<td>SelectMgr_SensitiveEntity</td>
<td>It is a child of SelectMgr_Selection. Children are SelectMgr_SensitiveEntity</td>
</tr>
<tr>
<td>OCAF element Type</td>
<td>Column</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>AIS_InteractiveObject</td>
<td>0</td>
</tr>
<tr>
<td>AIS_InteractiveObject, SelectMgr_SensitiveEntity, SelectBasics_EntityOwner</td>
<td>1</td>
</tr>
<tr>
<td>SelectMgr_Selection, SelectMgr_SensitiveEntity, SelectBasics_EntityOwner</td>
<td>all</td>
</tr>
</tbody>
</table>

Context popup menu in tree view:

<table>
<thead>
<tr>
<th>Action</th>
<th>Item</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Export to ShapeView</td>
<td>AIS_InteractiveObject</td>
<td>Exports TopoDS_Shape of AIS_Interactive presentation to ShapeView plugin. It should be AIS_Shape presentation and ShapeView plugin should be registered in Inspector. Dialog about exporting element to ShapeView is shown with a possibility to activate this plugin immediatelly.</td>
</tr>
<tr>
<td>Show</td>
<td>AIS_InteractiveObject</td>
<td>Display presentation in AIS_InteractiveContext</td>
</tr>
</tbody>
</table>
Update

It synchronizes content of the plugin to the current state of AIS_InteractiveContext. It updates the presence of items and the current selection for the items.

Selection controls

Selection controls switch on/off the possibility to set selection in the context from VInspector plugin.

<table>
<thead>
<tr>
<th>Action</th>
<th>Tree view item</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Presentations</td>
<td>AIS_InteractiveObject</td>
<td>Calls AddOrRemoveSelected of interactive object for the selected item</td>
</tr>
<tr>
<td>Select Owners</td>
<td>SelectMgr_EntityOwner or SelectMgr_SensitiveEntity</td>
<td>Calls AddOrRemoveSelected of SelectMgr_EntityOwner for the selected item</td>
</tr>
</tbody>
</table>

Please note, that the initial selection in context will be cleared. If the button is toggled, the button selection is active. Only one button may be toggled at the moment.

History view

At present the History view is under implementation and may be used only in a custom application where Inspector is loaded.

To fill this view, VInspectorAPI_CallBack should be redefined in the application and send signals about some actions applied to context. After, the call back should be given as parameter in the plugin. If done, new items will be created in the history view for each action.

Elements cooperation
Vinspector marks current selected presentations in AIS_InteractiveContext with blue background in tree items. Use "Update" button to synchronize VInspector selected items state to the context.

It is also possible to perform selection in context using "Selection controls" VInspector. However, it should be performed carefully as it clears the current selection in AIS_InteractiveContext.

Selection change:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIS_InteractiveContext</td>
<td>VInspector</td>
<td>perform selection in AIS_InteractiveContext</td>
</tr>
<tr>
<td>VInspector</td>
<td>AIS_InteractiveContext</td>
<td>activate one of Selection controls and select one or several elements in tree view</td>
</tr>
</tbody>
</table>
ShapeView Plugin

Overview

This plugin visualizes content of TopoDS_Shape in a tree view.

Elements
Elements of the view are TopoDS_Shape objects. This shape is exploded into sub-shapes using TopoDS_Iterator of the TopoDS_Shape. Child sub-shapes are presented in the view as children of the initial shape. Iterating recursively by all shapes we obtain a tree view of items shown in the ShapeView.

Columns of the View show some information about TopoDS_Shape of the item. The most informative column is the last column of TopoDS_Vertex and TopoDS_Edge shape types.

For TopoDS_Vertex it contains the point coordinates,

for TopoDS_Edge it contains the first and the last point coordinates, the edge length and some other parameters.

Context popup menu in tree view:

<table>
<thead>
<tr>
<th>Action</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Opens selected file and appends the result</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BREP file</td>
<td>TopoDS.Shape into tree view</td>
</tr>
<tr>
<td>Remove all shape items</td>
<td>Clears tree view</td>
</tr>
<tr>
<td>BREP view</td>
<td>Shows text view with BREP content of the selected item. It creates BREP file in temporary directory of the plugin.</td>
</tr>
<tr>
<td>Close All BREP views</td>
<td>Closes all opened text views</td>
</tr>
<tr>
<td>BREP directory</td>
<td>Displays folder where temporary BREP files have been stored.</td>
</tr>
</tbody>
</table>

**Elements cooperation**

Selection of one or several items in TopoDS._Shape View creates AIS._Shape presentation for it and displays it in the 3D View.
Common controls

3D View

Overview

Control for OCCT 3D viewer. It creates visualization view components with possibilities to perform some user actions for the view.

Elements
3D View contains:

<table>
<thead>
<tr>
<th>Element</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D view</td>
<td>V3d viewer with mouse events processing</td>
</tr>
<tr>
<td>Context</td>
<td>choice of another context that should be used in the plugin. It is possible to use the next contexts: Own - context of this view, External - context come in parameters which initializes plugin, None - do not perform visualization at all</td>
</tr>
<tr>
<td>Multi/Simple</td>
<td>Buttons defined what to do with the previous displayed objects: Multi displays new presentations in additional to already displayed, Single removes all previous displayed presentations</td>
</tr>
<tr>
<td>Clean</td>
<td>Removes all displayed presentations</td>
</tr>
<tr>
<td>Fit All, Fit Area, Zoom,</td>
<td>Scene manipulation actions</td>
</tr>
<tr>
<td>Pan, Rotation</td>
<td>Display Mode</td>
</tr>
</tbody>
</table>
TInspectorEXE sample

Inspector functionality can be tried using this sample.

Use `inspector.bat` script file placed in binary directory of OCCT to launch it.

This script accepts the names of plugin's DLL that should be loaded. By default it loads all described above plugins.

![TDocStd_Application](image)

TStandaloneEXE

Click on the Open button shows the dialog to select a file. The user is able to select one of the sample files or load own one.
Depending on the active plugin, the following files should be selected in the dialog: OCAF document or STEP files for DFBrowser and BREP files for VInspector and ShapeView plugins.

It is possible to click the file name in the proposed directory, enter it manually or using Browser button. The last Loading icon becomes enabled if file name is correct.

By default TInspectorEXE opens the next files for plugins:

<table>
<thead>
<tr>
<th>Plugin DLL library name</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>TKDFBrowser</td>
<td>step/screw.step</td>
</tr>
<tr>
<td>TKVInspector</td>
<td>occ/hammer.brep</td>
</tr>
<tr>
<td>TKShapeView</td>
<td>occ/face1.brep,</td>
</tr>
<tr>
<td></td>
<td>occ/face2.brep</td>
</tr>
</tbody>
</table>

These files are found relatively `CSF_OCCTDataPath`.

Source code of TInspectorEXE is a good sample for Using Inspector in a custom application.
Launch in DRAW Test Harness

TKToolsDraw plugin is created to provide DRAW commands for Qt tools. Use INSPECTOR parameter of `pload` command to download commands of this library. It contains `tinspector` command to start Inspector under DRAW. See more detailed description of the `tinspector` command.

The simple code to start Inspector with all plugins loaded:

```
pload INSPECTOR
tinspector
```

Result of this command is the next:

- all available Plugins are presented in the Inspector. These are `DFBrowser`, `VInspector` and `ShapeView`.
- DFBrowser is an active plugin
- tree of OCAF is empty.

After, we should create objects in DRAW and update `tinspector`. 
Using in a custom application

To use Inspector in an application, the next steps should be done:

- Set dependencies to OCCT and Qt in application (Header and Link)
- Create an instance of TInspector_Communicator.
- Register plugins of interest in the communicator by DLL library name
- Initialize communicator with objects that will be investigated
- Set visible true for communicator

C++ code is similar:

```c++
#include <inspector/TInspector_Communicator.hxx>

static TInspector_Communicator* MyTCommunicator;

void CreateInspector()
{
    NCollection_List<Handle(Standard_Transient)> aParameters;
    //... append parameters in the list

    if (!MyTCommunicator)
    {
        MyTCommunicator = new TInspector_Communicator();

        MyTCommunicator->RegisterPlugin ("TKDFBrowser");
        MyTCommunicator->RegisterPlugin ("TKVInspector");
        MyTCommunicator->RegisterPlugin ("TKShapeView");

        MyTCommunicator->Init (aParameters);
        MyTCommunicator->Activate ("TKDFBrowser");
    }
    MyTCommunicator->SetVisible (true);
}
<table>
<thead>
<tr>
<th>TKDFBrowser</th>
<th>TDocStd_Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>TKVInspector</td>
<td>AIS_InteractiveContext</td>
</tr>
<tr>
<td>TKShapeView</td>
<td>TopoDS_TShape</td>
</tr>
</tbody>
</table>
Build procedure

By default the Inspector compilation is off. To compile it, set the `BUILD_Inspector` flag to "ON". See Configuration process.

When this option is switched On, MS Visual Studio project has an additional tree of folders:

Inspector packages in MS Visual Studio
Sources and packaging

OCCT sources are extended by the /tools directory.

Distribution of packages participated in plugins:

<table>
<thead>
<tr>
<th>Sources packages</th>
<th>Plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFBrowser, DFBrowserPane, DFBrowserPaneXDE, TKDFBrowser</td>
<td>DFBrowser</td>
</tr>
<tr>
<td>VInspector, TKVInspector</td>
<td>VInspector</td>
</tr>
<tr>
<td>ShapeView, TKShapeView</td>
<td>ShapeView</td>
</tr>
</tbody>
</table>

Other packages:

<table>
<thead>
<tr>
<th>Sources packages</th>
<th>Used in</th>
</tr>
</thead>
<tbody>
<tr>
<td>TInspectorAPI, TKInspectorAPI</td>
<td>Interface for connection to plugin.</td>
</tr>
<tr>
<td>TreeModel, TKTTreeView</td>
<td>Items-oriented model to simplify work with GUI tree control.</td>
</tr>
<tr>
<td>View, TKView</td>
<td>3D View component</td>
</tr>
<tr>
<td>TInspector, TKTInspector</td>
<td>Inspector window where plugins are placed</td>
</tr>
<tr>
<td>ToolsDraw, TKToolsDraw</td>
<td>Plugin for DRAW to start Inspector</td>
</tr>
</tbody>
</table>

In MSVC studio the separate folder contains Inspector projects.
Glossary

- **Component** – OCCT part, e.g. OCAF, VISUALIZATION, MODELING and others.
- **Plugin** – library that is loaded in some executable/library. Here, the plugins are:
  - DFBrowser,
  - ShapeView,
  - VInspector.
# TDF_Attribute Simple types

Types where the content is a single value

<table>
<thead>
<tr>
<th>Type</th>
<th>Kind of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDataStd_AsciiString</td>
<td>TDataStd_AsciiString</td>
</tr>
<tr>
<td>TDataStd_Comment</td>
<td>TCollection_ExtendedString</td>
</tr>
<tr>
<td>TDataStd_Integer</td>
<td>Standard_Integer</td>
</tr>
<tr>
<td>TDataStd_Name</td>
<td>TCollection_ExtendedString</td>
</tr>
<tr>
<td>TDataStd_Real</td>
<td>Standard_Real</td>
</tr>
<tr>
<td>TDF_Reference</td>
<td>TDF_Label</td>
</tr>
<tr>
<td>TDF_TagSource</td>
<td>Standard_Integer</td>
</tr>
</tbody>
</table>
# TDF_Attribute List types

<table>
<thead>
<tr>
<th>Type</th>
<th>Kind of value (container of)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDataStd_BooleanList</td>
<td>Standard_Boolean</td>
</tr>
<tr>
<td>TDataStd_ExtStringList</td>
<td>TCollection_ExtendedString</td>
</tr>
<tr>
<td>TDataStd_IntegerList</td>
<td>Standard_Integer</td>
</tr>
<tr>
<td>TDataStd_RealList</td>
<td>Standard_Real</td>
</tr>
<tr>
<td>TDataStd_ReferenceList</td>
<td>TDF_Label</td>
</tr>
</tbody>
</table>
# TDF_Attribute Array types

<table>
<thead>
<tr>
<th>Type</th>
<th>Kind of value (container of)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDataStd_BooleanArray</td>
<td>Standard_Boolean</td>
</tr>
<tr>
<td>TDataStd_ByteArray</td>
<td>Standard_Byte</td>
</tr>
<tr>
<td>TDataStd_ExtStringArray</td>
<td>TCollection_ExtendedString</td>
</tr>
<tr>
<td>TDataStd_IntegerArray</td>
<td>Standard_Integer</td>
</tr>
<tr>
<td>TDataStd_RealArray</td>
<td>Standard_Real</td>
</tr>
<tr>
<td>TDataStd_ReferenceArray</td>
<td>TDF_Label</td>
</tr>
</tbody>
</table>
### XDE tree node ID description

<table>
<thead>
<tr>
<th>GUID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCAFDoc::ShapeRefGUID()</td>
<td>Shape Instance Link</td>
</tr>
<tr>
<td>XCAFDoc::ColorRefGUID (XCAFDoc_ColorGen)</td>
<td>Generic Color Link</td>
</tr>
<tr>
<td>XCAFDoc::ColorRefGUID (XCAFDoc_ColorSurf)</td>
<td>Surface Color Link</td>
</tr>
<tr>
<td>XCAFDoc::ColorRefGUID (XCAFDoc_ColorCurv)</td>
<td>Curve Color Link</td>
</tr>
<tr>
<td>XCAFDoc::DimTolRefGUID()</td>
<td>DGT Link</td>
</tr>
<tr>
<td>XCAFDoc::DatumRefGUID()</td>
<td>Datum Link</td>
</tr>
<tr>
<td>XCAFDoc::MaterialRefGUID()</td>
<td>Material Link</td>
</tr>
</tbody>
</table>
The following documents provide information on OCCT building, development and testing:

- Building OCCT from sources
- Documentation system
- Coding Rules
- Contribution Workflow
- Guide to installing and using Git for OCCT development
- Automatic Testing system
- Debugging tools and hints

The following guide provides information relevant to upgrading applications developed with previous versions of OCCT, to recent one:

- Upgrade from previous OCCT versions
Building OCCT from sources

Before building OCCT, make sure to have all the required third-party libraries installed. The list of required libraries depends on what OCCT modules will be used, and your preferences. The typical minimum is **Freetype** (necessary for Visualization) and **Tcl/Tk** (for DRAW Test Harness). See "Third-party libraries" section in [Overview](#) for a full list.

On Windows, the easiest way to install third-party libraries is to download archive with pre-built binaries from [http://www.opencascade.com/content/3rd-party-components](http://www.opencascade.com/content/3rd-party-components). On Linux and OS X, it is recommended to use the version installed in the system natively.

You can also build third-party libraries from their sources:

- Building 3rd-party libraries on Windows
- Building 3rd-party libraries on Linux
- Building 3rd-party libraries on MacOS X

Build OCCT using your preferred build tool.

- Building with CMake (cross-platform)
- Building with CMake for Android (cross-platform)
- Building on Windows with MS Visual Studio projects
- Building on Mac OS X with Code::Blocks projects
- Building on Mac OS X with Xcode projects

The current version of OCCT can be consulted in the file `src/Standard/Standard_Version.hxx`
Building 3rd-party libraries on Windows

Table of Contents

- Introduction
- Building Mandatory Third-party Products
  - Tcl/Tk
    - Installation from sources: Tcl
    - Installation from sources: Tk
  - FreeType
- Building Optional Third-party Products
  - TBB
  - gl2ps
  - FreelImage
  - VTK
Introduction

This document presents guidelines for building third-party products used by Open CASCADE Technology (OCCT) and samples on Windows platform. It is assumed that you are already familiar with MS Visual Studio / Visual C++.

You need to use the same version of MS Visual Studio for building all third-party products and OCCT itself, in order to receive a consistent set of run-time binaries.

The links for downloading the third-party products are available on the web site of OPEN CASCADE SAS at http://www.opencascade.com/content/3rd-party-components.

There are two types of third-party products used by OCCT:

- Mandatory products:
  - Tcl/Tk 8.5 – 8.6;
  - FreeType 2.4.10 – 2.5.3.

- Optional products:
  - TBB 3.x – 4.x;
  - gl2ps 1.3.5 – 1.3.8;
  - FreImage 3.14.1 – 3.16.0;
  - VTK 6.1.0.

It is recommended to create a separate new folder on your workstation, where you will unpack the downloaded archives of the third-party products, and where you will build these products (for example, c:\occ3rdparty).

Further in this document, this folder is referred to as 3rdparty.
Building Mandatory Third-party Products

Tcl/Tk

Tcl/Tk is required for DRAW test harness.

Installation from sources: Tcl

Download the necessary archive from http://www.tcl.tk/software/tcltk/download.html and unpack it.

1. In the win sub-directory, edit file buildall.vc.bat:
   ○ Edit the line "call ... vcvars32.bat" to have correct path to the version of Visual Studio to be used for building, for instance:

```
call "%VS80COMNTOOLS%\vsvars32.bat"
```

If you are building 64-bit version, set environment accordingly, e.g.:
```
call "%VS80COMNTOOLS%\..\..\VC\vcvarsall.bat"
```

   ○ Define variable INSTALLDIR pointing to directory where Tcl/Tk will be installed, e.g.:

```
set INSTALLDIR=D:\OCCT\3rdparty\tcltk-86-32
```

   ○ Add option install to the first command line calling nmake:

```
nmake -nologo -f makefile.vc release htmlhelp install %1
```

   ○ Remove second call to nmake (building statically linked executable)

2. Edit file rules.vc replacing line

```
SUFX      = tsgx
```
by

```
SUFX = sgx
```

This is to avoid extra prefix 't' in the library name, which is not recognized by default by OCCT build tools.

3. By default, Tcl uses dynamic version of run-time library (MSVCRT), which must be installed on the system where Tcl will be used. You may wish to link Tcl library with static version of run-time to avoid this dependency. For that:
   - Edit file `makefile.vc` replacing strings "crt = -MD" by "crt = -MT"
   - Edit source file `tclMain.c` (located in folder `generic`) commenting out forward declaration of function `isatty()`.

4. In the command prompt, run `buildall.vc.bat`

You might need to run this script twice to have `tclsh` executable installed; check subfolder `bin` of specified installation path to verify this.

5. For convenience of use, we recommend making a copy of `tclsh` executable created in subfolder `bin` of `INSTALLDIR` and named with Tcl version number suffix, as `tclsh.exe` (with no suffix)

```
> cd D:\OCCT\3rdparty\tcltk-86-32\bin
> cp tclsh86.exe tclsh.exe
```

**Installation from sources: Tk**

Download the necessary archive from [http://www.tcl.tk/software/tcltk/download.html](http://www.tcl.tk/software/tcltk/download.html) and unpack it.

Apply the same steps as described for building Tcl above, with the same `INSTALLDIR`. Note that Tk produces its own executable, called `wish`.

You might need to edit default value of `TCLDIR` variable defined in `buildall.vc.bat` (should be not necessary if you unpack both Tcl and Tk sources in the same folder).
FreeType

FreeType is required for text display in a 3D viewer. You can download its sources from http://sourceforge.net/projects/freetype/files/

The building procedure

1. Unpack the downloaded archive of FreeType product into the 3rdparty folder. As a result, you will get a folder named, for example, 3rdparty\freetype-2.4.10. Further in this document, this folder is referred to as freetype.
2. Open the solution file freetype\builds\win32\vc20xx\freetype.sln in Visual Studio. Here vc20xx stands for your version of Visual Studio.
3. Select the configuration to build: either Debug or Release.
4. Build the freetype project.

As a result, you will get a freetype import library (.lib) in the freetype\obj\win32\vc20xx folder.

5. If you build FreeType for a 64 bit platform, select in the main menu Build - Configuration Manager and add x64 platform to the solution configuration by copying the settings from Win32 platform:
Update the value of the Output File for x64 configuration:

![Freetype Property Pages](image)

Build the freetype project.

As a result, you will obtain a 64 bit import library (.lib) file in the freetype\x64\vc20xx folder.

To build FreeType as a dynamic library (.dll) follow steps 6, 7 and 8 of this procedure.

6. Open menu Project-> Properties-> Configuration Properties-> General and change option **Configuration Type** to *Dynamic Library (.dll)*.

7. Edit file freetype\include\freetype\config\ftoption.h:

   in line 255, uncomment the definition of macro *FT_EXPORT* and change it as follows:

   ```
   #define FT_EXPORT(x) __declspec(dllexport) x
   ```

8. Build the freetype project.
As a result, you will obtain the files of the import library (.lib) and the dynamic library (.dll) in folders `freetype \objs\release` or `\objs\debug`.

If you build for a 64 bit platform, follow step 5 of the procedure.

To facilitate the use of FreeType libraries in OCCT with minimal adjustment of build procedures, it is recommended to copy the include files and libraries of FreeType into a separate folder, named according to the pattern: `freetype-compiler-bitness-building mode`, where:

- **compiler** is `vc8` or `vc9` or `vc10` or `vc11`;
- **bitness** is 32 or 64;
- **building mode** is *opt* (for Release) or *deb* (for Debug).

The `include` subfolder should be copied as is, while libraries should be renamed to `freetype.lib` and `freetype.dll` (suffixes removed) and placed to subdirectories `lib *and *bin`, respectively. If the Debug configuration is built, the Debug libraries should be put into subdirectories `libd and bind`. 
Building Optional Third-party Products

TBB

This third-party product is installed with binaries from the archive that can be downloaded from http://threadingbuildingblocks.org/. Go to the Download page, find the release version you need (e.g. tbb30_018oss) and pick the archive for Windows platform.

Unpack the downloaded archive of TBB product into the 3rdparty folder.

Further in this document, this folder is referred to as tbb.
gl2ps

This third-party product should be built as a dynamically loadable library (dll file). You can download its sources from http://geuz.org/gl2ps/src/.

The building procedure

1. Unpack the downloaded archive of gl2ps product (e.g. gl2ps-1.3.5.tgz) into the 3rdparty folder.

As a result, you will get a folder named, for example, 3rdparty\gl2ps-1.3.5-source.

Rename it into gl2ps-platform-compiler-building mode, where

- **platform** – win32 or win64;
- **compiler** – vc8, vc9 or vc10;
- **building mode** – opt (for release) or deb (for debug).

For example, gl2ps-win64-vc10-deb

Further in this document, this folder is referred to as gl2ps.

2. Download (from http://www.cmake.org/cmake/resources/software.html) and install the CMake build system.

3. Edit the file gl2ps\CMakeLists.txt.

After line 113 in CMakeLists.txt:

```cmake
set_target_properties(shared PROPERTIES COMPIL
_FLAGS "$\"-DGL2PSDLL -DGL2PSDLL_EXPORTS\""
```

add the following line:

```cmake
add_definitions(-D_USE_MATH_DEFINES)
```

Attention: If Cygwin was installed on your computer, make sure that there is no path to it in the PATH variable to avoid possible conflicts during the configuration.
4. Launch CMake (cmake-gui.exe) using the Program menu.

In CMake:

- Define where the source code is. This path must point to gl2ps folder.
- Define where to build the binaries. This path must point to the folder where generated gl2ps project binaries will be placed (for example, gl2ps\bin). Further in this document, this folder is referred to as gl2ps_bin.
- Press Configure button.

- Select the generator (the compiler and the target platform – 32 or 64 bit) in the pop-up window.
- Press **Finish** button to return to the main CMake window. Expand the ENABLE group and uncheck ENABLE_PNG and ENABLE_ZLIB check boxes.

- Expand the CMAKE group and define `CMAKE_INSTALL_PREFIX` which is the path where you want to install the build results, for example, `c:\occ3rdparty\gl2ps-1.3.5`. 
Press **Configure** button again, then press **Generate** button to generate Visual Studio projects. After completion, close CMake application.

5. Open the solution file `gl2ps_bin\gl2ps.sln` in Visual Studio.
   - Select a configuration to build
     - Choose **Release** to build Release binaries.
     - Choose **Debug** to build Debug binaries.
   - Select a platform to build.
     - Choose **Win32** to build for a 32 bit platform.
     - Choose **x64** to build for a 64 bit platform.
   - Build the solution.
   - Build the **INSTALL** project.

As a result, you should have the installed gl2ps product in the **CMAKE_INSTALL_PREFIX** path.
**FreelImage**

This third-party product should be built as a dynamically loadable library (.dll file). You can download its sources from http://sourceforge.net/projects/freeimage/files/Source%20Distribution/

**The building procedure:**

1. Unpack the downloaded archive of FreelImage product into 3rdparty folder.

   As a result, you should have a folder named 3rdparty\FreelImage.

   Rename it according to the rule: *freeimage-platform-compiler-building mode*, where

   - **platform** is *win32* or *win64*;
   - **compiler** is *vc8* or *vc9* or *vc10* or *vc11*;
   - **building mode** is *opt* (for release) or *deb* (for debug)

   Further in this document, this folder is referred to as *freeimage*.

2. Open the solution file *freeimage\FreelImage..sln* in your Visual Studio.

   If you use a Visual Studio version higher than VC++ 2008, apply conversion of the workspace. Such conversion should be suggested automatically by Visual Studio.

3. Select a configuration to build.

   - Choose **Release** if you are building Release binaries.
   - Choose **Debug** if you are building Debug binaries.

   *Note:*

   If you want to build a debug version of FreelImage binaries then you need to rename the following files in FreelImage and FreelimagePlus projects:

   - Project -> Properties -> Configuration Properties -> Linker ->
General -> Output File

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeImage<em>d</em>.dll</td>
<td>FreeImage.dll</td>
</tr>
<tr>
<td>FreeImagePlus<em>d</em>.dll</td>
<td>FreeImagePlus.dll</td>
</tr>
</tbody>
</table>

Project -> Properties -> Configuration Properties -> Linker -> Debugging -> Generate Program Database File

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeImage<em>d</em>.pdb</td>
<td>FreeImage.pdb</td>
</tr>
<tr>
<td>FreeImagePlus<em>d</em>.pdb</td>
<td>FreeImagePlus.pdb</td>
</tr>
</tbody>
</table>

Project -> Properties -> Configuration Properties -> Linker -> Advanced-Import Library

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeImage<em>d</em>.lib</td>
<td>FreeImage.lib</td>
</tr>
<tr>
<td>FreeImagePlus<em>d</em>.lib</td>
<td>FreeImagePlus.lib</td>
</tr>
</tbody>
</table>

Project -> Properties -> Configuration Properties -> Build Events -> Post -> Build Event -> Command Line

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeImage<em>d</em>.dll</td>
<td>FreeImage.dll</td>
</tr>
<tr>
<td>FreeImage* d*.lib</td>
<td>FreeImage.lib</td>
</tr>
<tr>
<td>FreeImagePlus<em>d</em>.dll</td>
<td>FreeImagePlus.dll</td>
</tr>
<tr>
<td>FreeImagePlus<em>d</em>.lib</td>
<td>FreeImagePlus.lib</td>
</tr>
</tbody>
</table>

Additionally, rename in project FreeImagePlus

Project -> Properties -> Configuration Properties -> Linker -> Input -> Additional Dependencies

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>from FreeImage<em>d</em>.lib</td>
<td>to FreeImage.lib</td>
</tr>
</tbody>
</table>

4. Select a platform to build.
   - Choose Win32 if you are building for a 32 bit platform.
   - Choose x64 if you are building for a 64 bit platform.

5. Start the building process.

As a result, you should have the library files of FreeImage product in freeimage\Dist folder (FreeImage.dll and FreeImage.lib) and in freeimage\Wrapper\FreeImagePlus\dist folder (FreeImagePlus.dll and FreeImagePlus.lib).
VTK

VTK is an open-source, freely available software system for 3D computer graphics, image processing and visualization. VTK Integration Services component provides adaptation functionality for visualization of OCCT topological shapes by means of VTK library.

The building procedure:

1. Download the necessary archive from http://www.vtk.org/VTK/resources/software.html and unpack it into 3rdparty folder.
   
   As a result, you will get a folder named, for example, 3rdparty\VTK-6.1.0.
   
   Further in this document, this folder is referred to as VTK.

2. Use CMake to generate VS projects for building the library:
   ○ Start CMake-GUI and select VTK folder as source path, and the folder of your choice for VS project and intermediate build data.
   ○ Click Configure.
   ○ Select the VS version to be used from the ones you have installed (we recommend using VS 2010) and the architecture (32 or 64-bit).
   ○ Generate VS projects with default CMake options. The open solution VTK.sln will be generated in the build folder.

3. Build project VTK in Release mode.
Building 3rd-party libraries on Linux

Table of Contents

- Introduction
- Building Mandatory Third-party Products
  - Tcl/Tk
    - Installation from sources: Tcl
    - Installation from sources: Tk
  - FreeType
- Building Optional Third-party Products
  - TBB
  - gl2ps
  - FreImage
  - VTK
- Installation From Official Repositories
  - Debian-based distributives
Introduction

This document presents additional guidelines for building third-party products used by Open CASCADE Technology and samples on Linux platform.

The links for downloading the third-party products are available on the web site of OPEN CASCADE SAS at http://www.opencascade.com/content/3rd-party-components.

There are two types of third-party products, which are necessary to build OCCT:

- Mandatory products:
  - Tcl/Tk 8.5 - 8.6;
  - FreeType 2.4.10 - 2.5.3;
- Optional products:
  - TBB 3.x - 4.x;
  - gl2ps 1.3.5 - 1.3.8;
  - FreImage 3.14.1 - 3.16.0;
  - VTK 6.1.0.
Building Mandatory Third-party Products

Tcl/Tk

Tcl/Tk is required for DRAW test harness.

Installation from sources: Tcl

Download the necessary archive from http://www.tcl.tk/software/tcltk/download.html and unpack it.

1. Enter the unix sub-directory of the directory where the Tcl source files are located (TCL_SRC_DIR).
   
   ```
   cd TCL_SRC_DIR/unix
   ```

2. Run the `configure` command:
   
   ```
   configure --enable-gcc --enable-shared --enable-threads --prefix=TCL_INSTALL_DIR
   ```

   For a 64 bit platform also add `--enable-64bit` option to the command line.

3. If the configure command has finished successfully, start the building process:
   
   ```
   make
   ```

4. If building is finished successfully, start the installation of Tcl. All binary and service files of the product will be copied to the directory defined by `TCL_INSTALL_DIR`
   
   ```
   make install
   ```

Installation from sources: Tk

Download the necessary archive from
http://www.tcl.tk/software/tcltk/download.html and unpack it.

1. Enter the unix sub-directory of the directory where the Tk source files are located (TK_SRC_DIR)
   
   ```bash
   cd TK_SRC_DIR/unix
   ```

2. Run the configure command, where TCL_LIB_DIR is TCL_INSTALL_DIR/lib.
   
   ```bash
   configure --enable-gcc --enable-shared --enable-threads --with-tcl=TCL_LIB_DIR --prefix=TK_INSTALL_DIR
   ```

   For a 64 bit platform also add `--enable-64bit` option to the command line.

3. If the configure command has finished successfully, start the building process:
   
   ```bash
   make
   ```

4. If the building has finished successfully, start the installation of Tk. All binary and service files of the product will be copied to the directory defined by TK_INSTALL_DIR (usually it is TCL_INSTALL_DIR)
   
   ```bash
   make install
   ```
FreeType

FreeType is required for text display in the 3D viewer. Download the necessary archive from http://sourceforge.net/projects/freetype/files/ and unpack it.

1. Enter the directory where the source files of FreeType are located (FREETYPE_SRC_DIR).

   cd FREETYPE_SRC_DIR

2. Run the configure command:

   configure --prefix=FREETYPE_INSTALL_DIR

   For a 64 bit platform also add CFLAGS='--m64 -fPIC' CPPFLAGS='--m64 -fPIC' option to the command line.

3. If the configure command has finished successfully, start the building process:

   make

4. If the building has finished successfully, start the installation of FreeType. All binary and service files of the product will be copied to the directory defined by FREETYPE_INSTALL_DIR

   make install
Building Optional Third-party Products

TBB

This third-party product is installed with binaries from the archive that can be downloaded from http://threadingbuildingblocks.org. Go to the Download page, find the release version you need and pick the archive for Linux platform. To install, unpack the downloaded archive of TBB product.
gl2ps

Download the necessary archive from http://geuz.org/gl2ps/ and unpack it.

1. Install or build cmake product from the source file.
2. Start cmake in GUI mode with the directory where the source files of gl2ps are located:

   ```
   ccmake GL2PS_SRC_DIR
   ```
   - Press [c] to make the initial configuration;
   - Define the necessary options in `CMAKE_INSTALL_PREFIX`
   - Press [c] to make the final configuration
   - Press [g] to generate Makefile and exit

   or just run the following command:

   ```
   cmake -DCMAKE_INSTALL_PREFIX=GL2PS_INSTALL_DIR
   -DCMAKE_BUILD_TYPE=Release
   ```

3. Start the building of gl2ps:

   ```
   make
   ```

4. Start the installation of gl2ps. Binaries will be installed according to the `CMAKE_INSTALL_PREFIX` option.

   ```
   make install
   ```
**FreImage**

Download the necessary archive from [http://sourceforge.net/projects/freeimage/files/Source%20Distribution/](http://sourceforge.net/projects/freeimage/files/Source%20Distribution/) and unpack it. The directory with unpacked sources is further referred to as `FREEIMAGE_SRC_DIR`.

1. Modify

   ```c
   #include string.h
   ```

2. Enter the directory where the source files of FreImage are located (`FREEIMAGE_SRC_DIR`).
   ```bash
cd FREEIMAGE_SRC_DIR
   ```

3. Run the building process
   ```bash
   make
   ```

4. Run the installation process
   a. If you have the permission to write into directories `/usr/include` and `/usr/lib`, run the following command:
      ```bash
      make install
      ```
   b. If you do not have this permission, you need to modify file `FREEIMAGE_SRC_DIR/Makefile.gnu`:
      Change lines 7-9 from:
      ```make
      DESTDIR ?= /
      INCDIR  ?= $(DESTDIR)/usr/include
      INSTALLDIR ?= $(DESTDIR)/usr/lib
      ```
      to:
      ```make
      DESTDIR  ?= $(DESTDIR)
      INCDIR   ?= $(DESTDIR)/include
      ```
INSTALLDIR ?= $(DESTDIR)/lib

Change lines 65-67 from:

```
install  -m 644 -o root -g root $(HEADER) $(INCDIR)
install  -m 644 -o root -g root $(STATICLIB) $(INSTALLDIR)
install  -m 755 -o root -g root $(SHAREDLIB) $(INSTALLDIR)
```

to:

```
install  -m 755 $(HEADER) $(INCDIR)
install  -m 755 $(STATICLIB) $(INSTALLDIR)
install  -m 755 $(SHAREDLIB) $(INSTALLDIR)
```

Change line 70 from:

```
ldconfig
```

to:

```
#ldconfig
```

Then run the installation process by the following command:

```
make DESTDIR=FREEIMAGE_INSTALL_DIR install
```

5. Clean temporary files

```
make clean
```
VTK

You can download VTK sources from http://www.vtk.org/VTK/resources/software.html

The building procedure:

Download the necessary archive from http://www.vtk.org/VTK/resources/software.html and unpack it.

1. Install or build cmake product from the source file.
2. Start cmake in GUI mode with the directory where the source files of VTK are located:

```
cmake VTK_SRC_DIR
```

   ○ Press [c] to make the initial configuration
   ○ Define the necessary options in VTK_INSTALL_PREFIX
   ○ Press [c] to make the final configuration
   ○ Press [g] to generate Makefile and exit

3. Start the building of VTK:

```
make
```

4. Start the installation of gl2ps. Binaries will be installed according to the VTK_INSTALL_PREFIX option.

```
make install
```
Installation From Official Repositories

Debian-based distributives

All 3rd-party products required for building of OCCT could be installed from official repositories. You may install them from console using apt-get utility:

```
sudo apt-get install tcllib tklib tcl-dev tk-dev libfreetype-dev libxt-dev libxmu-dev libxi-dev libgl1-mesa-dev libglu1-mesa-dev libfreeimage-dev libtbb-dev libgl2ps-dev
```

To launch binaries built with WOK you need to install C shell and 32-bit libraries on x86_64 distributives:

```
# you may need to add i386 if not done already by command "dpkg --add-architecture i386"
sudo apt-get install csh libstdc++6:i386 libxt6:i386 libxext6:i386 libxmu6:i386
```

Building is possible with C++ compliant compiler:

```
sudo apt-get install g++
```
Building 3rd-party libraries on MacOS X

Table of Contents

Introduction

Building Mandatory Third-party Products

Tcl/Tk 8.5

Installation from sources: Tcl 8.5

Installation from sources: Tk 8.5

FreeType 2.4.10

Building Optional Third-party Products

TBB 3.x or 4.x

gl2ps 1.3.5

FreelImage 3.14.1 or 3.15.x
Introduction

This document presents additional guidelines for building third-party products used by Open CASCADE Technology and samples on Mac OS X platform (10.6.4 and later).

The links for downloading the third-party products are available on the web site of OPEN CASCADE SAS at http://www.opencascade.com/content/3rd-party-components.

There are two types of third-party products, which are necessary to build OCCT:

- Mandatory products:
  - Tcl/Tk 8.5 - 8.6;
  - FreeType 2.4.10 - 2.5.3.
- Optional products:
  - TBB 3.x - 4.x;
  - gl2ps 1.3.5 - 1.3.8;
  - FreelImage 3.14.1 - 3.16.0
Building Mandatory Third-party Products

Tcl/Tk 8.5

Tcl/Tk is required for DRAW test harness. Version 8.5 or 8.6 can be used with OCCT.

Installation from sources: Tcl 8.5

Download the necessary archive from http://www.tcl.tk/software/tcltk/download.html and unpack it.

1. Enter the `macosx` sub-directory of the directory where the Tcl source files are located (\textit{TCL\_SRC\_DIR}).

   \begin{verbatim}
   cd TCL\_SRC\_DIR/macosx
   \end{verbatim}

2. Run the \textit{configure} command

   \begin{verbatim}
   configure --enable-gcc --enable-shared --enable-threads --prefix=TCL\_INSTALL\_DIR
   \end{verbatim}

   For a 64 bit platform also add \textit{--enable-64bit} option to the command line.

3. If the \textit{configure} command has finished successfully, start the building process

   \begin{verbatim}
   make
   \end{verbatim}

4. If building is finished successfully, start the installation of Tcl. All binary and service files of the product will be copied to the directory defined by \textit{TCL\_INSTALL\_DIR}.

   \begin{verbatim}
   make install
   \end{verbatim}

Installation from sources: Tk 8.5
Download the necessary archive from http://www.tcl.tk/software/tcltk/download.html and unpack it.

1. Enter the `macosx` sub-directory of the directory where the source files of Tk are located (`TK_SRC_DIR`).
   ```sh
cd TK_SRC_DIR/macosx
   ```

2. Run the `configure` command, where TCL_LIB_DIR is `TCL_INSTALL_DIR/lib`
   ```sh
   configure --enable-gcc --enable-shared --enable-threads --with-tcl=TCL_LIB_DIR --prefix=TK_INSTALL_DIR
   ```
   For a 64 bit platform also add `--enable-64bit` option to the command line.

3. If the `configure` command has finished successfully, start the building process:
   ```sh
   make
   ```

4. If the building has finished successfully, start the installation of Tk. All binary and service files of the product will be copied to the directory defined by `TK_INSTALL_DIR` (usually it is `TCL_INSTALL_DIR`)
   ```sh
   make install
   ```
FreeType 2.4.10

FreeType is required for text display in the 3D viewer.

Download the necessary archive from http://sourceforge.net/projects/freetype/files/ and unpack it.

1. Enter the directory where the source files of FreeType are located (FREETYPE_SRC_DIR).
   
   ```
   cd FREETYPE_SRC_DIR
   ```

2. Run the `configure` command
   
   ```
   configure --prefix=FREETYPE_INSTALL_DIR
   ```
   For a 64 bit platform also add `CFLAGS=-m64 -fPIC' `CPPFLAGS=-m64 -fPIC'` option to the command line.

3. If the `configure` command has finished successfully, start the building process
   
   ```
   make
   ```

4. If building has finished successfully, start the installation of FreeType. All binary and service files of the product will be copied to the directory defined by `FREETYPE_INSTALL_DIR`.
   
   ```
   make install
   ```
Building Optional Third-party Products

TBB 3.x or 4.x

This third-party product is installed with binaries from the archive that can be downloaded from http://threadingbuildingblocks.org/. Go to the Download page, find the release version you need (e.g. tbb30_018oss) and pick the archive for Mac OS X platform. To install, unpack the downloaded archive of TBB 3.0 product (tbb30_018oss_osx.tgz).
gl2ps 1.3.5

Download the necessary archive from http://geuz.org/gl2ps/ and unpack it.

1. Install or build cmake product from the source file.
2. Start cmake in GUI mode with the directory, where the source files of fl2ps are located:

   ```
   ccmake  GL2PS_SRC_DIR
   ```

   - Press [c] to make the initial configuration;
   - Define the necessary options in `CMAKE_INSTALL_PREFIX`;
   - Press [c] to make the final configuration;
   - Press [g] to generate Makefile and exit.

   or just run the following command:

   ```
   cmake  -DCMAKE_INSTALL_PREFIX=GL2PS_INSTALL_DIR
          -DCMAKE_BUILD_TYPE=Release
   ```

3. Start the building of gl2ps

   ```
   make
   ```

4. Start the installation of gl2ps. Binaries will be installed according to the `CMAKE_INSTALL_PREFIX` option

   ```
   make install
   ```
Freelimage 3.14.1 or 3.15.x

Download the necessary archive from http://sourceforge.net/projects/freeimage/files/Source%20Distribution/ and unpack it. The directory with unpacked sources is further referred to as FREEIMAGE_SRC_DIR.

Note that for building Freelimage on Mac OS X 10.7 you should replace Makefile.osx in FREEIMAGE_SRC_DIR by the corrected file, which you can find in attachment to issue #22811 in OCCT Mantis bug tracker (http://tracker.dev.opencascade.org/file_download.php?file_id=6937&type=bug).

1. If you build Freelimage 3.15.x you can skip this step. Modify FREEIMAGE_SRC_DIR/Source/OpenEXR/Imath/ImathMatrix.h:

   In line 60 insert the following:

   ```
   #include string.h
   ```

   Modify FREEIMAGE_SRC_DIR/Source/Freelmage/PluginTARGA.cpp:

   In line 320 replace:

   ```
   SwapShort(value);
   ```

   with:

   ```
   SwapShort(&value);
   ```

2. Enter the directory where the source files of Freelimage are located (FREEIMAGE_SRC_DIR).

   ```
   cd FREEIMAGE_SRC_DIR
   ```

3. Run the building process

   ```
   make
   ```

4. Run the installation process

   1. If you have the permission to write into /usr/local/include and
/usr/local/lib directories, run the following command:

```
make install
```

2. If you do not have this permission, you need to modify file `FREEIMAGE_SRC_DIR/Makefile.osx`:

Change line 49 from:

```
PREFIX ?= /usr/local
```

to:

```
PREFIX ?= $(PREFIX)
```

Change lines 65-69 from:

```
install -d -m 755 -o root -g wheel $(INCDIR) $(INSTALLDIR)
install  -m 644 -o root -g wheel $(HEADER) $(INCDIR)
install  -m 644 -o root -g wheel $(SHAREDLIB) $(STATICLIB) $(INSTALLDIR)
ranlib  -sf $(INSTALLDIR)/$(STATICLIB)
ln     -sf $(SHAREDLIB) $(INSTALLDIR)/$(VERLIBNAME)
```


to:

```
install  -d $(INCDIR) $(INSTALLDIR)
install  -m 755 $(HEADER) $(INCDIR)
install  -m 755 $(STATICLIB) $(INSTALLDIR)
install  -m 755 $(SHAREDLIB) $(STATICLIB) $(INSTALLDIR)
ln     -sf $(SHAREDLIB) $(INSTALLDIR)/$(VERLIBNAME)
ln     -sf $(VERLIBNAME) $(INSTALLDIR)/$(LIBNAME)
```

Then run the installation process by the following command:

```
make PREFIX=FREEIMAGE_INSTALL_DIR install
```
5. Clean temporary files

make clean
Open CASCADE Technology 7.2.0

Building with CMake

Table of Contents

- General
- Start CMake
- Configuration process
- 3rd party search mechanism
- Projects generation
- Building
- Installation
General

This article describes the CMake-based build process, which is now suggested as a standard way to produce the binaries of Open CASCADE Technology from sources. *OCCT requires CMake version 2.8.12 or later.*

**Note**

Compared to the previous (6.x) releases of Open CASCADE Technology, OCCT 7.x has a complete set of CMake scripts and projects, so that there is no need to use WOK anymore. Moreover, CMake gives you a powerful configuration tool, which allows to control many aspects of OCCT deployment. At the same time this tool is quite intuitive, which is a significant advantage over the legacy WOK utilities.

Here we describe the build procedure on the example of Windows platform with Visual Studio 2010. However, CMake is cross-platform and can be used to build OCCT on Linux and OS X in essentially the same way.

**Note**

Before you start, make sure to have installed all 3-rd party products that you are going to use with OCCT; see *Building OCCT from sources.*
Start CMake

CMake is a tool that generates the actual project files for the selected target build system (e.g. Unix makefiles) or IDE (e.g. Visual Studio 2010).

For unexperienced users we recommend to start with cmake-gui – a cross-platform GUI tool provided by CMake on Windows, Mac and Linux. A command-line alternative, ccmake can also be used.

CMake deals with three directories: source, build or binary and installation.

- The source directory is where the sources of OCCT are located in your file system;
- The build or binary directory is where all files created during CMake configuration and generation process will be located. The mentioned process will be described below.
- The installation directory is where binaries will be installed after building the INSTALL project that is created by CMake generation process, along with header files and resources required for OCCT use in applications.

The good practice is not to use the source directory as a build one. Different configurations should be built in different build directories to avoid conflicts. It is however possible to choose one installation directory for several configurations of OCCT (differentiated by platform, bitness, compiler and build type), for example:

```
d:/occt/  -- the source directory
```
```
d:/tmp/occt-build-vc10-x64  -- the build directory with the generated solution and other intermediate files created during a CMake tool working
```
```
d:/occt-install  -- the installation directory that is able to contain several OCCT configurations
```

Configuration process

If the command-line tool is used, run the tool from the build directory with a single argument indicating the source (relative or absolute path) directory:

```
cd d:/tmp/occt-build-vc10-x64
cmake d:/occt
```

Press `c` to configure.

All actions required in the configuration process with the GUI tool will be described below.

If the GUI tool is used, run this tool without additional arguments and after that specify the source directory by clicking **Browse Source** and the build (binary) one by clicking **Browse Build**.
Note: Each configuration of the project should be built in its own directory. When building multiple configurations it is recommended to indicate in the name of build directories the system, bitness and compiler (e.g., `d:/occt/build/win32-vc10`).

Once the source and build directories are selected, "Configure" button should be pressed in order to start manual configuration process. It begins with selection of a target configurator. It is "Visual Studio 10 2010 Win64" in our example.
To build OCCT for **Universal Windows Platform (UWP)** specify the path to toolchain file for cross-compiling
\textit{d:/occt/adm/templates/uwp.toolchain.config.cmake}.

Alternatively, if you are using CMake from the command line add options
\texttt{-DCMAKE\_SYSTEM\_NAME=WindowsStore -DCMAKE\_SYSTEM\_VERSION=10.0}.

**Note:** Universal Windows Platform (UWP) is supported only on "Visual Studio 14 2015". File \texttt{d:/occt/samples/xaml/ReadMe.md} describes the building procedure of XAML (UWP) sample.

Once "Finish" button is pressed, the first pass of the configuration process is executed. At the end of the process, CMake outputs the list of environment variables, which have to be properly specified for successful configuration.
The error message provides some information about these variables. This message will appear after each pass of the process until all required variables are specified correctly.

The change of the state of some variables can lead to the appearance of new variables. The new variables appeared after the pass of the configuration process are highlighted with red color by CMake GUI tool.

Note: There is "grouped" option, which groups variables with a common prefix.

The following table gives the full list of environment variables used at the configuration stage:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMAKE_BUILD_TYPE</td>
<td>String</td>
<td>Specifies the build type for single-configuration generators (such as make). Possible values are Debug, Release, and RelWithDebInfo.</td>
</tr>
<tr>
<td>USE_FREEIMAGE</td>
<td>Boolean</td>
<td>Indicates whether the FreeImage product should be used in OCCT visualization module for support of...</td>
</tr>
<tr>
<td>Flag</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>USE_GL2PS</td>
<td>Boolean</td>
<td>Indicates whether GL2PS product should be used in OCCT visualization module for support of vector image formats (PS, EPS, etc.).</td>
</tr>
<tr>
<td>USE_TBB</td>
<td>Boolean</td>
<td>Indicates whether TBB 3rd party is used or not. TBB stands for Threading Building Blocks, the technology of Intel Corp, which comes with different mechanisms and patterns for injecting parallelism into your application. OCCT remains parallel even without TBB product.</td>
</tr>
<tr>
<td>USE_VTK</td>
<td>Boolean</td>
<td>Indicates whether VTK 3rd party is used or not. VTK stands for Visualization ToolKit, the technology of Kitware Inc intended for general-purpose scientific visualization. OCCT comes with a bridge between CAD data representation and VTK by means of its dedicated VIS component (VTK Integration Services). You may skip this 3rd party unless you are planning to use VTK visualization for OCCT geometry. See the official documentation (VIS) for the details.</td>
</tr>
<tr>
<td>3RDPARTY_DIR</td>
<td>Path</td>
<td>Defines the root directory where all required 3rd party products will be searched. Once you define this path it is very convenient to click &quot;Configure&quot; button in order to let CMake automatically detect all necessary products.</td>
</tr>
<tr>
<td>3RDPARTY_FREETYPE_*</td>
<td>Path</td>
<td>Path to Freetype binaries.</td>
</tr>
<tr>
<td>3RDPARTY_TCL_* 3RDPARTY_TK_*</td>
<td>Path</td>
<td>Path to Tcl/Tk binaries.</td>
</tr>
<tr>
<td>3RDPARTY_FREEIMAGE*</td>
<td>Path</td>
<td>Path to Freeimage binaries.</td>
</tr>
<tr>
<td>3RDPARTY_GL2PS_*</td>
<td>Path</td>
<td>Path to GL2PS binaries.</td>
</tr>
<tr>
<td>3RDPARTY_TBB*</td>
<td>Path</td>
<td>Path to TBB binaries.</td>
</tr>
<tr>
<td>3RDPARTY_VTK_*</td>
<td>Path</td>
<td>Path to VTK binaries.</td>
</tr>
<tr>
<td>BUILD_MODULE_&lt;MODULE&gt;</td>
<td>Boolean</td>
<td>Indicates whether the corresponding OCCT module should be built or not. It should be noted that some toolkits of a module can be built even if this module is not checked (this happens if ...</td>
</tr>
</tbody>
</table>

---

Popular graphics image formats (PNG, BMP, etc.)
<table>
<thead>
<tr>
<th>BUILD_LIBRARY_TYPE</th>
<th>String</th>
<th>Specifies the type of library to be created. &quot;Shared&quot; libraries are linked dynamically and loaded at runtime. &quot;Static&quot; libraries are archives of object files used when linking other targets.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD_ADDITIONAL_TOOLKITS</td>
<td>String</td>
<td>Semicolon-separated individual toolkits to include into build process. If you want to build some particular libraries only, you may uncheck all &quot;BUILD_MODULE_&lt;MODULE&gt;&quot; and provide the list of necessary libraries there. Of course, all dependencies will be resolved automatically.</td>
</tr>
<tr>
<td>BUILD_YACCLEX</td>
<td>Boolean flag</td>
<td>Enables Flex/Bison lexical analyzers. OCCT source files relating to STEP reader and ExprIntrp functionality are generated automatically with Flex/Bison. Checking this option leads to automatic search of Flex/Bison binaries and regeneration of the mentioned files.</td>
</tr>
<tr>
<td>BUILD_MODULE_MfcSamples</td>
<td>Boolean flag</td>
<td>Indicates whether MFC samples should be built together with OCCT. This option is only relevant to Windows platforms.</td>
</tr>
<tr>
<td>BUILD_Inspector</td>
<td>Boolean flag</td>
<td>Indicates whether Inspector should be built together with OCCT.</td>
</tr>
<tr>
<td>BUILD_DOC_Overview</td>
<td>Boolean flag</td>
<td>Indicates whether OCCT overview documentation project should be created together with OCCT. It is not built together with OCCT. Checking this option leads to automatic search of Doxygen binaries. Its building calls Doxygen command to generate the documentation in HTML format.</td>
</tr>
<tr>
<td>BUILD_PATCH</td>
<td>Path</td>
<td>Points to the directory recognized as a &quot;patch&quot; for OCCT. If specified, the files from this directory take precedence over the corresponding OCCT sources. This way you are able to introduce patches to Open CASCADE Technology not affecting the original source.</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BUILD_WITH_DEBUG</td>
<td>Boolean</td>
<td>Enables extended messages of many OCCT algorithms, usually printed to cout. These include messages on internal errors and special cases encountered, timing, etc.</td>
</tr>
<tr>
<td>BUILD_ENABLE_FPE_SIGNAL_HANDLER</td>
<td>Boolean</td>
<td>Enable/Disable the floating point exceptions (FPE) during DRAW execution only. Corresponding environment variable (CSF_FPE) can be changed manually in custom.bat/sh scripts without regeneration by CMake.</td>
</tr>
<tr>
<td>CMAKE_CONFIGURATION_TYPES</td>
<td>String</td>
<td>Semicolon-separated CMake configurations</td>
</tr>
<tr>
<td>INSTALL_DIR</td>
<td>Path</td>
<td>Points to the installation directory. It is a synonym of CMAKE_INSTALL_PREFIX. The user can specify INSTALL_DIR as INSTALL_DIR_BIN, INSTALL_DIR_SCRIPT, INSTALL_DIR_LIB, INSTALL_DIR_INCLUDE, INSTALL_DIR_RESOURCE, or INSTALL_DIR_LAYOUT.</td>
</tr>
<tr>
<td>INSTALL_DIR_BIN</td>
<td>Path</td>
<td>Relative path to the binaries installation directory (absolute path is $INSTALL_DIR/$INSTALL_DIR_BIN)</td>
</tr>
<tr>
<td>INSTALL_DIR_SCRIPT</td>
<td>Path</td>
<td>Relative path to the scripts installation directory (absolute path is $INSTALL_DIR/$INSTALL_DIR_SCRIPT)</td>
</tr>
<tr>
<td>INSTALL_DIR_LIB</td>
<td>Path</td>
<td>Relative path to the libraries installation directory (absolute path is $INSTALL_DIR/$INSTALL_DIR_LIB)</td>
</tr>
<tr>
<td>INSTALL_DIR_INCLUDE</td>
<td>Path</td>
<td>Relative path to the includes installation directory (absolute path is $INSTALL_DIR/$INSTALL_DIR_INCLUDE)</td>
</tr>
<tr>
<td>INSTALL_DIR_RESOURCE</td>
<td>Path</td>
<td>Relative path to the resources installation directory (absolute path is $INSTALL_DIR/$INSTALL_DIR_RESOURCE)</td>
</tr>
<tr>
<td>INSTALL_DIR_LAYOUT</td>
<td>String</td>
<td>Defines the structure of OCCT files (binaries, resources, headers, etc.) for the installation directory. Two variants are predefined: for Windows (standard OCCT layout) and for Unix operating systems (standard Linux layout). If needed, the layout can be customized with INSTALL_DIR_* variables.</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>INSTALL_DIR_DATA</td>
<td>Path</td>
<td>Relative path to the directory (absolute path is <code>${INSTALL_DIR}/${INSTALL_DIR_DATA}</code>)</td>
</tr>
<tr>
<td>INSTALL_DIR_SAMPLES</td>
<td>Path</td>
<td>Relative path to the samples installation directory. Note that only &quot;samples/tcl&quot; folder will be installed. (absolute path is <code>${INSTALL_DIR}/${INSTALL_DIR_SAMPLES}</code>)</td>
</tr>
<tr>
<td>INSTALL_DIR_TESTS</td>
<td>Path</td>
<td>Relative path to the tests installation directory (absolute path is <code>${INSTALL_DIR}/${INSTALL_DIR_TESTS}</code>)</td>
</tr>
<tr>
<td>INSTALL_DIR_DOC</td>
<td>Path</td>
<td>Relative path to the documentation installation directory (absolute path is <code>${INSTALL_DIR}/${INSTALL_DIR_DOC}</code>)</td>
</tr>
<tr>
<td>INSTALL_FREETYPE</td>
<td>Boolean flag</td>
<td>Indicates whether Freetype binaries should be installed into the installation directory</td>
</tr>
<tr>
<td>INSTALL_FREEIMAGE*</td>
<td>Boolean flag</td>
<td>Indicates whether Freeimage binaries should be installed into the installation directory</td>
</tr>
<tr>
<td>INSTALL_GL2PS</td>
<td>Boolean flag</td>
<td>Indicates whether GL2PS binaries should be installed into the installation directory</td>
</tr>
<tr>
<td>INSTALL_TBB</td>
<td>Boolean flag</td>
<td>Indicates whether TBB binaries should be installed into the installation directory</td>
</tr>
<tr>
<td>INSTALL_VTK</td>
<td>Boolean flag</td>
<td>Indicates whether VTK binaries should be installed into the installation directory</td>
</tr>
<tr>
<td>INSTALL_TCL</td>
<td>Boolean flag</td>
<td>Indicates whether TCL binaries should be installed into the installation directory</td>
</tr>
<tr>
<td>INSTALL_TEST_CASES</td>
<td>Boolean flag</td>
<td>Indicates whether non-regression OCCT test scripts should be installed into the installation directory</td>
</tr>
<tr>
<td>INSTALL_DOC_Overview</td>
<td>Boolean flag</td>
<td>Indicates whether OCCT overview documentation should be installed into the installation directory</td>
</tr>
</tbody>
</table>

**Note:** Only the forward slashes ("/"") are acceptable in the CMake options defining paths.
3rd party search mechanism

If 3RDPARTY_DIR directory is defined, then required 3rd party binaries are sought in it, and default system folders are ignored.

The procedure expects to find binary and header files of each 3rd party product in its own sub-directory: bin, lib and include.

The results of the search (achieved on the next pass of the configuration process) are recorded in the corresponding variables:

- 3RDPARTY_<PRODUCT>_DIR – path to the 3rdparty directory (with directory name) (e.g. D:/3rdparty/tcltk-86-32)
- 3RDPARTY_<PRODUCT>_LIBRARY_DIR – path to the directory containing a library (e.g. D:/3rdparty/tcltk-86-32/lib).
- 3RDPARTY_<PRODUCT>_INCLUDE_DIR – path to the directory containing a header file (e.g., D:/3rdparty/tcltk-86-32/include)
- 3RDPARTY_<PRODUCT>_DLL_DIR – path to the directory containing a shared library (e.g., D:/3rdparty/tcltk-86-32/bin) This variable is only relevant to Windows platforms.

Note: each library and include directory should be children of the product directory if the last one is defined.

The search process is as follows:

1. Common path: 3RDPARTY_DIR
2. Path to a particular 3rd-party library: 3RDPARTY_<PRODUCT>_DIR
3. Paths to headers and binaries:
   1. 3RDPARTY_<PRODUCT>_INCLUDE_DIR
   2. 3RDPARTY_<PRODUCT>_LIBRARY_DIR
   3. 3RDPARTY_<PRODUCT>_DLL_DIR

If a variable of any level is not defined (empty or <variable name>-NOTFOUND ) and the upper level variable is defined, the content of the non-defined variable will be sought at the next configuration step. If the search process at level 3 does not find the required files, it seeks in default places.
If a search result (include path, or library path, or dll path) does not meet your expectations, you can change 3RDPARTY_<PRODUCT>_DIR variable*, clear (if they are not empty) 3RDPARTY_<PRODUCT>_DLL_DIR, 3RDPARTY_<PRODUCT>_INCLUDE_DIR and 3RDPARTY_<PRODUCT>_LIBRARY_DIR variables (or clear one of them) and run the configuration process again.

At this time the search will be performed in the newly identified directory and the result will be recorded to corresponding variables (replace old value if it is necessary).

For example, 3RDPARTY_FREETYPE_DIR variable

```
d:/3rdparty/freetype-2.4.10
```

can be changed to

```
d:/3rdparty/freetype-2.5.3
```

During the configuration process the related variables (3RDPARTY_FREETYPE_DLL_DIR, 3RDPARTY_FREETYPE_INCLUDE_DIR and 3RDPARTY_FREETYPE_LIBRARY_DIR) will be filled with new found values.

**Note:** The names of searched libraries and header files are hard-coded. If there is the need to change their names, change appropriate cmake variables (edit CMakeCache.txt file or edit in cmake-gui in advance mode) without reconfiguration: 3RDPARTY_<PRODUCT>_INCLUDE for include, 3RDPARTY_<PRODUCT>_LIB for library and 3RDPARTY_<PRODUCT>_DLL for shared library.
Once the configuration process is done, the "Generate" button is used to prepare project files for the target IDE. In our exercise the Visual Studio solution will be automatically created in the build directory.
Building

Go to the build folder, start the Visual Studio solution OCCT.sln and build it by clicking Build -> Build Solution.

By default the build solution process skips the building of the INSTALL and Overview project.

When the building process is finished build:

- Overview project to generate OCCT overview documentation (if BUILD_DOC_Overview variable is checked)
- the INSTALL project to run the installation process

For this, right-click on the Overview/INSTALL project and select Project Only -> Build Only -> Overview/INSTALL in the solution explorer.
Installation

Installation is a process of extracting redistributable resources (binaries, include files etc) from the build directory into the installation one. The installation directory will be free of project files, intermediate object files and any other information related to the build routines.

Normally you use the installation directory of OCCT to link against your specific application.

The directory structure is as follows:

```
data -- data files for OCCT (brep, iges, stp)
doc  -- OCCT overview documentation in HTML format
inc  -- header files
samples -- samples
src  -- all required source files for OCC T
tests -- OCCT test suite
win32\vc10\bind -- binary files (installed 3rdpartie s and occt)
    \libd -- libraries (installed 3rdparties a nd occt)
```

**Note:** The above example is given for debug configuration. However, it is generally safe to use the same installation directory for the release build. In the latter case the contents of install directory will be enriched with subdirectories and files related to the release configuration. In particular, the binaries directory win64 will be expanded as follows:

```
\win32\vc10\bind
    \libd
    \bin
    \lib
```
If CMake installation flags are enabled for the 3rd party products (e.g. INSTALL_FREETYPE), then the corresponding binaries will be copied to the same bin(d) and lib(d) directories together with the native binaries of OCCT. Such organization of libraries can be especially helpful if your OCCT-based software does not use itself the 3rd parties of Open CASCADE Technology (thus, there is no sense to pack them into dedicated directories).

The installation folder contains the scripts to run DRAWEXE (draw.bat or draw.sh), samples (if they were installed) and overview.html (short-cut for installed OCCT overview documentation).
Building with CMake for Android

This article describes the steps to build OCCT libraries for Android from a complete source package with GNU make (makefiles) on Windows 7 and Ubuntu 15.10.

The steps on Windows and Ubuntu are similar. There is the only one difference: makefiles are built with mingw32-make on Windows and native GNU make on Ubuntu.

Required tools (download and install if it is required):

- CMake v3.7+ [http://www.cmake.org/cmake/resources/software.html](http://www.cmake.org/cmake/resources/software.html)
Prerequisites

In toolchain file $CASROOT/adm/templates/android.toolchain.config.cmake:

- Set CMAKE_ANDROID_NDK variable equal to your Android NDK path.
- Set CMAKE_ANDROID_STL_TYPE variable to specify which C++ standard library to use.

The default value of CMAKE_ANDROID_STL_TYPE is gnustl_shared (GNU libstdc++ Shared)
Generation of makefiles using CMake GUI tool

Run GUI tool provided by CMake: cmake-gui

Tools configuration

- Specify the root folder of OCCT ($CASROOT, which contains CMakelists.txt file) by clicking Browse Source.
- Specify the location (build folder) for Cmake generated project files by clicking Browse Build.

Click Configure button. It opens the window with a drop-down list of generators supported by CMake project.

Select "MinGW Makefiles" item from the list

- Choose "Specify toolchain file for cross-compiling"
• Click "Next"

• Specify a toolchain file at the next dialog by
  `android.toolchain.config.cmake` . It is contained by cross-compilation toolchain for CMake
• Click "Finish"
If on Windows the message is appeared: "CMake Error: CMake was unable to find a build program corresponding to "MinGW Makefiles" CMAKE_MAKE_PROGRAM is not set. You probably need to select a different build tool.", specify `CMAKE_MAKE_PROGRAM` to mingw32-make executable.

OCCT Configuration

How to configure OCCT, see "OCCT Configuration" section of Building with CMake

Generation of makefiles

Click Generate button and wait until the generation process is finished. Then makefiles will appear in the build folder (e.g. `D:/occt/build-android`).
Generation of makefiles using CMake from the command line

Alternatively one may specify the values without a toolchain file:

```bash
cmake -G "MinGW Makefiles" -
DCMAKE_SYSTEM_NAME=Android -
DCMAKE_ANDROID_NDK=D:/DevTools/android-ndk-r13b -
DCMAKE_ANDROID_STL_TYPE=gnustl_shared -
DCMAKE_SYSTEM_VERSION=15 -
DCMAKE_ANDROID_ARCH_ABI=armeabi-v7a -
DCMAKE_MAKE_PROGRAM=D:/DevTools/MinGW/bin/mingw32-make.exe -
D3RDPARTY_DIR=D:/occt-3rdparty D:/occt
```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

D:\D:
D:\\occt D:\\build-android

D:\\\\build-android\cmake -G "MinGW Makefiles" -Dcmake_SYSTEM_NAME=Android -Dcmake_ANDROID_NDK=D:\DevTools/android-ndk-r12b -Dcmake_ANDROID_STL_TYPE=gnustl_shared -Dcmake_SYSTEM_VERSION=15 -Dcmake_ANDROID_ARCH_ABI=armeabi-v7a -Dcmake CMAKE_PROGRAM:D:\DevTools\MinGW\bin\mingw32-make.exe -Dthird PARTY_DIR=D:\occt\3rdparty D:\occt

- Android: Targeting ABI '16' with architecture 'arm', ABI 'armeabi-v7a', and processor 'armv7-a'
- Android: Selected GCC toolchain 'arm-linux-androideabi-4.9'
- The C compiler identification is GNU 4.9.0
- The CXX compiler identification is GNU 4.9.0
- Check for working C compiler: D:\DevTools\android-ndk-r12b\toolchains\arm-linux-androideabi-4.9\prebuilt\windows-x86_64\bin\arm-linux-androideab i-gcc.exe -- works
- Detecting C compiler ABI info - done
- Detecting C compiler features - done
- Detecting CXX compiler ABI info - done
- Detecting CXX compiler features - done
- INFO. Draw module is turned off due to it is not supported on Android
- INFO. Detecting doxygen
- Found Doxygen: D:\DevTools\doxygen\bin\doxygen.exe (found suitable version "1.8.9.1", minimum required is "1.8.4")
- INFO. Doxygen is found and can be used
- INFO. Overview building is turned on
- INFO: FreeType is used by OCCT
- Found FreeType: D:\occt\3rdparty\FreeType-2.5.3\lib\libfreetype.so (found version "2.5.3")
- INFO: TKVtk and TKVtkDraw toolkits excluded due to VTK usage is disabled
- INFO: The directories of 3rdparty headers:
  D:\occt\3rdparty\FreeType-2.5.3\include\freetype
- INFO: The directories of 3rdparty libraries:
  D:\occt\3rdparty\FreeType-2.5.3\lib

INFO: (12:22:45) Start collecting all OCCT header files into D:\occt\build-android\inc ...
- INFO: (12:22:44) Compare FILES with files in package directories...
- Warning. File D:\occt\src\StepFile\step.lex is not listed in D:\occt\src\StepFile\FILES
- Warning. File D:\occt\src\StepFile\step yacc is not listed in D:\occt\src\StepFile\FILES
- INFO: (12:22:44) Create header links in inc folder...
- INFO: (12:22:50) Checking headers in inc folder...
- INFO: (12:23:00) End the collecting
- Configuring done
- Generating done
- Build files have been written to: D:\occt\build-android

D:\occt\build-android>
Building makefiles of OCCT

Open console and go to the build folder. Type "mingw32-make" (Windows) or "make" (Ubuntu) to start build process.

```
mingw32-make
```

or

```
make
```

Parallel building can be started with using **"-jN"** argument of "mingw32-make/make", where N is the number of building threads.

```
mingw32-make -j4
```

or

```
make -j4
```
Install built OCCT libraries

Type "mingw32-make/make" with argument "install" to place the libraries to the install folder (see "OCCT Configuration" section of Building with CMake)

<table>
<thead>
<tr>
<th>mingw32-make install</th>
</tr>
</thead>
</table>

or

<table>
<thead>
<tr>
<th>make install</th>
</tr>
</thead>
</table>
Building with MS Visual C++
General

This page describes steps to build OCCT libraries from a complete source archive on Windows with **MS Visual C++** using projects generated by **genproj** tool. It is an alternative to use of CMake build system (see **Building with CMake**).

**genproj** is a legacy tool (originated from command "wgenproj" in WOK) for generation of Visual Studio, Code.Blocks, and XCode project files used for building Open CASCADE Technology. These project files are placed inside OCCT directory (in *adm* subfolder) and use relative paths, thus can be moved together with sources.

The project files included in official distribution of OCCT are generated by this tool. If you have official distribution with project files included, you can use them directly without a need to call **genproj**.
Third-party libraries

Before building OCCT, make sure to have all the required third-party libraries installed.

The easiest way to install third-party libraries is to download archive with pre-built binaries, corresponding to version of Visual Studio you are using, from http://www.opencascade.com/content/3rd-party-components.

You can also build third-party libraries from their sources, see Building 3rd-party libraries on Windows for instructions.
Configuration

If you have Visual Studio projects already available (pre-installed or generated), you can edit file `custom.bat` manually to adjust the environment:

- **VCVER** – specification of format of project files, defining also version of Visual Studio to be used, and default name of the sub-folder for binaries:

<table>
<thead>
<tr>
<th>VCVER</th>
<th>Visual Studio version</th>
<th>Windows Platform</th>
<th>Binaries folder name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vc10</td>
<td>2010 (10)</td>
<td>Desktop (Windows API)</td>
<td>vc10</td>
</tr>
<tr>
<td>vc11</td>
<td>2012 (11)</td>
<td>Desktop (Windows API)</td>
<td>vc11</td>
</tr>
<tr>
<td>vc12</td>
<td>2013 (12)</td>
<td>Desktop (Windows API)</td>
<td>vc12</td>
</tr>
<tr>
<td>vc14</td>
<td>2015 (14)</td>
<td>Desktop (Windows API)</td>
<td>vc14</td>
</tr>
<tr>
<td>vc14-uwp</td>
<td>2015 (14)</td>
<td>UWP (Universal Windows Platform)</td>
<td>vc14-uwp</td>
</tr>
<tr>
<td>vc141</td>
<td>2017 (15)</td>
<td>Desktop (Windows API)</td>
<td>vc14</td>
</tr>
<tr>
<td>vc141-uwp</td>
<td>2017 (15)</td>
<td>UWP (Universal Windows Platform)</td>
<td>vc14-uwp</td>
</tr>
</tbody>
</table>

- **ARCH** – architecture (32 or 64), affects only PATH variable for execution
- **HAVE_*** – flags to enable or disable use of optional third-party products
- **CSF_OPT_*** – paths to search for includes and binaries of all used third-party products
- **SHORTCUT_HEADERS** – defines method for population of folder `inc` by header files. Supported methods are:
  - *Copy* - headers will be copied from `src`;
- **ShortCut** - short-cut header files will be created, redirecting to same-named header located in `src`;
- "HardLink*" - hard links to headers located in `src` will be created.

Alternatively, you can launch **genconf**, a GUI tool allowing to configure build options interactively. That tool will analyze your environment and propose you to choose available options:

- Version of Visual Studio to be used (from the list of installed ones, detected by presence of environment variables like `VS100COMNTOOLS`).
- Method to populate folder `inc` (short-cuts by default).
- Location of third-party libraries (usually downloaded from OCCT web site, see above).
- Path to common directory where third-party libraries are located (optional).
- Paths to headers and binaries of the third-party libraries (found automatically basing on previous options; click button "Reset" to update).
- Generation of PDB files within Release build ("Release with Debug info", false by default).
Click "Save" to store the specified configuration in *custom.bat* file.
Projects generation

Launch `genproj` to update content of `inc` folder and generate project files after changes in OCCT code affecting layout or composition of source files.

**Note**
To use `genproj` and `genconf` tools you need to have Tcl installed and accessible by PATH. If Tcl is not found, the tool may prompt you to enter the path to directory where Tcl can be found.

```
$ genproj.bat
```

Note that if `custom.bat` is not present, `genproj` will start `genconf` to configure environment.
Building

Launch `msvc.bat` to start Visual Studio with all necessary environment variables defined, and build the whole solution or required toolkits.

Note: the MSVC project files are located in folders `adm\msvc\vc...`. Binaries are produced in `win32` or `win64` folders.

To start DRAW, launch `draw.bat`. 

---

Generated on Wed Aug 30 2017 17:04:22 for Open CASCADE Technology by doxygen 1.8.13
Building with Code::Blocks
General

This file describes steps to build OCCT libraries from sources using Code::Blocks, a cross-platform IDE, using project files generated by OCCT legacy tool genproj. It can be used as an alternative to CMake build system (see Building with CMake) for all supported platforms.
Third-party libraries

Before building OCCT, make sure to have all the needed third-party libraries installed, see Building OCCT from sources.
Configuration

Before building it is necessary to set up build environment.

The environment is defined in the file `custom.sh` (on Linux and OS X) or `custom.bat` (on Windows) which can be edited directly:

- Add paths to includes of used third-party libraries in variable `CSF_OPT_INC`.
- Add paths to their binary libraries in variable `CSF_OPT_LIB64`.
- Set variable `SHORTCUT_HEADERS` to specify a method for population of folder `inc` by header files. Supported methods are:
  - `Copy` - headers will be copied from `src`;
  - `ShortCut` - short-cut header files will be created, redirecting to same-named header located in `src`;
  - "HardLink*" - hard links to headers located in `src` will be created.
- For optional third-party libraries, set corresponding environment variable `HAVE_<LIBRARY_NAME>` to either `false`, e.g.:
  ```
  export HAVE_GL2PS=false
  ```

Alternatively, or when `custom.sh` or `custom.bat` does not exist, you can launch `genconf` tool to configure environment interactively:
Click "Save" to store the specified configuration in custom.sh or custom.bat file.
Projects generation

Launch **genproj** tool with option *cbp* to update content of *inc* folder and generate project files after changes in OCCT code affecting layout or composition of source files:

```
$ cd /dev/OCCT/opencascade-7.0.0
$ ./genproj cbp
```

The generated Code::Blocks project are placed into subfolder *adm/<OS>/cbp*.

**Note**

To use **genproj** and **genconf** tools you need to have Tcl installed and accessible by PATH.
Building

To start **Code::Blocks**, launch script `codeblocks.sh`.

To build all toolkits, click **Build->Build workspace** in the menu bar.

To start **DRAWEXE**, which has been built with **Code::Blocks** on Mac OS X, run the script

```
./draw.sh cbp [d]
```

Option *d* is used if OCCT has been built in **Debug** mode.
Open CASCADE Technology  7.2.0

Building with Xcode

Table of Contents

- General
- Third-party libraries
- Configuration
- Projects generation
- Building
- Launching DRAW
General

This file describes steps to build OCCT libraries from sources on Mac OS X with Xcode projects, generated by OCCT legacy tool genproj.
Third-party libraries

Before building OCCT, make sure to have all the needed third-party libraries installed. On OS X we recommend to use native libraries. You can also build third-party libraries from their sources, see Building 3rd-party libraries on MacOS X for instructions.
Configuration

Before building it is necessary to set up build environment.

The environment is defined in the file `custom.sh` which can be edited directly:

- Add paths to includes of used third-party libraries in variable `CSF_OPT_INC` (use colon `:` as path separator).
- Add paths to their binary libraries in variable `CSF_OPT_LIB64`.
- Set variable `SHORTCUT_HEADERS` to specify a method for population of folder `inc` by header files. Supported methods are:
  - *Copy* - headers will be copied from `src`;
  - *Shortcut* - short-cut header files will be created, redirecting to same-named header located in `src`;
  - "HardLink*" - hard links to headers located in `src` will be created.
- For optional third-party libraries, set corresponding environment variable `HAVE_<LIBRARY_NAME>` to either `false`, e.g.:
  ```sh
  export HAVE_GL2PS=false
  ```

Alternatively, or when `custom.sh` does not exist, you can launch `genconf.sh` to configure environment interactively:
Click "Save" to store the specified configuration in *custom.sh* file.
Projects generation

Launch `genproj` tool to update content of `inc` folder and generate project files after changes in OCCT code affecting layout or composition of source files.

**Note**
To use `genproj` and `genconf` tools you need to have Tcl installed and accessible by PATH.

For instance, in Terminal application:

```
$ cd /dev/OCCT/opencascade-7.0.0
$ ./genproj
```
Building

To start Xcode, launch script xcode.sh.

To build a certain toolkit, select it in Scheme drop-down list in Xcode toolbar, press Product in the menu and click Build button.

To build the entire OCCT:

- Create a new empty project (select File -> New -> Project -> Empty project in the menu; input the project name, e.g. OCCT; then click Next and Create).
- Drag and drop the OCCT folder in the created OCCT project in the Project navigator.
- Select File -> New -> Target -> Aggregate in the menu.
- Enter the project name (e.g. OCCT) and click Finish. The Build Phases tab will open.
- Click "+" button to add the necessary toolkits to the target project. It is possible to select all toolkits by pressing Command+A combination.
Launching DRAWX

To start DRAWXE, which has been built with Xcode on Mac OS X, perform the following steps:

1. Open Terminal application

2. Enter <OCCT_ROOT_DIR>:

   
   ```
   cd <OCCT_ROOT_DIR>
   ```

3. Run the script

   ```
   ./draw_cbp.sh xcd [d]
   ```

Option d is used if OCCT has been built in Debug mode.
Introduction

This document provides practical guidelines for generation and editing of OCCT user documentation.
Prerequisites

You need to have the following software installed to generate the documentation.

**Tcl/Tk** Version 8.5 or 8.6: [http://www.tcl.tk/software/tcltk/download.html](http://www.tcl.tk/software/tcltk/download.html)

**Doxygen** Version 1.8.4 or above: [http://www.stack.nl/~dimitri/doxygen/download.html](http://www.stack.nl/~dimitri/doxygen/download.html)


**MiKTeX** or other package providing **pdflatex** command (only needed for generation of PDF documents): [http://miktex.org/download](http://miktex.org/download)

**Inkscape** (only needed for generation of PDF documents containing SVG images): [http://www.inkscape.org/download](http://www.inkscape.org/download)

When generating PDF documentation, **pdflatex** and **inkscape** executables should be accessible by PATH variable. You can use **custom.bat** file to add necessary paths to the **PATH** variable.

Note that in the process of PDF generation MiKTeX may need some packages not installed by default. We recommend setting option "Install missing packages on-the-fly" to "Ask me first" (default) during MiKTeX installation:
On the first run of `pdflatex` it will open a dialog window prompting for installation of missing packages. Follow the instructions to proceed (define proxy settings if needed, select a mirror site to download from, etc.).

**MathJax** is used for rendering math formulas in browser (HTML and CHM outputs): [http://www.mathjax.org](http://www.mathjax.org).

By default MathJAX scripts and fonts work on-line and no installation of MathJAX is necessary if Internet is accessible. If you need to use OCCT documentation while off-line, you can install a local copy of MatJAX, see [https://docs.mathjax.org/en/v2.7-latest/start.html#installing-your-own-copy-of-mathjax](https://docs.mathjax.org/en/v2.7-latest/start.html#installing-your-own-copy-of-mathjax). See **Formulas** for more details on inserting mathematical expressions.
Documentation Generation

Run command `gendoc` from command prompt (with OCCT directory as current one) to generate OCCT documentation. The synopsis is:

```
gendoc \[-h\] {-refman|-overview} \[-html|-pdf|-chm\] \[-m=<list of modules>|-ug=<list of docs>\] \[-v\] \[-s=<search_mode>\] \[-mathjax=<path>\]
```

Here the options are:

- **Choice of documentation to be generated:**
  - `-overview`: To generate Overview and User Guides (cannot be used with `-refman`)
  - `-refman`: To generate class Reference Manual (cannot be used with `-overview`)

- **Choice of output format:**
  - `-html`: To generate HTML files (default, cannot be used with `-pdf` or `-chm`)
  - `-pdf`: To generate PDF files (cannot be used with `-refman`, `-html`, or `-chm`)
  - `-chm`: To generate CHM files (cannot be used with `-html` or `-pdf`)

- **Additional options:**
  - `-m=<modules_list>`: List of OCCT modules (separated with comma), for generation of Reference Manual
  - `-ug=<docs_list>`: List of MarkDown documents (separated with comma), to use for generation of Overview / User Guides
  - `-mathjax=<path>`: To use local or alternative copy of MathJax
  - `-s=<search_mode>`: Specifies the Search mode of HTML documents; can be: none | local | server | external
  - `-h`: Prints this help message
  - `-v`: Enables more verbose output

**Note**

- In case of PDF output the utility generates a separate PDF file for each document;
- In case of HTML output the utility generates a common Table of
contents containing references to all documents.
- In case of CHM output single CHM file is generated

**Examples**

To generate the output for a specific document specify the path to the corresponding MarkDown file (paths relative to dox sub-folder can be given), for instance:

```
> gendoc -overview -
  ug=dev_guides/documentation/documentation.md
```

To generate Reference Manual for the whole Open CASCADE Technology library, run:

```
> gendoc -refman
```

To generate Reference Manual for Foundation Classes and Modeling Data modules only, with search option, run:

```
> gendoc -refman -
  m=FoundationClasses,ModelingData,ModelingAlgorit
  hms -s=local
```
Documentation Conventions

This section contains information about file format conventions, directories structure, etc.
File Format

The format used for documentation is MarkDown with Doxygen extensions. The MarkDown files have a *md extension and are based on rules described in Appendix 1: Document Syntax section.
Each document has its own folder if there are any images used in it. These images are stored in `images` subfolder.

If you want to use the same image for several documents, you can place it in `dox/resources` folder.

**Note:** To avoid incorrect image display, use a relative path to the image (starting from `dox` folder). For instance:

```plaintext
@figure{/dev_guides/documentation/images/documentation_test_image.svg,"",420}
```

The documentation is generated in subfolder `doc`:

- **html** – a directory for generated HTML pages;
- **pdf** – a directory for generated PDF files.
Adding a New Document

Place a new document in the folder taking into account its logical position in the documentation hierarchy. For instance, the document `svn.md` about the use of SVN to work with OCCT source code can be placed into `/dox/dev_guides/`.

If there are images in the document, it should be placed in its own folder containing a subfolder for images. For instance:

- `/dox/dev_guides/svn/` – for `svn.md` file;

Add a relative path to `svn.md` in file `dox/FILES.txt`. For instance

```
dev_guides/svn/svn.md
```

**Note** that the order of paths to documents in `FILES.txt` is reproduced in the Table of Contents in the HTML output. Please, place them logically.

**Note** that you should specify a file tag, not the document name. See **Header and hierarchic document structure** section for details.
Additional Resources

More information about OCCT can be found at http://www.opencascade.com and http://dev.opencascade.org sites.

The information on formula syntax can be found at: http://en.wikipedia.org/wiki/Help:Displaying_a_formula

More information on MarkDown and Doxygen syntax can be found at: http://www.stack.nl/~dimitri/doxygen/manual
Appendix 1: Document Syntax

A document file in *.md format must start with a proper header defining a caption and a unique tag.

Documentation System {#dev_guides__documentation}

The document structure is formed by sections that must be defined consistently.

The document can contain plain text, lists, tables, code snippets, images, math, etc. Any specific text elements can be introduced by Markdown language tags or by usual HTML tags.

The table of contents, page numbers (in PDF), and figure numbers (in PDF) are generated automatically.
Headers and hierarchic document structure

Headers of different levels can be specified with the following tags:

- `@section` – for the first-level headers;
- `@subsection` – for the second level headers;
- `@subsubsection` – for the third level headers.

For example:

```
@section occt_ocaf_1 Basic Concepts
@subsection occt_ocaf_1_1 Applications and Documents
@subsubsection occt_ocaf_1_1_1 The document and the data framework
```

Please, note that section names can be used for references within the document and in other documents, so it is necessary to use the common prefix indicative of the document name for all section names in the given document. For example, `occt_ocaf` for sections in Open CASCADE Application Framework manual.

The remaining part of section names in most documents consists only of numbers, for example `1_1`. Actually, the hierarchical structure of the output table of contents is not based on these numbers and is generated automatically.

The numbers are only indicative of a section location in the body of the document. However, duplicate section names in a document inevitably cause errors during generation.

If you insert a section in the middle of a big document, do not renumber the document to the end (which is inefficient and error prone), but choose an arbitrary number or letter, not yet used in the document section naming, and base the naming in this section on it.

The section hierarchy is limited to three levels and further levels cannot be presented in the Table of Contents.
However, the fourth and fifth level headers can be tagged with #### and ###### correspondingly.

It is also possible to use tags ## and ### for second and third level headers if you do not wish to show them in the table of contents or make references to them.
Plain Text

A plain text is organized in paragraphs, separated by empty lines in MarkDown source. The length of lines is not restricted; it is recommended to put each sentence on a separate line – this is optimal for easier comparison of different versions of the same document.

To insert special symbols, like <, > or \, prepend them with \ character: \ <, \>, \\, etc. To emphasize a word or a group of words, wrap the text with one pair of asterisks (*) or underscores (__) to make it *italic* and two pairs of these symbols to make it **Bold**.

**Note** that if your emphasized text starts or ends with a special symbol, the asterisks may not work. Use explicit HTML tags <i></i> and <b></b> instead.
Lists

To create a bulleted list, start each line with a hyphen or an asterisk, followed by a space. List items can be nested. This code:

```
* Bullet 1
* Bullet 2
  - Bullet 2a
  - Bullet 2b
* Bullet 3
```

produces this list:

- Bullet 1
- Bullet 2
  - Bullet 2a
  - Bullet 2b
- Bullet 3

To create a numbered list, start each line with number and a period, then a space. Numbered lists can also be nested. Thus this code

```
1. List item 1
   1. Sub-item 1
   2. Sub-item 2
2. List item 2
4. List item 3
```

produces this list:

1. List item 1
   1. Sub-item 1
   2. Sub-item 2
2. List item 2
3. List item 3

**Note** that numbers of list items in the output are generated so they do not necessarily follow the numbering of source items.
In some cases automatic generation adversely restarts the numbering, i.e. you get list items 1. 1. 1. instead of 1. 2. 3. in the output. The use of explicit HTML tags `<ol></ol>` and `<li></li>` can help in this case.

Each list item can contain several paragraphs of text; these paragraphs must have the same indentation as text after bullet or number in the numbered list item (otherwise numbering will be broken).

Code blocks can be inserted as paragraphs with additional indentation (4 spaces more). Note that fenced code blocks do not work within numbered lists and their use may cause numeration to be reset.

Example of a complex nested list:

1. List item 1
   
   Additional paragraph
   
   code fragment
   
   One more paragraph
   
   1. Sub-item 1
      
      code fragment for sub-item 1
   
   2. Sub-item 2
      
      Paragraph for sub-item 2
      
      Yet one more paragraph for list item 1
   
2. List item 2
Tables

A table consists of a header line, a separator line, and at least one row line. Table columns are separated by the pipe (|) character. The following example:

<table>
<thead>
<tr>
<th>First Header</th>
<th>Second Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Content Cell</td>
<td>Content Cell</td>
</tr>
<tr>
<td>Content Cell</td>
<td>Content Cell</td>
</tr>
</tbody>
</table>

will produce the following table:

<table>
<thead>
<tr>
<th>First Header</th>
<th>Second Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content Cell</td>
<td>Content Cell</td>
</tr>
<tr>
<td>Content Cell</td>
<td>Content Cell</td>
</tr>
</tbody>
</table>

Column alignment can be controlled via one or two colons at the header separator line:

<table>
<thead>
<tr>
<th>Right</th>
<th>Center</th>
<th>Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>----:</td>
<td>:----:</td>
<td>:----</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

which will looks as follows:

<table>
<thead>
<tr>
<th>Right</th>
<th>Center</th>
<th>Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

Note that each table row should be contained in one line of text; complex tables can be created using HTML tags.
Code Blocks

Paragraphs indented with 4 or more spaces are considered as code fragments and rendered using Courier font. Example:

```
This line is indented by 4 spaces and rendered as a code block.
```

A fenced code block does not require indentation, and is defined by a pair of "fence lines". Such line consists of 3 or more tilde (~) characters on a line. The end of the block should have the same number of tildes. Thus it is strongly advised to use only three or four tildes.

By default the output is the same as for a normal code block. To highlight the code, the developer has to indicate the typical file extension, which corresponds to the programming language, after the opening fence. For highlighting according to the C++ language, for instance, write the following code (the curly braces and dot are optional):

```
~~~{.cpp}
int func(int a, int b) { return a*b; }
~~~
```

which will produce:

```
int func(int a, int b) { return a*b; }
```

Smaller code blocks can be inserted by wrapping with tags `@code` and `@endcode`.

Verbatim content (same as code but without syntax highlighting) can be inserted by wrapping with tags `@verbatim` and `@endverbatim`.
Quotes

Text quoted from other sources can be indented using ">" tag. For example:

> [Regression in 6.9.0] *IGES - Export of a reversed face leads to wrong data*

will produce

[Regression in 6.9.0] IGES - Export of a reversed face leads to wrong data

Note that this tag should prefix each line of the quoted text. Empty lines in the quoted text, if any, should not have trailing spaces after the ">" (lines with trailing spaces will break the quote block).
References

To insert a reference to a website, it is sufficient to write an URL. For example: `http://en.wikipedia.org`

To insert a reference to a document or its subsection, use command `@ref` followed by the document or section tag name. For instance,

```
@ref OCCT_DM_SECTION_A
```

will be rendered as *Appendix 1: Document Syntax*.

Note that links between documents will not work in PDF output if each document is generated independently. Hence it is recommended to add a name of the referenced section after the tag name in the `@ref` command (in quotes): this will guarantee that the reference is recognizable for the reader even if the cross-link is not instantiated. For instance:

```
@ref occt_modat_1 "Geometry Utilities"
```

will be rendered as *Geometry Utilities*. 
Images

For inserting images into the document use the command @figure, as follows:

@figure{/relative/path/to/image/image_file_name.png,"Image caption"}

The first argument is a path to the image file, relative to the dox folder. The supported formats for images are PNG, JPG, and SVG. The file extension must be lowercase and correspond to the file format. The image file name should have no dots except for the one before extension (names with more than one dot confuse \texttt{pdflatex}).

The second argument is optional, it defines the caption for the image to be inserted. The caption argument, if given, should be quoted, even if it is a single word. Captions are included below the image; in PDF output the images with caption are numbered automatically.

Example:

@figure{/dev_guides/documentation/images/documentation_test_image.svg,"Test SVG image"}

is rendered as:

\begin{figure}
\centering
\includegraphics{documentation_test_image.svg}
\caption{Test SVG image}
\end{figure}

We recommend using \texttt{Inkscape} for creation and edition of vector graphics. The graphics created in MS Word Draw and some other vector
editors can be copy-pasted to Inkscape and saved as SVG images.

Note that the image that will be included in documentation is the whole page of the Inkscape document; use option "Resize page to content" in menu **File -> Document properties** of Inkscape to fit page dimensions to the picture (adding margins as necessary).

Note that the *figure* command is an alias to the standard Doxygen command *image* repeated twice: once for HTML and then for Latex output (used for PDF generation). Thus if HTML and PDF outputs should include different images or captions, command "image" can be used:

```plaintext
@image html /relative/path/to/image/occ_logo_for_html.png
@image latex /relative/path/to/image/occ_logo_for_pdf.png
```
Table Of Contents

Use @tableofcontents tag to get the table of contents at the beginning of the document.

Actually, it is not strictly necessary now because TreeView option for HTML is used. The TOC in the PDF document will be generated automatically.
Formulas

Formulas within MarkDown documents can be defined using LaTeX syntax.

Equations can be written by several ways:

1. Unnumbered displayed formulas that are centered on a separate line. These formulas should be put between \[ and \] tags. An example:

\[
|I_2| = \left| \int_0^T \psi(t) \left\{ u(a,t) - \int_{\gamma(t)}^a \frac{d\theta}{k(\theta,t)} \int_a^\theta c(\xi)u_t(\xi,t)\,d\xi \right\} \,dt \right|
\]

gives the following result:

\[
|I_2| = \left| \int_0^T \psi(t) \left\{ u(a,t) - \int_{\gamma(t)}^a \frac{d\theta}{k(\theta,t)} \int_a^\theta c(\xi)u_t(\xi,t)\,d\xi \right\} \,dt \right|
\]

2. Formulas can also be put between \begin{align} and \end{align} tags.

\[
\begin{align}
|I_2| &= \left| \int_0^T \psi(t) \left\{ u(a,t) - \int_{\gamma(t)}^a \frac{d\theta}{k(\theta,t)} \int_a^\theta c(\xi)u_t(\xi,t)\,d\xi \right\} \,dt \right|
\end{align}
\]
For example:

\begin{align}
\dot{x} & = \sigma(y-x) \\
\dot{y} & = \rho x - y - xz \\
\dot{z} & = -\beta z + xy
\end{align}

gives the following result:

\begin{align}
\dot{x} & = \sigma(y-x) \\
\dot{y} & = \rho x - y - xz \\
\dot{z} & = -\beta z + xy
\end{align}

3. Inline formulas can be specified using this syntax:

@f$\sqrt{3x-1}+(1+x)^2$@f$

that leads to the following result: \((\sqrt{3x-1}+(1+x)^2\)
Table of Contents

- Introduction
  - Scope of the document
- Naming Conventions
  - General naming rules
  - Names of development units
  - Names of variables
- Formatting rules
- Documentation rules
- Application design
- General C/C++ rules
- Portability issues
- Stability issues
- Performance issues
- Draw Harness command
- Examples
Introduction

The purpose of this document is to define a common programming style for Open CASCADE Technology.

The common style facilitates understanding and maintaining a code developed cooperatively by several programmers. In addition, it enables construction of tools that incorporate knowledge of these standards to help in the programming.

OCCT programming style follows common and appropriate best practices, so some guidelines have been excerpted from the public domain.

The guide can be improved in the future as new ideas and enhancements are added.
Scope of the document

Rules in this document refer to C++ code. However, with minor exceptions due to language restrictions, they are applicable to any sources in Open CASCADE Technology framework, including:

- C/C++
- GLSL programs
- OpenCL kernels
- TCL scripts and test cases
Naming Conventions

General naming rules

The names considered in this section mainly refer to the interface of Open CASCADE Technology libraries or source code itself.

International language [MANDATORY]

Open CASCADE Technology is an open source platform available for an international community, thus all names need to be composed of English words or their abbreviations.

Meaningful names

Names should be meaningful or, at least, contain a meaningful part. To better understand this requirement, let us examine the existing names of toolkits, packages, classes and methods:

- Packages containing words Geom or Geom2d in their names are related to geometrical data and operations.
- Packages containing words TopoDS or BRep in their names are related to topological data and operations.
- Packages ending with ...Test define Draw Harness plugins.
- Methods starting with Get... and Set... are usually responsible for correspondingly retrieving and storing data.

Related names

Names related to a logically connected functionality should have the same prefix (start with the same letters) or, at least, have any other common part. For example, method GetCoord returns a triple of real values and is defined for directions, vectors and points. The logical connection is obvious.

Camel Case style
Camel Case style is preferred for names. For example:

```plaintext
Standard_Integer aWidthOfBox;  // this is bad
Standard_Integer width_of_box; // this is bad
Standard_Integer aWidthOfBox;  // this is OK
```
Names of development units

Usually a unit (e.g. a package) is a set of classes, methods, enumerations or any other sources implementing a common functionality, which is self-contained and independent from other parts of the library.

No underscores in unit names [MANDATORY]

Names of units should not contain underscores, unless the use of underscores is allowed explicitly.

File name extensions [MANDATORY]

The following extensions should be used for source files, depending on their type:

- `.cxx` – C++ source files
- `.hxx` – C++ header files
- `.lxx` – additional headers containing definitions of inline methods and auxiliary code

Note that `.lxx` files should be avoided in most cases - inline method should be placed in header file instead.

Prefix for toolkit names [MANDATORY]

Toolkit names are prefixed by `TK`, followed by a meaningful part of the name explaining the domain of functionality covered by the toolkit (e.g. `TKOpenGL`).

Names of classes

Usually the names of source files located in a unit start from the unit name separated from the other part of the file name by underscore "_".

Thus, the names of files containing sources of C++ classes that belong to a package are constructed according to the following template:
For example, file `Adaptor2d_Curve2d.cxx` belongs to the package `Adaptor2d`.

Files that contain sources related to the whole unit are called by the unit name with appropriate extension.

### Names of functions

The term **function** here is defined as:

- Any class method
- Any package method
- Any non-member procedure or function

It is preferred to start names of public methods from an upper case character and to start names of protected and private methods from a lower case character.

```cpp
class MyPackage_MyClass
{

public:

    Standard_Integer Value() const;
    void SetValue (const Standard_Integer theValue);

private:

    void setIntegerValue (const Standard_Integer theValue);

};
```
Names of variables

There are several rules that describe currently accepted practices for naming variables.

Naming of variables

Name of a variable should not conflict with the existing or possible global names (for packages, macros, functions, global variables, etc.).

The name of a variable should not start with an underscore.

See the following examples:

```plaintext
Standard_Integer Elapsed_Time = 0;  // this is bad - possible class name
Standard_Integer gp = 0;            // this is bad - existing package name
Standard_Integer aGp = 0;           // this is OK
Standard_Integer _KERNEL = 0;       // this is bad
Standard_Integer THE_KERNEL = 0;    // this is OK
```

Names of function parameters

The name of a function (procedure, class method) parameter should start with prefix `the` followed by the meaningful part of the name starting with a capital letter.

See the following examples:

```plaintext
void Package_MyClass::MyFunction (const gp_Pnt& p);  // this is bad
void Package_MyClass::MyFunction (const gp_Pnt& theP); // this is OK
void Package_MyClass::MyFunction (const gp_Pnt& thePoint); // this is preferred
```
Names of class member variables

The name of a class member variable should start with prefix *my* followed by the meaningful of the name starting with a capital letter.

See the following examples:

```plaintext
Standard_Integer counter;     // This is bad
Standard_Integer myC;         // This is OK
Standard_Integer myCounter;   // This is preferred
```

Names of global variables

It is strongly recommended to avoid defining any global variables. However, as soon as a global variable is necessary, its name should be prefixed by the name of a class or a package where it is defined followed with *_my*.

See the following examples:

```plaintext
Standard_Integer MyPackage_myGlobalVariable = 0;
Standard_Integer MyPackage_MyClass_myGlobalVariable = 0;
```

Static constants within the file should be written in upper-case and begin with prefix *THE*:

```plaintext
namespace
{
    static const Standard_Real THE_CONSTANT_COEF = 3.14;
}
```

Names of local variables

The name of a local variable should be distinguishable from the name of a function parameter, a class member variable and a global variable.

It is preferred to prefix local variable names with *a* and *an* (or *is*, *to* and *has* for Boolean variables).
See the following example:

```
Standard_Integer theI;     // this is bad
Standard_Integer i;        // this is bad
Standard_Integer index;    // this is bad
Standard_Integer anIndex;  // this is OK
```

**Avoid dummy names**

Avoid dummy names, such as $i, j, k$. Such names are meaningless and easy to mix up.

The code becomes more and more complicated when such dummy names are used there multiple times with different meanings, or in cycles with different iteration ranges, etc.

See the following examples for preferred style:

```
void Average (const Standard_Real** theArray,
             Standard_Integer theRowsNb,
             Standard_Integer theRowLen,
             Standard_Real& theResult)
{
    theResult = 0.0;
    for (Standard_Integer aRow = 0; aRow < aRowsNb; ++aRow)
    {
        for (Standard_Integer aCol = 0; aCol < aRowLen; ++aCol)
        {
            theResult += theArray[aRow][aCol];
        }
    }
    theResult /= Standard_Real(aRowsNb * aRowLen);
}
```
Formatting rules

To improve the open source readability and, consequently, maintainability, the following set of rules is applied.

International language [MANDATORY]

All comments in all sources must be in English.

Line length

Try to stay within the limit of 120 characters per line in all sources.

C++ style comments

Prefer C++ style comments in C++ sources.

Commenting out unused code

Delete unused code instead of commenting it or using #define.

Indentation in sources [MANDATORY]

Indentation in all sources should be set to two space characters. Use of tabulation characters for indentation is disallowed.

Separating spaces

Punctuation rules follow the rules of the English language.

- C/C++ reserved words, commas, colons and semicolons should be followed by a space character if they are not at the end of a line.
- There should be no space characters after '(' and before ')'. Closing and opening brackets should be separated by a space character.
- For better readability it is also recommended to surround conventional operators by a space character. Examples:
while (true)                      // NOT:
   while( true ) ...  

{                                  // NOT:
   DoSomething (theA, theB, theC, theD); // NOT: 
       DoSomething(theA, theB, theC, theD);  
}

for (anIter = 0; anIter < 10; ++anIter) // NOT: for
   (anIter=0; anIter<10;++anIter){
   theA = (theB + theC) * theD;          // NOT: theA=
       (theB+theC)*theD

// Declaration of pointers and references

In declarations of simple pointers and references put asterisk (*) or ampersand (&) right after the type without extra space.

Since declaration of several variables with mixed pointer types contradicts this rule, it should be avoided. Instead, declare each variable independently with fully qualified type.

Examples:

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard_Integer</td>
<td>*theVariable; // not</td>
</tr>
<tr>
<td>Standard_Integer</td>
<td>&amp;theVariable; // not</td>
</tr>
<tr>
<td>Standard_Integer*</td>
<td>theVariable; // this is OK</td>
</tr>
<tr>
<td>Standard_Integer*</td>
<td>&amp;theVariable; // not</td>
</tr>
<tr>
<td>Standard_Integer*</td>
<td>&amp;theVariable; // this is OK</td>
</tr>
<tr>
<td>Standard_Integer**</td>
<td>theVariable; // not</td>
</tr>
</tbody>
</table>
Separate logical blocks

Separate logical blocks of code with one blank line and comments.

See the following example:

```plaintext
// check arguments
Standard_Integer anArgsNb = argCount();
if (anArgsNb < 3 || isSmthInvalid)
{
    return THE_ARG_INVALID;
}

// read and check header
...
...

// do our job
...
...
```

Notice that multiple blank lines should be avoided.

Separate function bodies [MANDATORY]

Use function descriptive blocks to separate function bodies from each other. Each descriptive block should contain at least a function name and purpose description.

See the following example:
// function : TellMeSmthGood
// purpose : Gives me good news
// ---------------------------------------------------------------
//
void TellMeSmthGood()
{
  ...
}
//
// function : TellMeSmthBad
// purpose : Gives me bad news
// ---------------------------------------------------------------
void TellMeSmthBad()
{
  ...
}

**Block layout [MANDATORY]**

Figure brackets {} and each operator (for, if, else, try, catch) should be written on a dedicated line.

In general, the layout should be as follows:

```c
while (expression)
{
  ...
}
```
Entering a block increases and leaving a block decreases the indentation by one tabulation.

**Single-line operators**

Single-line conditional operators *(if, while, for, etc.)* can be written without brackets on the following line.

```
if (!myIsInit) return Standard_False; // bad

if (thePtr == NULL) return Standard_False; // OK

if (!theAlgo.IsNull()) { // preferred
  DoSomething();
}
```

Having all code in the same line is less convenient for debugging.

**Comparison expressions with constants**

In comparisons, put the variable *(in the current context)* on the left side and constant on the right side of expression. That is, the so called "Yoda style" is to be avoided.

```
if (NULL != thePointer) // Yoda style, not recommended
if (thePointer != NULL) // OK

if (34 < anIter) // Yoda style, not recommended
if (anIter > 34) // OK

if (theNbValues >= anIter) // bad style (constant function argument vs. local variable)
if (anIter <= theNbValues) // OK
```
if (THE_LIMIT == theValue) // bad style (global constant vs. variable)
if (theValue == THE_LIMIT) // OK

Alignment

Use alignment wherever it enhances the readability. See the following example:

MyPackage_MyClass anObject;
Standard_Real aMinimum = 0.0;
Standard_Integer aVal = theVal;
switch (aVal)
{
  case 0: computeSomething(); break;
  case 12: computeSomethingElse (aMinimum); break;
  case 3:
    default: computeSomethingElseYet(); break;
}

Indentation of comments

Comments should be indented in the same way as the code to which they refer or they can be in the same line if they are short.

The text of the comment should be separated from the slash character by a single space character.

See the following example:

while (expression) //bad comment
{
  // this is a long multi-line comment
  // which is really required
  DoSomething();    // maybe, enough
  DoSomethingMore(); // again
}
Early return statement

Use an early return condition rather than collect indentations.

Write like this:

```cpp
Standard_Integer ComputeSumm (const Standard_Integer* theArray,
                          const Standard_Size theSize)
{
    Standard_Integer aSumm = 0;
    if (theArray == NULL || theSize == 0)
    {
        return 0;
    }
    ...
    return aSumm;
}
```

Rather than:

```cpp
Standard_Integer ComputeSumm (const Standard_Integer* theArray,
                          const Standard_Size theSize)
{
    Standard_Integer aSumm = 0;
    if (theArray != NULL && theSize != 0)
    {
        ...
    }
    return aSumm;
}
```

This helps to improve readability and reduce the unnecessary indentation depth.

Trailing spaces
Trailing spaces should be removed whenever possible. Spaces at the end of a line are useless and do not affect functionality.

**headers order**

Split headers into groups: system headers, headers per each framework, project headers; sort the list of includes alphabetically. Within the class source file, the class header file should be included first.

This rule improves readability, allows detecting useless multiple header inclusions and makes 3rd-party dependencies clearly visible. Inclusion of class header on top verifies consistency of the header (e.g. that header file does not use any undefined declarations due to missing includes of dependencies).

An exception to the rule is ordering system headers generating a macros declaration conflicts (like "windows.h" or "X11/Xlib.h") - these headers should be placed in the way solving the conflict.

```cpp
// the header file of implemented class
#include <PackageName_ClassName.hxx>

// OCCT headers
#include <gp_Pnt.hxx>
#include <gp_Vec.hxx>
#include <NCollection_List.hxx>

// Qt headers
#include <QDataStream>
#include <QString>

// system headers
#include <iostream>
#include <windows.h>
```
Documentation rules

The source code is one of the most important references for documentation. The comments in the source code should be complete enough to allow understanding the corresponding code and to serve as basis for other documents.

The main reasons why the comments are regarded as documentation and should be maintained are:

- The comments are easy to reach – they are always together with the source code;
- It is easy to update a description in the comment when the source is modified;
- The source by itself is a good context to describe various details that would require much more explanations in a separate document;
- As a summary, this is the most cost-effective documentation.

The comments should be compatible with Doxygen tool for automatic documentation generation (thus should use compatible tags).

Documenting classes [MANDATORY]

Each class should be documented in its header file (.hxx). The comment should give enough details for the reader to understand the purpose of the class and the main way of work with it.

Documenting class methods [MANDATORY]

Each class or package method should be documented in the header file (.hxx).

The comment should explain the purpose of the method, its parameters, and returned value(s). Accepted style is:

```cpp
//! Method computes the square value.
//! @param theValue the input value
//! @return squared value
```
Documenting C/C++ sources

It is very desirable to put comments in the C/C++ sources of the package/class.

They should be detailed enough to allow any person to understand what each part of code does.

It is recommended to comment all static functions (like methods in headers), and to insert at least one comment per each 10-100 lines in the function body.

There are also some rules that define how comments should be formatted, see Formatting Rules.

Following these rules is important for good comprehension of the comments. Moreover, this approach allows automatically generating user-oriented documentation directly from the commented sources.
Application design

The following rules define the common style, which should be applied by any developer contributing to the open source.

Allow possible inheritance

Try to design general classes (objects) keeping possible inheritance in mind. This rule means that the user who makes possible extensions of your class should not encounter problems of private implementation. Try to use protected members and virtual methods wherever you expect extensions in the future.

Avoid friend declarations

Avoid using 'friend' classes or functions except for some specific cases (for example, iteration) 'Friend' declarations increase coupling.

Set/get methods

Avoid providing set/get methods for all fields of the class. Intensive set/get functions break down encapsulation.

Hiding virtual functions [MANDATORY]

Avoid hiding a base class virtual function by a redefined function with a different signature. Most of the compilers issue warning on this.

Avoid mixing error reporting strategies

Try not to mix different error indication/handling strategies (exceptions or returned values) on the same application level.

Minimize compiler warnings [MANDATORY]

When compiling the source pay attention to and try to minimize compiler warnings.
Avoid unnecessary inclusions

Try to minimize compilation dependencies by removing unnecessary inclusions.
General C/C++ rules

This section defines the rules for writing a portable and maintainable C/C++ source code.

Wrapping of global variables [MANDATORY]

Use package or class methods returning reference to wrap global variables to reduce possible name space conflicts.

Avoid private members

Use protected members instead of private wherever reasonable to enable future extensions. Use private fields if future extensions should be disabled.

Constants and inlines over defines [MANDATORY]

Use constant variables (const) and inline functions instead of defines (#define).

Avoid explicit numerical values [MANDATORY]

Avoid usage of explicit numeric values. Use named constants and enumerations instead. Numbers produce difficulties for reading and maintenance.

Three mandatory methods

If a class has a destructor, an assignment operator or a copy constructor, it usually needs the other two methods.

Virtual destructor

A class with virtual function(s) ought to have a virtual destructor.
Overriding virtual methods

Declaration of overriding method should contains specifiers "virtual" and "override" (using Standard_OVERRIDE alias for compatibility with old compilers).

```cpp
class MyPackage_BaseClass
{
public:
    Standard_EXPORT virtual Standard_Boolean Perform();
};
class MyPackage_MyClass : public MyPackage_BaseClass
{
public:
    Standard_EXPORT virtual Standard_Boolean Perform() Standard_OVERRIDE;
};
```

This makes class definition more clear (virtual methods become highlighted).

Declaration of interface using pure virtual functions protects against incomplete inheritance at first level, but does not help when method is overridden multiple times within nested inheritance or when method in base class is intended to be optional.

And here "override" specifier introduces additional protection against situations when interface changes might be missed (class might contain old methods which will be never called).

Default parameter value
Do not redefine a default parameter value in an inherited function.

**Use const modifier**

Use `const` modifier wherever possible (functions parameters, return values, etc.)

**Usage of goto [MANDATORY]**

Avoid `goto` statement unless it is really needed.

**Declaring variable in for() header**

Declare a cycle variable in the header of the `for()` statement if not used out of cycle.

```cpp
Standard_Real aMinDist = Precision::Infinite();
for (NCollection_Sequence<gp_Pnt>::Iterator aPntIter (theSequence);
    aPntIter.More(); aPntIter.Next())
{
    aMinDist = Min (aMinDist, theOrigin.Distance (aPntIter.Value()));
}
```

**Condition statements within zero**

Avoid usage of C-style comparison for non-boolean variables:

```cpp
void Function (Standard_Integer theValue,
               Standard_Real* thePointer)
{
    if (!theValue) // bad style - ambiguous logic
    {
        DoSome();
    }
```
if (theValue == 0)  // OK
{
    DoSome();
}

if (thePointer != NULL) // OK, predefined NULL makes pointer comparison cleaner to reader
{
    // (nullptr should be used instead as soon as C++11 will be available)
    DoSome2();
}
Portability issues

This chapter contains rules that are critical for cross-platform portability.

Provide code portability [MANDATORY]

The source code must be portable to all platforms listed in the official 'Technical Requirements'. The term 'portable' here means 'able to be built from source'.

The C++ source code should meet C++03 standard. Any usage of compiler-specific features or further language versions (for example, C++11, until all major compilers on all supported platforms implement all its features) should be optional (used only with appropriate preprocessor checks) and non-exclusive (an alternative implementation compatible with other compilers should be provided).

Avoid usage of global variables [MANDATORY]

Avoid usage of global variables. Usage of global variables may cause problems when accessed from another shared library.

Use global (package or class) functions that return reference to static variable local to this function instead of global variables.

Another possible problem is the order of initialization of global variables defined in various libraries that may differ depending on platform, compiler and environment.

Avoid explicit basic types

Avoid explicit usage of basic types (int, float, double, etc.), use Open CASCADE Technology types from package Standard: Standard_Integer, Standard_Real, Standard_ShortReal, Standard_Boolean, Standard_CString and others or a specific typedef instead.

Use sizeof() to calculate sizes [MANDATORY]
Do not assume sizes of types. Use `sizeof()` instead to calculate sizes.

**Empty line at the end of file [MANDATORY]**

In accordance with C++03 standard source files should be trailed by an empty line. It is recommended to follow this rule for any plain text files for consistency and for correct work of git difference tools.
Stability issues

The rules listed in this chapter are important for stability of the programs that use Open CASCADE Technology libraries.

Use **OSD::SetSignal()** to catch exceptions

When using Open CASCADE Technology in an application, call **OSD::SetSignal()** function when the application is initialized.

This will install C handlers for run-time interrupt signals and exceptions, so that low-level exceptions (such as access violation, division by zero, etc.) will be redirected to C++ exceptions that use `try {...} catch (Standard_Failure) {...}` blocks.

The above rule is especially important for robustness of modeling algorithms.

Cross-referenced handles

Take care about cycling of handled references to avoid chains, which will never be freed. For this purpose, use a pointer at one (subordinate) side.

See the following example:

```cpp
class Slave;

class Master : public Standard_Transient
{
    ...
    void SetSlave (const Handle(Slave)& theSlave)
    {
        mySlave = theSlave;
    }
    ...
private:
    Handle(Slave) theSlave;  // smart pointer
```
...)
}
class Slave : public Standard_Transient
{
...
void SetMaster (const Handle(Master)& theMaster)
{
    myMaster = theMaster.get();
}
...
private:
    Master* theMaster; // simple pointer
...
}

C++ memory allocation

In C++ use new and delete operators instead of malloc() and free(). Try not to mix different memory allocation techniques.

Match new and delete [MANDATORY]

Use the same form of new and delete.

```
  aPtr1 = new TypeA[n];            ... ; delete[]
  aPtr1;
  aPtr2 = new TypeB();            ... ; delete
  aPtr2;
  aPtr3 = Standard::Allocate (4096); ... ;
      Standard::Free (aPtr3);
```

Methods managing dynamical allocation [MANDATORY]

Define a destructor, a copy constructor and an assignment operator for classes with dynamically allocated memory.
**Uninitialized variables [MANDATORY]**

Every variable should be initialized.

```plaintext
Standard_Integer aTmpVar1; // bad
Standard_Integer aTmpVar2 = 0; // OK
```

Uninitialized variables might be kept only within performance-sensitive code blocks and only when their initialization is guaranteed by subsequent code.

**Do not hide global `new`**

Avoid hiding the global `new` operator.

**Assignment operator**

In `operator=()` assign to all data members and check for assignment to self.

**Float comparison**

Don't check floats for equality or non-equality; check for GT, GE, LT or LE.

```plaintext
if (Abs (theFloat1 - theFloat2) < theTolerance)
{
   DoSome();
}
```

Package *Precision* provides standard values for SI units and widely adopted by existing modeling algorithms:

- *Precision::Confusion()* for lengths in meters;
- *Precision::Angular()* for angles in radians.

as well as definition of infinite values within normal range of double precision:
• Precision::Infinite()
• Precision::IsInfinite()
• Precision::IsPositiveInfinite()
• Precision::IsNegativeInfinite()

Non-indexed iteration

Avoid usage of iteration over non-indexed collections of objects. If such iteration is used, make sure that the result of the algorithm does not depend on the order of iterated items.

Since the order of iteration is unpredictable in case of a non-indexed collection of objects, it frequently leads to different behavior of the application from one run to another, thus embarrassing the debugging process.

It mostly concerns mapped objects for which pointers are involved in calculating the hash function. For example, the hash function of TopoDS_Shape involves the address of TopoDS_TShape object. Thus the order of the same shape in the TopTools_MapOfShape will vary in different sessions of the application.

Do not throw in destructors

Do not throw from within a destructor.

Assigning to reference [MANDATORY]

Avoid the assignment of a temporary object to a reference. This results in a different behavior for different compilers on different platforms.
Performance issues

These rules define the ways of avoiding possible loss of performance caused by ineffective programming.

Class fields alignment

Declare fields of a class in the decreasing order of their size for better alignment. Generally, try to reduce misaligned accesses since they impact the performance (for example, on Intel machines).

Fields initialization order [MANDATORY]

List class data members in the constructor's initialization list in the order they are declared.

```cpp
class MyPackage_MyClass
{

public:
    MyPackage_MyClass()
    : myPropertyA (1),
        myPropertyB (2) {}

    // NOT
    // : myPropertyB (2),
    //    myPropertyA (1) {}

private:
    Standard_Integer myPropertyA;
    Standard_Integer myPropertyB;
}
```
Initialization over assignment

Prefer initialization over assignment in class constructors.

```cpp
MyPackage_MyClass()
: myPropertyA (1) // preferred
{
    myPropertyB = 2; // not recommended
}
```

Optimize caching

When programming procedures with extensive memory access, try to optimize them in terms of cache behavior. Here is an example of how the cache behavior can be impacted:

On x86 this code

```cpp
Standard_Real anArray[4096][2];
for (Standard_Integer anIter = 0; anIter < 4096;
    ++anIter)
{
    anArray[anIter][0] = anArray[anIter][1];
}
```

is more efficient then

```cpp
Standard_Real anArray[2][4096];
for (Standard_Integer anIter = 0; anIter < 4096;
    ++anIter)
{
    anArray[0][anIter] = anArray[1][anIter];
}
```

since linear access does not invalidate cache too often.
Draw Harness command

Draw Harness provides TCL interface for OCCT algorithms.

There is no TCL wrapper over OCCT C++ classes, instead interface is provided through the set of TCL commands implemented in C++.

There is a list of common rules which should be followed to implement well-formed Draw Harness command.

Return value

Command should return 0 in most cases even if the executed algorithm has failed. Returning 1 would lead to a TCL exception, thus should be used in case of a command line syntax error and similar issues.

Validate input parameters

Command arguments should be validated before usage. The user should see a human-readable error description instead of a runtime exception from the executed algorithm.

Validate the number of input parameters

Command should warn the user about unknown arguments, including cases when extra parameters have been pushed for the command with a fixed number of arguments.

```cpp
if (theArgsNb != 3) {
    std::cout << "Syntax error - wrong number of arguments!\n";
    return 1;
}

Standard_Integer anArgIter = 1;
Standard_CString aResName = theArgVec[anArgIter++];
```
Message printing

Informative messages should be printed into standard output `std::cout`, whilst command results (if any) – into Draw Interpreter.

Information printed into Draw Interpreter should be well-structured to allow usage in TCL script.

Long list of arguments

Any command with a long list of obligatory parameters should be considered as ill-formed by design. Optional parameters should start with flag name (with '-' prefix) and followed by its values:

```
1 myCommand -flag1 value1 value2 -flag2 value3
```

Arguments parser

- Integer values should be read using `Draw::Atoi()` function.
- Real values should be read using `Draw::Atof()` function.
- Flags names should be checked in case insensitive manner.

Functions `Draw::Atof()` and `Draw::Atoi()` support expressions and read values in C-locale.

```
Standard_Real aPosition[3] = {0.0, 0.0, 0.0};
for (Standard_Integer anArgIter = 1; anArgIter <
```
theArgsNb; ++anArgIter)
{
    Standard_CString anArg = theArgVec[anArgIter];
    TCollection_AsciiString aFlag (anArg);
    aFlag.LowerCase(); //!< for case insensitive comparison
    if (aFlag == "position")
    {
        if ((anArgIt + 3) >= theArgsNb)
        {
            std::cerr << "Wrong syntax at argument '" " << anArg << "'!
";
            return 1;
        }
        aPosition[0] = Draw::Atof (theArgVec[++anArgIt]);
        aPosition[1] = Draw::Atof (theArgVec[++anArgIt]);
        aPosition[2] = Draw::Atof (theArgVec[++anArgIt]);
    }
    else
    {
        std::cout << "Syntax error! Unknown flag '" " << anArg << "'
";
        return 1;
    }
}
Examples

Sample documented class

class Package_Class
{
public:  ///< @name public methods

  ///< Method computes the square value.
  ///< @param theValue the input value
  ///< @return squared value
  Standard_Export Standard_Real Square (const Standard_Real theValue);

private:  ///< @name private methods

  ///< Auxiliary method
  void increment();

private:  ///< @name private fields

  Standard_Integer myCounter;  ///< usage counter
};

#include <Package_Class.hxx>

///
/// @name public methods

/// @name private methods

/// @name private fields
Standard_Real Package_Class::Square (const Standard_Real theValue)
{
    increment();
    return theValue * theValue;
}

//
// function : increment
// purpose :
//

void Package_Class::increment()
{
    ++myCounter;
}

---

TCL script for Draw Harness

```
# show fragments (solids) in shading with different colors
proc DisplayColored {theShape} {
    set aSolids [uplevel #0 explode $theShape]
    set aColorIter 0
    set THE_COLORS {red green blue1 magenta1 yellow cyan1 brown}
    foreach aSolIter $aSolids {
        uplevel #0 vdisplay $aSolIter
        uplevel #0 vsetcolor $aSolIter [lindex $THE_COLORS [expr [incr aColorIter] % [llength $THE_COLORS]]]
        uplevel #0 vsetdispmode $aSolIter 1
        uplevel #0 vsetmaterial $aSolIter
    }
```
plastic

11

12

13

14

15 # load modules
16 pload MODELING VISUALIZATION
17
18 # create boxes
19 box bc 0 0 0 1 1 1
20 box br 1 0 0 1 1 2
21 compound bc br c
22
23 # show fragments (solids) in shading with different colors
24 vinit View1
25 vc当地
26 vaxo
27 vzbufftrihedron
28 DisplayColored c
29 vfit
30 vdump $imagedir/${casename}.png 512 512

GLSL program:

vec3 Ambient; /// Ambient contribution of light sources
vec3 Diffuse; /// Diffuse contribution of light sources
vec3 Specular; /// Specular contribution of light sources

/// Computes illumination from light sources
vec4 ComputeLighting (in vec3 theNormal,
in vec3 theView,
in vec4 thePoint)
{
// clear the light intensity accumulators
Ambient = occlightAmbient.rgb;
Diffuse = vec3 (0.0);
Specular = vec3 (0.0);
vec3 aPoint = thePoint.xyz / thePoint.w;
for (int anIndex = 0; anIndex <
    occLightSourcesCount; ++anIndex)
{
    int aType = occLight_Type (anIndex);
    if (aType == OcclightType_Direct)
    {
        directionalLight (anIndex, theNormal, theView);
    }
    else if (aType == OcclightType_Point)
    {
        pointLight (anIndex, theNormal, theView, aPoint);
    }
}

return vec4 (Ambient, 1.0) *
occFrontMaterial_Ambient()
    + vec4 (Diffuse, 1.0) *
occFrontMaterial_Diffuse()
    + vec4 (Specular, 1.0) *
occFrontMaterial_Specular();

//! Entry point to the Fragment Shader
void main()
{
    gl_FragColor = computeLighting (normalize (Normal),
        normalize (View),
        Position);
}
Contribution Workflow

Table of Contents

- Introduction
  - Use of issue tracker system
  - Access levels
- Standard workflow for an issue
  - General scheme
  - Issue registration
  - Assigning the issue
  - Resolving the issue
    - Requirements to the code modification
    - Providing a test case
    - Updating user and developer guides
    - Submission of change as a Git branch
    - Requirements to the commit message
    - Marking issue as resolved
- Code review
- Testing
- Integration of a solution
Closing an issue
Additional workflow elements
  - Requesting more information or specific action
  - Defining relationships between issues
  - Submission of a change as a patch
  - Updating branches in Git
  - Minor corrections
  - Handling non-reproducible issues
Appendix: Issue attributes
  - Category
  - Severity
  - Status
  - Resolution
Introduction

The purpose of this document is to describe standard workflow for processing contributions to certified version of OCCT.
Use of issue tracker system

Each contribution should have corresponding issue (bug, or feature, or integration request) registered in the MantisBT issue tracker system accessible by URL http://tracker.dev.opencascade.org. The issue is processed according to the described workflow.
Access levels

Access level defines the permissions of the user to view, register and modify issues in the issue tracker. The correspondence of access level and user permissions is defined in the table below.

<table>
<thead>
<tr>
<th>Access level</th>
<th>Granted to</th>
<th>Permissions</th>
<th>Can set statuses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viewer</td>
<td>Everyone (anonymous access)</td>
<td>View public issues only</td>
<td>None</td>
</tr>
<tr>
<td>Updater</td>
<td>Users registered on dev.opencascade.org, in Open CASCADE project</td>
<td>View and comment issues</td>
<td>None</td>
</tr>
<tr>
<td>Reporter</td>
<td>Users registered on dev.opencascade.org, in Community project</td>
<td>View, report, and comment issues</td>
<td>New, Resolved, Feedback</td>
</tr>
<tr>
<td>Developer</td>
<td>OCC developers and (in Community project) external contributors who signed the CLA</td>
<td>View, report, modify, and handle issues</td>
<td>New, Assigned, Resolved, Reviewed, Feedback</td>
</tr>
<tr>
<td>Tester</td>
<td>OCC engineer devoted to certification testing</td>
<td>View, report, modify, and handle issues</td>
<td>Assigned, Tested, Feedback</td>
</tr>
<tr>
<td>Maintainer</td>
<td>Person responsible for a project or OCCT component</td>
<td>View, report, modify, and handle issues</td>
<td>New, Resolved, Reviewed, Tested, Closed, Feedback</td>
</tr>
<tr>
<td>Bugmaster</td>
<td>Person responsible for Mantis issue tracker, integrations, certification, and releases</td>
<td>Full access</td>
<td>All statuses</td>
</tr>
</tbody>
</table>

According to his access level, the user can participate in the issue
handling process under different roles, as described below.
Standard workflow for an issue

General scheme

Reporter registers an issue

New

Feedback

Report provides more info

Not clear: more info requested

Maintainer checks the description

YES, bug

Assigned

Developer resolves the issue

Resolved

Reviewer verifies the solution

Good? 

YES, code is good

Reviewed

Tester verifies the solution

Fixed? 

YES, fixed, no regressions

Tested

Conflict with other change is detected

Integrator merges the fix to trunk

Integration to the trunk Verified

Reporter is not satisfied with the fix

Conflict? 

YES, fixed

Reporter can re-check the fix

OK?

YES, fixed

Delivery of the release Closed + Resolution

YES, fix provided
Standard life cycle of an issue
**Issue registration**

An issue is registered in Mantis bugtracker by the Reporter with definition of the necessary attributes (see also Appendix: Issue attributes):

**Category** – indicates the OCCT component, to which the issue relates. (If in doubt, assign to OCCT:Foundation Classes.)

**Severity** – indicates the impact of the issue in the context where it was discovered.

**Profile** – specifies the configuration, on which the problem was detected. For specific configurations it is possible to specify separately platform, OS, and version. These fields can be left empty if the issue is not configuration-specific. Additional details relevant for the environment where the issue is reproduced (such as compiler version, bitness, etc.) can be provided in the Description.

**Products Version** – defines the OCCT version, on which the problem has been detected.

It is preferable to indicate the version of the earliest known official release where the problem can be reproduced. If the issue is reported on the current development version of OCCT, the current development version should be used (for convenience, this version is marked by asterisk in Mantis).

**Note**

OCCT version number can be consulted in the file Standard_Version.hxx (value of OCC_VERSION_COMPLETE macro).

**Assign to** – developer to whom the issue will be assigned. By default, it is set to Maintainer of the OCCT component selected in Category field.

**Target Version** – defines the target version for the fix to be provided. By default, it is set to the current version under development.

**Summary** – a short, one sentence description of the issue.
The **Summary** has a limit of 128 characters. It should be informative and useful for the developers. It is not allowed to mention the issue originator, and in particular the customer, in the name of the registered issue.

A good practice is to start the issue with indication of the relevant component (OCCT module, package, class etc.) to better represent its context.

The summary should be given in imperative mood when it can be formulated as goal to be achieved or action to be done. In particular, this applies to feature requests and improvements, for instance:

| Visualization - provide a support of zoom persistent selection |

If the issue reports a problem, the summary should be given in Present Simple. If reported problem is believed to be a regression, it is recommended to indicate this in the summary, like this:

| [Regression in 6.9.0] IGES - Export of a reversed face leads to wrong data |

**Description** – should contain a detailed definition of the nature of the registered issue depending on its type.

For a bug it is required to submit a detailed description of the incorrect behavior, including the indication of the cause of the problem (if known at this stage), and details on the context where the issue has been detected.

For a feature or integration request it is necessary to describe the proposed feature in details (as much as possible at that stage), including the changes required for its implementation and the main features of the new functionality.

Example:

| Currently selection does not work correctly for non-zoomable objects (those defined using transform persistence). To provide correct selection for such objects, first-level (object) BVH structures must be updated on each camera change, and frustum must be rebuilt accordingly. |
Note

In the description and notes to the issues you can refer to another issue by its ID prefixed by number sign (e.g.: #12345), and refer to a note by its ID prefixed by tilde (e.g.: ~20123). These references will be expanded by Mantis into links to the corresponding issue or note. When the number sign or the tilde followed by digits are a part of a normal text, add a space before digits (e.g.: "face # 12345 contains ~1000 edges") to avoid this conversion.

Steps To Reproduce – allows describing in detail how to reproduce the issue.

This information is crucial for the developer to investigate the cause of the problem and to create the test case. The optimal approach is to give a sequence of DRAW Test Harness commands to reproduce the problem in DRAW. This information can also be provided as a DRAW Tcl script attached to the issue (in Upload File field).

Additional information and documentation updates – any additional information, remarks to be taken into account in Release Notes, etc..

Upload File – allows attaching the shapes, snapshots, scripts, documents, or modified source files of OCCT.

This field can be used to attach a prototype test case in form of a Tcl script for DRAW, a C++ code which can be organized in DRAW commands, sample shapes, documents describing proposed change or analysis of the problem, or other data required for reproduction of the issue. Where applicable, pictures demonstrating a problem and/or desired result can be attached.

The newly registered issue gets status NEW and is assigned to the person indicated in the Assign to field.
Assigning the issue

The description of the new issue is checked by the Maintainer and if it is feasible, he may assign the issue to a Developer. Alternatively, any user with Developer access level or higher can assign the issue to himself if he wants to provide a solution.

The recommended way to handle contributions is that the Reporter assigns the issue to himself and provides a solution.

The Maintainer or Bugmaster can close or reassign the issue (in FEEDBACK state) to the Reporter after it has been registered, if its description does not contain sufficient details to reproduce the bug or explain the need of the new feature. That decision shall be documented in the comments to the issue in the Bugtracker.

The assigned issue has status ASSIGNED.
Resolving the issue

The Developer responsible for the issue assigned to him provides a solution including:

- Changes in the code, with appropriate comments;
- Test case (when applicable) and data necessary for its execution;
- Changes in the user and developer guides (when necessary).

The change is integrated to branch named CRxxxxx (where xxxxx is issue number) in the OCCT Git repository, based on current master, and containing a single commit with the appropriate description. Then the issue is switched to RESOLVED for further review and testing.

The following sub-sections describe this process, relevant requirements and options, in more details.

Requirements to the code modification

The amount of code affected by the change should be limited to the changes required for the bug fix or improvement. Change of layout or re-formatting of the existing code is allowed only in the parts where meaningful changes related to the issue have been made.

Note

If deemed useful, re-formatting or cosmetic changes affecting considerable parts of the code can be made within a dedicated issue.

The changes should comply with the OCCT Coding Rules. It is especially important to comment the code properly so that other people can understand it easier.

The modification should be tested by running OCCT tests (on the platform and scope available to Developer) and ensuring absence of regressions. In case if modification affects results of some existing test case and the new result is correct, such test case should be updated to report OK (or BAD), as described in Automated Test System / Interpretation of Test Results.
Providing a test case

For modifications affecting OCCT functionality, a test case should be created (unless already exists) and included in the commit or patch. See Automated Test System / Creating a New Test for relevant instructions.

The data files required for a test case should be attached to the corresponding issue in Mantis (i.e. not included in the commit).

When the test case cannot be provided for any reason, the maximum possible information on how the problem can be reproduced and how to check the fix should be provided in the Steps to Reproduce field of an issue.

Updating user and developer guides

If the change affects a functionality described in User Guides, the corresponding user guide should be updated to reflect the change.

If the change affects OCCT test system, build environment, or development tools described in Developer Guides, the corresponding guide should be updated.

The changes that break compatibility with the previous versions of OCCT (i.e. affecting API or behavior of existing functionality in the way that may require update of existing applications based on an earlier official release of OCCT to work correctly) should be described in the document Upgrade from previous OCCT versions. It is recommended to add a sub-section for each change described. The description should provide the explanation of the incompatibility introduced by the change, and describe how it can be resolved (at least, in known situations). When feasible, the automatic upgrade procedure (adm/upgrade.tcl) can be extended by a new option to perform the required upgrade of the dependent code automatically.

Submission of change as a Git branch

The modification of sources should be provided in the dedicated branch of the official OCCT Git repository.
The branch should contain a single commit, with the appropriate commit message (see Requirements to the commit message below).

In general, this branch should be based on the recent version of the master branch. It is highly preferable to submit changes basing on the current master. In case if the fix is implemented on the previous release of OCCT, the branch can be based on the corresponding tag in Git, instead of the master.

The branch name should be composed of letters CR (abbreviation of "Change Request") followed by the issue ID number (without leading zeros). It is possible to add an optional suffix to the branch name after the issue ID, e.g. to distinguish between several versions of the fix (see Updating branches in Git).

See Guide to using Git for help.

Note
When a branch with the name given according to the above rule is pushed to Git, a note is automatically added to the corresponding issue in Mantis, indicating the person who has made the push, the commit hash, and (for new commits) the description.

Requirements to the commit message

The commit message posted in Git constitutes an integral part of both the fix and the release documentation.

The first line of the commit message should contain the Summary of the issue (starting with its ID followed by colon, e.g. "0022943: Bug in TDataXtd_PatternStd"), followed by an empty line.

The following lines should provide a description of the context and details on the changes made. The contents and the recommended structure of the description depend on the nature of the bug.

In a general case, the following elements should be present:

- **Problem** – a description of the unwanted behavior;
- **Change** – a description of the implemented changes, including the names of involved classes / methods / enumerations etc.;
• **Result** – a description of the current behavior (after the implementation).

Example:

<table>
<thead>
<tr>
<th>0026330: BRepOffsetAPI_ThruSections creates invalid shape.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods BRep_Tool::CurveOnSurface() and</td>
</tr>
<tr>
<td>BRepCheck_Edge::InContext() now properly handle parametric</td>
</tr>
<tr>
<td>range on a 3D curve when it is used to generate a p-curve</td>
</tr>
<tr>
<td>dynamically (on a planar surface) and both the surface and</td>
</tr>
<tr>
<td>the 3D curve have non-null locations.</td>
</tr>
</tbody>
</table>

Provide sufficient context so that potential user of the affected functionality can understand what has been changed and how the algorithm works now. Describe reason and essence of the changes made, but do not go too deep into implementation details – these should be reflected in comments in the code.

**Marking issue as resolved**

To mark the change as ready for review and testing, the corresponding issue should be switched to **RESOLVED** state. By default, the issue gets assigned to the **Maintainer** of the component, who is thus responsible for its review. Alternatively, another person can be selected as a reviewer at this step.

When the issue is switched to **RESOLVED**, it is required to update or fill the field **Steps to reproduce**. The possible variants are:

- The name of an existing or new test case (preferred variant);
- A sequence of DRAW commands;
- N/A (Not required / Not possible / Not applicable);
- Reference to an issue in the bug tracker of another project.
Code review

The **Reviewer** analyzes the proposed solution for applicability in accordance with OCCT **Coding Rules** and examines all changes in the sources, test case(s), and documentation to detect obvious and possible errors, misprints, or violations of the coding style.

If the Reviewer detects some problems, he can either:

- Fix these issues and provide a new solution. The issue can then be switched to **REVIEWED**.

  In case of doubt or possible disagreement the **Reviewer** can reassign the issue (in **RESOLVED** state) to the **Developer**, who then becomes a **Reviewer**. Possible disagreements should be resolved through discussion, which is done normally within issue notes (or on the OCCT developer’s forum if necessary).

- Reassign the issue back to the **Developer**, providing detailed list of remarks. The issue then gets status **ASSIGNED** and a new solution should be provided.

If Reviewer does not detect any problems, or provides a corrected version, he changes status to **REVIEWED**. The issue gets assigned to the **Bugmaster**.
Testing

The issues that are in REVIEWED state are subject of certification (non-regression) testing. The issue is assigned to an OCCT Tester when he starts processing it.

If the branch submitted for testing is based on obsolete status of the master branch, Tester rebases it on master HEAD. In case of conflicts, the issue is assigned back to Developer in FEEDBACK status, requesting for a rebase.

Certification testing includes:

- Addition of new data models (if required for a new test case) to the data base;
- Revision of the new test case(s) added by developer, and changes in the existing test cases included in commit. The Tester can amend tests to ensure their correct behavior in the certification environment.
- Building OCCT on a sub-set of supported configurations (OS and compiler), watching for errors and warnings;
- Execution of tests on sub-set of supported platforms (at least, one Windows and one Linux configuration), watching for regressions;
- Building OCCT samples, watching for errors;
- Building and testing of OCC products based on OCCT.

If the Tester does not detect problems or regressions, he changes the status to TESTED for further integration.

If the Tester detects build problems or regressions, he changes the status to ASSIGNED and reassigns the issue to the Developer with a detailed description of the problems. The Developer should analyze the reported problems and, depending on results of this analysis, either:

- Confirm that the detected problems are expected changes and they should be accepted as a new status of the code. Then the issue should be switched to FEEDBACK and assigned to the Bugmaster.
- Produce a new solution (see Resolving the issue, and also Minor corrections).
Integration of a solution

Before integration into the master branch of the repository the **Integrator** checks the following conditions:

- the change has been reviewed;
- the change has been tested without regressions (or with regressions treated properly);
- the test case has been created for this issue (when applicable), and the change has been rechecked on this test case;
- the change does not conflict with other changes integrated previously.

If the result of check is successful the Integrator integrates the solution into the branch. The integrations are performed weekly; integration branches are named following the pattern IR-YYYY-MM-DD.

Each change is integrated as a single commit without preserving the history of changes made in the branch (by rebase, squashing all intermediate commits if any), however, preserving the author when possible. This is done to have the master branch history plain and clean. The following picture illustrates the process:

Integration of several branches

The new integration branch is tested against possible regressions that might appear due to interference between separate changes. When the tests are OK, the integration branch is pushed as the new master to the official repository. The issue status is set then to **VERIFIED** and is assigned to the **Reporter** so that he could check the fix as integrated.
The branches corresponding to the integrated fixes are removed from the repository by the **Bugmaster**.
Closing an issue

When possible, the **Reporter** should check whether the problem is actually resolved in the environment where it has been discovered, after the fix is integrated to master. If the fix does not actually resolve the original problem, the issue in **VERIFIED** status can be reopened and assigned back to the **Developer** for rework. The details on how to check that the issue is still reproducible should be provided. However, if the issue does resolve the problem as described in the original report, but a similar problem is discovered for another input data or configuration, or the fix has caused a regression, that problem should be registered as a separate (**related**) issue.

If the fix integrated to master causes regressions, **Bugmaster** can revert it and reopen the issue.

The **Bugmaster** closes the issue after the regular OCCT Release, provided that the issue status is **VERIFIED** and the change was actually included in the release. The final issue state is **CLOSED**.

The field **Fixed in Version** of the issue is set to the OCCT version where it is fixed.
Additional workflow elements

Requesting more information or specific action

If, at any step of the issue lifetime, the person responsible for it cannot process it due to absence of required information, expertise, or rights, he can switch it to status FEEDBACK and assign to the person who is (presumably) able to resolve the block. Some examples of typical situations where FEEDBACK is used are:

- The Maintainer or the Developer requests for more information from the Reporter to reproduce the issue;
- The Tester requests the Developer or the Maintainer to help him in the interpretation of testing results;
- The Developer or the Maintainer asks the Bugmaster to close the issue that is found irrelevant or already fixed (see Handling non-reproducible issues).

In general, issues with status FEEDBACK should be processed as fast as possible, to avoid unjustified delays.
Defining relationships between issues

When two or more issues are related to each other, this relationship should be reflected in the issue tracker. It is also highly recommended to add a note to explain the relationship. Typical cases of relationships are:

- Issue A is caused by previous fix made for issue B (A is a child of B);
- Issue A describes the same problem as issue B (A is a duplicate of B);
- Issues A and B relate to the same piece of code, functionality etc., in the sense that the fix for one of these issues will affect the other (A is related to B)

When the fix made for one issue also resolves another one, these issues should be marked as related or duplicate. In general, the duplicate issue should have the same status, and, when closed, be marked as fixed in the same OCCT version, as the main one.
Submission of a change as a patch

In some cases (if Git is not accessible for the contributor), external contributions can be submitted as a patch file (generated by `diff` command) or as modified sources attached to the Mantis issue. The OCCT version, for which the patch is generated, should be clearly specified (e.g. as hash code of Git commit if the patch is based on an intermediate state of the master).

Note
   Such contributions should be put to Git by someone else (e.g. the Reviewer), this may cause delay in their processing.
Updating branches in Git

Updates of the existing branch (e.g. taking into account the remarks of the **Reviewer**, or fixing regressions) should be provided as new commits on top of previous state of the branch.

It is allowed to rebase the branch on the new state of the master and push it to the repository under the same name (with `--force` option) provided that the original sequence of commits is preserved.

When a change is squashed into a single commit (e.g. to be submitted for review), it should be pushed into a branch with a different name.

The recommended approach is to add a numeric suffix (index) indicating the version of the change, e.g. "CR12345_5". Usually it is worth keeping a non-squashed branch in Git for reference.

To avoid confusions, the branch corresponding to the latest version of the change should have a greater index.

**Note**

- Make sure to revise the commit message after squashing a branch, to keep it meaningful and comprehensive.
Minor corrections

In some cases review remarks or results of testing require only minor corrections to be done in the branch containing a change. "Minor" implies that the correction does not impact the functionality and does not affect the description of the previously committed change.

As an exception to general single-commit rule, it is allowed to put such minor corrections on top of the existing branch as a separate commit, and re-submit it for further processing in the same branch, without squashing.

Minor commits should have a single-line message starting with #. These messages will be ignored when the branch is squashed at integration.

Typical cases of minor corrections are:

- Amendments of test cases (including those made by the Tester to adjust a test script to a specific platform);
- Trivial corrections of compilation warnings (such as removal of an unused variable);
- Addition or correction of comments or documentation;
- Corrections of code formatting (e.g. reversion of irrelevant formatting changes made in the main commit).
Handling non-reproducible issues

Investigation of each issue starts with reproducing it on current development version (master).

If it cannot be reproduced on the current master, but can be reproduced on one of previous releases (or previous state of the master), it is considered as solved by a change made for another issue. If that "fixing" issue can be identified (e.g. by parsing Git log), it should be set as related to that issue. The issue should be switched to FEEDBACK and assigned to the Bugmaster for further processing.

The Bugmaster decides whether it is necessary to create a test case for that issue, and if so may assign it to the Developer or the Tester to create a test. The issue then follows the standard workflow.

Otherwise, if the issue is fixed in one of previous releases, the Bugmaster closes it setting the appropriate value in Fixed in Version field, or, if the issue is fixed after the last release, switches it to VERIFIED status.

If the issue cannot be reproduced due to an unclear description, missing data, etc., it should be assigned back to the Reporter in FEEDBACK status, requesting for more information. The Reporter should provide additional information within one month; after that time, if no new information is provided, the issue should be closed by the Bugmaster with resolution Unable to reproduce.
## Appendix: Issue attributes

### Category

The category corresponds to the component of OCCT where the issue is found:

<table>
<thead>
<tr>
<th>Category</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCCT:Foundation Classes</td>
<td>Foundation Classes module</td>
</tr>
<tr>
<td>OCCT:Modeling Data</td>
<td>Modeling Data classes</td>
</tr>
<tr>
<td>OCCT:Modeling Algorithms</td>
<td>Modeling Algorithms, except shape healing and meshing</td>
</tr>
<tr>
<td>OCCT:Shape Healing</td>
<td>Shape Healing component (TKShapeHealing)</td>
</tr>
<tr>
<td>OCCT:Mesh</td>
<td>BRepMesh algorithm</td>
</tr>
<tr>
<td>OCCT:Data Exchange</td>
<td>Data Exchange module</td>
</tr>
<tr>
<td>OCCT:Visualization</td>
<td>Visualization module</td>
</tr>
<tr>
<td>OCCT:Application Framework</td>
<td>OCAF</td>
</tr>
<tr>
<td>OCCT:DRAW</td>
<td>DRAW Test Harness</td>
</tr>
<tr>
<td>OCCT:Tests</td>
<td>Automatic Test System</td>
</tr>
<tr>
<td>OCCT:Documentation</td>
<td>Documentation</td>
</tr>
<tr>
<td>OCCT:Coding</td>
<td>General code quality</td>
</tr>
<tr>
<td>OCCT:Configuration</td>
<td>Configuration, build system, etc.</td>
</tr>
<tr>
<td>OCCT:Releases</td>
<td>Official OCCT releases</td>
</tr>
<tr>
<td>Website:Tracker</td>
<td>OCCT Mantis issue tracker, tracker.dev.opencascade.org</td>
</tr>
<tr>
<td>Website:Portal</td>
<td>OCCT development portal, dev.opencascade.org</td>
</tr>
<tr>
<td>Website:Git</td>
<td>OCCT Git repository, git.dev.opencascade.org</td>
</tr>
</tbody>
</table>
Severity

Severity shows at which extent the issue affects the product. The list of used severities is given in the table below in the descending order.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crash</td>
<td>Crash of the application or OS, loss of data</td>
</tr>
<tr>
<td>block</td>
<td>Regression corresponding to the previously delivered official version. Impossible operation of a function on any data with no work-around. Missing function previously requested in software requirements specification. Destroyed data.</td>
</tr>
<tr>
<td>major</td>
<td>Impossible operation of a function with existing work-around. Incorrect operation of a function on a particular dataset. Impossible operation of a function after intentional input of incorrect data. Incorrect behavior of a function after intentional input of incorrect data.</td>
</tr>
<tr>
<td>minor</td>
<td>Incorrect behavior of a function corresponding to the description in software requirements specification. Insufficient performance of a function.</td>
</tr>
<tr>
<td>tweak</td>
<td>Ergonomic inconvenience, need of light updates.</td>
</tr>
<tr>
<td>text</td>
<td>Non-conformance of the program code to the Coding Rules, mistakes and non-functional errors in the source text (e.g. unnecessary variable declarations, missing comments, grammatical errors in user manuals).</td>
</tr>
<tr>
<td>trivial</td>
<td>Cosmetic issues.</td>
</tr>
<tr>
<td>feature</td>
<td>Request for a new feature or improvement.</td>
</tr>
<tr>
<td>integration request</td>
<td>Requested integration of an existing feature into the product.</td>
</tr>
<tr>
<td>just a question</td>
<td>A question to be processed, without need of any changes in the product.</td>
</tr>
</tbody>
</table>
The bug statuses that can be applied to the issues are listed in the table below.

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>A new, just registered issue.</td>
</tr>
<tr>
<td>Acknowledged</td>
<td>Can be used to mark the issue as postponed.</td>
</tr>
<tr>
<td>Confirmed</td>
<td>Can be used to mark the issue as postponed.</td>
</tr>
<tr>
<td>Feedback</td>
<td>The issue requires more information or a specific action.</td>
</tr>
<tr>
<td>Assigned</td>
<td>Assigned to a developer.</td>
</tr>
<tr>
<td>Resolved</td>
<td>The issue has been fixed, and now is waiting for review.</td>
</tr>
<tr>
<td>Reviewed</td>
<td>The issue has been reviewed, and now is waiting for testing (or being tested).</td>
</tr>
<tr>
<td>Tested</td>
<td>The fix has been internally tested by the tester with success on the full non-regression database or its part and a test case has been created for this issue.</td>
</tr>
<tr>
<td>Verified</td>
<td>The fix has been integrated into the master of the corresponding repository</td>
</tr>
<tr>
<td>Closed + resolution</td>
<td>The fix has been integrated to the master. The corresponding test case has been executed successfully. The issue is no longer reproduced.</td>
</tr>
</tbody>
</table>
Resolution

Resolution is set when the bug is closed. "Reopen" resolution is added automatically when the bug is reopened.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>The issue is pending.</td>
</tr>
<tr>
<td>Fixed</td>
<td>The issue has been successfully fixed.</td>
</tr>
<tr>
<td>Reopened</td>
<td>The bug has been reopened because of insufficient fix or regression.</td>
</tr>
<tr>
<td>Unable to reproduce</td>
<td>The bug is not reproduced.</td>
</tr>
<tr>
<td>Not fixable</td>
<td>The bug cannot be fixed because e.g. it is a bug of third party software, OS or hardware limitation, etc.</td>
</tr>
<tr>
<td>Duplicate</td>
<td>The bug for the same issue already exists in the tracker.</td>
</tr>
<tr>
<td>Not a bug</td>
<td>It is a normal behavior in accordance with the specification of the product.</td>
</tr>
<tr>
<td>No change required</td>
<td>The issue didn't require any change of the product, such as a question issue.</td>
</tr>
<tr>
<td>Suspended</td>
<td>The issue is postponed (for Acknowledged status).</td>
</tr>
<tr>
<td>Documentation updated</td>
<td>The documentation has been updated to resolve a misunderstanding causing the issue.</td>
</tr>
<tr>
<td>Won’t fix</td>
<td>It is decided to keep the existing behavior.</td>
</tr>
</tbody>
</table>
Guide to installing and using Git for OCCT development

Table of Contents

- Overview
  - Purpose
  - Git URL
  - Content
  - Short rules of use
  - Version of Git
- Installing Tools for Work with Git
  - Windows platform
    - Installation of Git for Windows
    - Installation and configuration of TortoiseGit
  - Linux platform
- Getting access to the repository
  - Prerequisites
  - How to generate a key
    - Generating key with Putty
    - Generating key with command-line tools
    - Generating key with Git GUI
  - Adding public key in...
your account

↓ Work with repository: developer operations
   ↓ General workflow
   ↓ Cloning official repository
   ↓ Branch creation
   ↓ Branch switching
   ↓ Committing branch changes
   ↓ Pushing branch to the remote repository
   ↓ Synchronizing with remote repository
   ↓ Applying a fix made on older version of OCCT
   ↓ Rebasing with history clean-up

↓ Work with repository: Reviewer operations
   ↓ Review branch changes using GitWeb
   ↓ Review branch changes with TortoiseGit
Overview

Purpose

The purpose of this document is to provide a practical introduction to Git to OCCT developers who are not familiar with this tool and to facilitate the use of the official OCCT Git repository for code contribution to OCCT.

Reading this document does not exempt from the need to learn Git concepts and tools. Please consult a book or manual describing Git to get acquainted with this tool. Many good books on Git can be found at http://git-scm.com/documentation

For the experienced Git users it can be enough to read sections 1 and 3 of this document to start working with the repository.

Please make sure to get familiar with the Contribution Workflow document that describes how Git is used for processing contributions to OCCT.

This and related documents are available at the Resources page of the OCCT development portal at http://dev.opencascade.org/index.php?q=home/resources.
Git URL

URL of the official OCCT source code Git repository (accessed by SSH protocol) is:

```
gitolite@git.dev.opencascade.org:occt
```

or

```
ssh://gitolite@dev.opencascade.org/occt.git
```
Content

The official repository contains:

- The current certified version of OCCT: the "master" branch. This branch is updated by the Bugmaster only. Official OCCT releases are marked by tags.
- Topic branches created by contributors to submit changes for review / testing or for collaborative development. The topic branches should be named by the pattern "CR12345" where 12345 is the ID of the relevant issue registered in Mantis (without leading zeroes), and "CR" stands for "Change Request". The name can have an additional postfix used if more than one branch was created for the same issue.
- Occasionally topic branches with non-standard names can be created by the Bugmaster for special needs.
Short rules of use

The name specified in the user.name field in Git configuration should correspond to your login name on the OCCT development portal. This is important to clearly identify the authorship of commits. (The full real name can be used as well; in this case add the login username in parentheses.)

By default, contributors are allowed to push branches only with the names starting with CR (followed by the relevant Mantis issue ID). Possibility to work with other branches can be enabled by the Bugmaster on request.

The branch is created by the developer in his local repository when the development of a contribution starts. The branch for new developments is to be created from the current master. The branch for integration of patches or developments based on an obsolete version is created from a relevant tag or commit. The branch should be pushed to the official repo only when sharing with other people (for collaborative work or review / testing) is needed.

Rebasing the local branch to the current master is encouraged before the first submission to the official repository. If rebasing was needed after the branch is pushed to the official repo, the rebased branch should have a different name (use suffix).

Integration of contributions that have passed certification testing is made exclusively by the Bugmaster. Normally this is made by rebasing the contribution branch on the current master and squashing it into a single commit. This is made to have the master branch history plain and clean, following the general rule “one issue – one commit”. The description of the commit integrated to the master branch is taken from the Mantis issue (ID, 'Summary', followed by the information from 'Documentation' field if present).

In special cases when it is important to save the commits history in the branch (e.g. in case of a long-term development integration) it can be integrated by merge (no fast-forward).

The authorship of the contribution is respected by preserving the Author
field of the commit when integrating. Branches are removed from the official repository when integrated to the master. The Bugmaster can also remove branches which have no commits during one-month period.

The Bugmaster may ask the developer (normally the one who produced the contribution) to rebase a branch on the current master, in the case if merge conflicts appear during integration.
Version of Git

The repository is tested to work with Git 1.7.6 and above. Please do not use versions below 1.7.1 as they are known to cause troubles.
Installing Tools for Work with Git

Windows platform

Installation of Git for Windows (provided by MSysGit project) is required.

In addition, it is recommended to install TortoiseGit to work with Git on Windows. If you do not install TortoiseGit or any other GUI tool, you can use GitGui and Gitk GUI tools delivered with Git and available on all platforms.

Installation of Git for Windows

Download Git for Windows distributive from https://git-for-windows.github.io/ During the installation:

- Check-in "Windows Explorer integration" options:
  - "Git Bash Here";
  - "Git GUI Here".
- To avoid a mess in your PATH, we recommend selecting "Run Git from Windows Prompt" in the environment settings dialog:
- In "Configuring the line ending conversions" dialog, select "Checkout Windows-style, commit Unix style endings".

Note that by default Git user interface is localized to the system default language. If you prefer to work with the English interface, remove or rename .msg localization file in subdirectories share/git-gui/lib/msgs and share/gitk/lib/msgs of the Git installation directory.

Before the first commit to the OCCT repository, make sure that your User Name in the Git configuration file (file .gitconfig in the $HOME directory) is equal to your username on the OCCT development portal.

Installation and configuration of TortoiseGit

Download TortoiseGit distributive from http://code.google.com/p/tortoisegit/downloads/list. Launch the
installation.

- Select your SSH client. Choose option
  - "OpenSSH, Git default SSH Client" if you prefer to use command-line tools for SSH keys generation, or
  - "TortoisePLink, coming from Putty, integrates with Windows better" if you prefer to use GUI tool (PuttyGen, see 3.2).
- Complete the installation.

TortoiseGit integrates into Windows Explorer, thus it is possible to use context menu in Windows Explorer to access its functionality:

![TortoiseGit Context Menu](image)

Note that if you have installed MSysGit or have Git installed in non-default path, on the first time you use TortoiseGit you may get the
message demanding to define path to Git. In such case, click on **Set MSysGit path** button and add the path to git.exe and path to MigGW libraries in the Settings dialog.

- After the installation select Start -> Programs -> TortoiseGit Settings to configure TortoiseGit.

Select Git->Config to add your user name and Email address to the local `.gitconfig` file

![Git configuration settings](image)

Optionally, you can set up TortoiseGit to use visual diff utility for SVG images used in OCCT documentation. For that, click on item "Diff Viewer" in the Settings dialog, then click button "Advanced..." on the right tab add new record with the following parameters:

- **Extension**: .svg
- **External program**: `<path_to_OCCT>\adm\svgdiff.bat %base %mine %bname %yname`
Linux platform

We assume that Linux users have Git already installed and available in the PATH.

Make sure to configure Git so that the user name is equal to your username on the OCCT development portal, and set SafeCrLf option to true:

> git config --global user.name "Your User Name"
> git config --global user.email your@mail.address
> git config --global your@mail.address
Getting access to the repository

Prerequisites

Access to the repository is granted to the users who have signed the Contributor License Agreement.

The repository is accessed by SSH protocol, thus you need to register your public SSH key on the development portal to get access to the repository.

SSH keys are used for secure authentication of the user when accessing the Git server. Private key is the one stored on the user workstation (optionally encrypted). Open (or public) key is stored in the user account page on the web site. When Git client accesses the remote repository through SSH, it uses this key pair to identify the user and acquire relevant access rights.

Normally when you have Git installed, you should have also SSH client available. On Unix/Linux it is installed by default in the system. On Windows it is typical to have several SSH clients installed; in particular they are included with Cygwin, Git, TortoiseGit.

It is highly recommended to use the tools that come with the chosen Git client for generation of SSH keys. Using incompatible tools (e.g. ssh-keygen.exe from Cygwin for code generation, and TortoiseGit GUI with a default Putty client for connection to server) may lead to authentication problems.
How to generate a key

Generating key with Putty

Use this option if you have installed TortoiseGit (or other GUI Git client on Windows) and have chosen “TortoisePLink” (or other Putty client) as SSH client during installation.

To generate the key with this client, run Puttygen (e.g. from Start menu - > TortoiseGit - > Puttygen), then click Generate and move mouse cursor over the blank area until the key is generated.

After the key is generated, you will see GUI controls to define the public key comment and / or specify the password for the private key protection. When done, save both the public and the private key to the files of your choice (make sure to store your private key in a secure place!).

Copy the public key as shown by Puttygen to the clipboard to add it in
Generating key with command-line tools

Use this option if you work on Linux or if you have chosen “OpenSSH” as SSH client during installation of TortoiseGit (or other Windows tool).

Make sure that you have ssh and ssh-keygen commands in the path. On Windows, you might need to start Git Bash command prompt window.

Use the following command to generate SSH keys:

```
> ssh-keygen -t rsa -C "your@mail.address"
```

The last argument is an optional comment, which can be included with the public key and used to distinguish between different keys (if you have many). The common practice is to put here your mail address or workstation name.

The command will ask you where to store the keys. It is recommended to accept the default path $HOME/.ssh/id_rsa. Just press Enter for that. You will be warned if a key is already present in the specified file; you can either overwrite it by the new one, or stop generation and use the old key.

If you want to be on the safe side, enter password to encrypt the private key. You will be asked to enter this password each time you use that key (e.g. access a remote Git repository), unless you use the tool that caches the key (like TortoiseGit). If you do not want to bother, enter an empty string.

On Windows, make sure to note the complete path to the generated files (the location of your $HOME might be not obvious). Two key files will be created in the specified location (by default in $HOME/ssh/):

- id_rsa – private key
- id_rsa.pub – public key

The content of the public key file (one text line) is the key to be added to the user account on the site (see below).
Generating key with Git GUI

GitGUI (standard GUI interface included with Git) provides the option to either generate the SSH key (if not present yet) or show the existing one. Click Help/Show SSH key and copy the public key content for adding to the user account page (see below).
Adding public key in your account

Log in on the portal http://dev.opencascade.org and click on My account link to the right. If you have a Contributor status, you will see SSH keys tab to the right.

Click on that tab, then click Add a public key, and paste the text of the public key (see above sections on how to generate the key) into the text box.

Click Save to input the key to the system.

Note that a user can have several SSH keys. You can distinguish between these keys by the Title field ID; by default it is taken from SSH key comment. It is typical to use your e-mail address or workstation name for this field; no restrictions are set by the portal.

Please note that some time (5-10 min) is needed for the system to update the configuration after the new key is added. After that time, you can try accessing Git.
Work with repository: developer operations

General workflow

To start working with OCCT source repository, you need to create its clone in your local system. This cloned repository will manage your working copy of the sources and provide you the means to exchange code between your clone and the origin.

In most cases it is sufficient to have one clone of the repository; your working copy will be updated automatically by Git when you switch branches.

The typical development cycle for an issue is as follows:

- Create a new branch for your development, basing on the selected version of the sources (usually the current master) and switch your working copy to it
- Develop and test your change.
- Do as many commits in your branch as you feel convenient; the general recommendation is to commit every stable state (even incomplete), to record the history of your development.
- Push your branch to the repository when your development is complete or when you need to share it with other people (e.g. for review)
- Before the first push, rebase your local branch on the latest master; consider collapsing the history in one commit unless you think the history of your commits is interesting for others. Make sure to provide a good commit message.
- Do not amend the commits that have been already pushed in the remote repository, if you need to rebase your branch, commit the rebased branch under a different name, and remove the old branch.

You can switch to another branch at any moment (unless you have some uncommitted changes in the working copy) and return back to the branch when necessary (e.g. to take into account review remarks). Note that
only the sources that are different between the switched branches will be modified, thus required recompilation should be reasonably small in most cases.
Cloning official repository

Clone the official OCCT repository in one of following ways:

- From command line by command:

  ```bash
  > git clone gitolite@git.dev.opencascade.org:occt <path>
  ```

  where `<path>` is the path to the new folder which will be created for the repository.

- In TortoiseGit: create a new folder, open it and right-click in the Explorer window, then choose **Git Clone** in the context menu:

  ![Git Clone - TortoiseGit](image)

  If you have chosen Putty as SSH client during TortoiseGit installation, check the **Load Putty Key** option and specify the location of the private key file saved by PuttyGen (see 3.2.1). This shall be done for the first time only.

  Note that on the first connection to the repository server you may be requested to enter a password for your private SSH key; further you can get a message that the authenticity of the host cannot be established and will be asked if you want to continue connecting or not. Choose **Yes** to continue. The host’s key will be stored in `$HOME/.ssh/known_hosts` file.
Branch creation

You need to create a branch when you are going to start development of a new change, apply a patch, etc. It is recommended to fetch updates from the remote repository before this operation, to make sure you work with the up-to-date version.

Create a branch from the current master branch unless you need to base your development on a particular version or revision.

In the console:

```
> git checkout -b CR12345 origin/master
```

In TortoiseGit:

- Go to the local copy of the repository.
- Right-click in the Explorer window, then choose Git Create Branch.
- Select **Base On Branch** `remotes/origin/master`.

Check option **Switch to new branch** if you are going to start working with the newly created branch immediately.
Branch switching

If you need to switch to another branch, use Git command checkout for that. In the console:

> git checkout CR12345

In TortoiseGit: right-click in the explorer window and select in the context menu TortoiseGit -> Switch/Checkout.

Note that in order to work with the branch locally you need to set option Create new branch when you checkout the branch from the remote repository for the first time. Option Track stores association between the local branch and the original branch in a remote repository.
Committing branch changes

Commit your changes locally as soon as a stable status of the work is reached. Make sure to review carefully the committed changes beforehand to avoid unintentional commit of a wrong code.

- In the console:

```
> git diff
...
> git commit -a -m "Write meaningful commit message here"
```

Option -a tells the command to automatically include (stage) files that have been modified or deleted, but it will omit the new files that might have been added by you. To commit such new files, you must add (stage) them before commit command.

To find new unstaged files and them to commit, use commands:

```
> git status -s
?? file1.hxx
?? file2.cxx
> git add file1.hxx file2.cxx
```

- In TortoiseGit: right-click in the explorer window and select in the context menu **Git Commit -> CR...:**
Unstaged files will be shown if you check the option 'Show Unversioned Files'. Double-click on each modified file to see the changes to be committed (as a difference vs. the base version).
Pushing branch to the remote repository

When the code developed in your local branch is ready for review, or you need to share it with others, push your local changes to the remote repository.

- In the console:

```
> git push "origin" CR12345:CR12345
```

- In TortoiseGit: right-click in the explorer window and select in the context menu, TortoiseGit -> **Push**

Note that Git forbids pushing a branch if the corresponding remote branch already exists and has some changes, which are not in the history of your local branch. This may happen in different situations:

- You have amended the last commit which is already in the remote
repository. If you are sure that nobody else uses your branch, push again with **Force** option.

- You have rebased your branch, so that now it is completely different from the branch in the remote repository. In this case, push it under a different name (add a suffix):

Then remove the original remote branch so that other people recognize that it has been replaced by the new one. For that, select TortoiseGit -> **Push** again, select an empty line for your local branch name, and enter the name of the branch to be removed in **Remote** field:

- The other developer has committed some changes in the remote branch. In this case, **Pull** changes from the remote repository to have them merged with your version, and push your branch after it is successfully merged.
Synchronizing with remote repository

Maintain your repository synchronized with the remote one and clean unnecessary stuff regularly.

Use Git command `fetch` with option `prune` to get the update of all branches from the remote repository and to clean your local repository from the remote branches that have been deleted.

- In the console:
  ```shell
  > git fetch --prune
  ```
- In TortoiseGit: right-click in the explorer window and select in the context menu **TortoiseGit -> Fetch**. Check in **Prune** check-box.

If the branch you are working with has been changed in the remote repository, use Git command `pull` to get the remote changes and merge them with your local branch.

This operation is required in particular to update your local master branch when the remote master changes.

- In console:
> git pull

- In TortoiseGit: right-click in the explorer window and select in the context menu **TortoiseGit -> Pull**.

Note that the local branches of your repository are the primary place, where your changes are stored until they get integrated to the official version of OCCT (master branch). The branches submitted to official repository are for collaborative work, review, and integration – that repository should not be used for long-term storage of incomplete changes.

Remove the local branches that you do not need any more. Note that you cannot delete the current branch. It means that you need to switch to another one (e.g. master) if the branch you are going to delete is the current one.

- In the console:
  
  > git branch -d CR12345

- In TortoiseGit: right-click in the explorer window and select in the context menu **TortoiseGit -> Git Show Log**.
Select **All branches** check-box to view all branches. Right-click on the branch you want to delete and select **Delete** item in the context menu.

Note that many functions described above can be accessed from the Log View, which is a very convenient tool to visualize and manage branches.
Applying a fix made on older version of OCCT

If you have a fix made on a previous version of OCCT, perform the following sequence of operations to prepare it for testing and integration to the current development version:

- Identify the version of OCCT on which the fix has been made. In most cases, this will be an OCCT release, e.g. OCCT 6.7.0.
- Find a tag or a commit corresponding to this version in the Git history log of the master branch.
- Create a branch basing on this tag or commit. In TortoiseGit history log: right-click on the base commit, then select **Create branch at this version**.

- Check option **Switch to the new branch** to start working within the new branch immediately, or switch to it separately afterwards.
- Put your fix in the working copy, build and check that it works, then commit to the branch.
- Rebase the branch on the current master. In TortoiseGit: right-click on the working directory, choose **TortoiseGit -> Rebase**, select `remotes/origin/master` as UpStream revision, and click **Start**.
Note that you can get some conflicts during rebase. To resolve them, double-click on each conflicted file (highlighted by red in the file list) to open visual merge tool. Switch between conflicting fragments by red arrows, and for each one decide if the code of one or both conflicting versions is to be taken.
Rebasing with history clean-up

At some moments you might need to rebase your branch on the latest version of the master.

We recommend rebasing before the first submission of the branch for review or when the master has diverged substantially from your branch.

Rebasing is a good occasion to clean-up the history of commits in the branch. Consider collapsing (squashing, in terms of Git) the history of your branch into a single commit unless you deem that having separate commits is important for your future work with the branch or its code reviewing. Git also allows changing the order of commits, edit commit contents and messages, etc.

To rebase your branch into a single commit, you need to do the following:

- Switch to your branch (e.g. “CR12345”)
- In TortoiseGit history log, select a branch to rebase on (remotes/origin/master) and in the context menu choose **Rebase “CR12345” onto this**.
- In the **Rebase** dialog, check **Squash All**. You can also change the order of commits and define for each commit whether it should be kept (Pick), edited, or just skipped.
- Click Start.
- The process will stop if a conflict is detected. In that case, find files with status **Conflicted** in the list (marked by red), and double-click on them to resolve the conflict. When all conflicts are resolved, click **Continue**.

- At the end of the process, edit the final commit message (it should start from the issue ID and a description from Mantis in the first line, followed by a summary of actual changes), and click **Commit**.
Work with repository: Reviewer operations

Review branch changes using GitWeb

The changes made in the branch can be reviewed without direct access to Git, using GitWeb interface:

- Open GitWeb in your web browser: http://git.dev.opencascade.org/gitweb/?p=occt.git
- Locate the branch you want to review among heads (click ‘…’ at the bottom of the page to see the full list).
- Click log (or shortlog) to see the history of the branch.

Note that the branch can contain more than one commit, and you need to distinguish commits that belong to that branch (those to be reviewed) from the commits corresponding to the previous state of the master branch. Normally the first commit in the list that starts from the ID of the other issue indicates the branching point; commits above it are the ones to be reviewed.

- Click commitdiff on each log entry to review the changes (highlighted with color format).
Review branch changes with TortoiseGit

Use of TortoiseGit is recommended for convenient code review:

- Fetch the changes from the remote repository as described in **Synchronizing with remote repository** section.
- Right-click on the repository, choose **TortoiseGit** -> **Show log**;
- Locate the remote branch you need to review;
- To review commits one-by-one, select each commit in the log. The list of changed files is shown at the bottom of the window; double-click on the file will open visual compare tool.
- To review all changes made in the branch at once, or to compare two arbitrary revisions, select the corresponding commits in the log (e.g. the last commit in the branch and the branching point), ight-click for the context menu, and choose **Compare revisions**.

![TortoiseGit interface](image)
Automated Testing System

Table of Contents

- Introduction
  - Basic Information
  - Intended Use of Automatic Tests
- Quick Start
  - Setup
  - Running Tests
  - Running a Single Test
  - Creating a New Test
- Organization of Test Scripts
  - General Layout
  - Test Groups
    - Group Names
    - File "grids.list"
    - File "begin"
    - File "end"
    - File "parse.rules"
    - Directory "data"
- Test Grids
  - Grid Names
  - File "begin"
  - File "end"
  - File "cases.list"
  - Directory "data"
Test Cases

Creation And Modification Of Tests
  Choosing Group, Grid, and Test Case Name
  Adding Data Files Required for a Test
  Adding new DRAW commands
  Script Implementation
  Interpretation of test results
  Marking BAD cases
  Marking required output

Advanced Use
  Running Tests on Older Versions of OCCT
  Adding custom tests
  Parallel execution of tests
  Checking non-regression of performance, memory, and visualization

APPENDIX
  Test groups
    3rdparty
    blend
    boolean
    bugs
    caf
    chamfer
    demo
    draft
    feat
Mapping of OCCT functionality to grid names in group *bugs*

Recommended approaches to checking test results

- Shape validity
- Shape tolerance
- Shape volume, area, or length
- Memory leaks
- Visualization
- Number of free edges
- Compare numbers
- Check number of sub-shapes
- Check pixel color
- Compute length, area and volume of input shape
- Parse output dump and compare it with reference values
- Compute length of input curve
- Check maximum deflection, number of triangles and nodes in mesh
Introduction

This document provides OCCT developers and contributors with an overview and practical guidelines for work with OCCT automatic testing system.

Reading the Introduction should be sufficient for developers to use the test system to control non-regression of the modifications they implement in OCCT. Other sections provide a more in-depth description of the test system, required for modifying the tests and adding new test cases.
Basic Information

OCCT automatic testing system is organized around DRAW Test Harness, a console application based on Tcl (a scripting language) interpreter extended by OCCT-related commands.

Standard OCCT tests are included with OCCT sources and are located in subdirectory tests of the OCCT root folder. Other test folders can be included in the test system, e.g. for testing applications based on OCCT.

The tests are organized in three levels:

- Group: a group of related test grids, usually testing a particular OCCT functionality (e.g. blend);
- Grid: a set of test cases within a group, usually aimed at testing some particular aspect or mode of execution of the relevant functionality (e.g. buildenvol);
- Test case: a script implementing an individual test (e.g. K4).

See Test Groups chapter for the current list of available test groups and grids.

Note

Many tests involve data files (typically CAD models) which are located separately and (except a few) are not included with OCCT code. These tests will be skipped if data files are not available.
Intended Use of Automatic Tests

Each modification made in OCCT code must be checked for non-regression by running the whole set of tests. The developer who makes the modification is responsible for running and ensuring non-regression for the tests available to him.

Note that many tests are based on data files that are confidential and thus available only at OPEN CASCADE. The official certification testing of each change before its integration to master branch of official OCCT Git repository (and finally to the official release) is performed by OPEN CASCADE to ensure non-regression on all existing test cases and supported platforms.

Each new non-trivial modification (improvement, bug fix, new feature) in OCCT should be accompanied by a relevant test case suitable for verifying that modification. This test case is to be added by the developer who provides the modification.

If a modification affects the result of an existing test case, either the modification should be corrected (if it causes regression) or the affected test cases should be updated to account for the modification.

The modifications made in the OCCT code and related test scripts should be included in the same integration to the master branch.
Quick Start

Setup

Before running tests, make sure to define environment variable
CSF_TestDataPath pointing to the directory containing test data files.

For this it is recommended to add a file DrawAppliInit in the directory
which is current at the moment of starting DRAWEXE (normally it is
OCCT root directory, $CASROOT ). This file is evaluated automatically at
the DRAWE start.

Example (Windows)

```
1| set env(CSF_TestDataPath)
$env(CSF_TestDataPath);d:/occt/test-data
```

Note that variable CSF_TestDataPath is set to default value at DRAWE
start, pointing at the folder $CASROOT/data. In this example,
subdirectory d:/occt/test-data is added to this path. Similar code could be
used on Linux and Mac OS X except that on non-Windows platforms
colon ":" should be used as path separator instead of semicolon ";;".

All tests are run from DRAWE command prompt (run draw.bat or draw.sh
to start it).

Running Tests

To run all tests, type command testgrid

Example:

```
Draw[]> testgrid
```

To run only a subset of test cases, give masks for group, grid, and test
case names to be executed. Each argument is a list of file masks
separated with commas or spaces; by default "*" is assumed.

Example:
As the tests progress, the result of each test case is reported. At the end of the log a summary of test cases is output, including the list of detected regressions and improvements, if any.

Example:

<table>
<thead>
<tr>
<th></th>
<th>Tests summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>CASE 3rdparty export A1: OK</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>CASE pipe standard B1: BAD (known problem)</td>
</tr>
<tr>
<td>6</td>
<td>CASE pipe standard C1: OK</td>
</tr>
<tr>
<td>7</td>
<td>No regressions</td>
</tr>
<tr>
<td>8</td>
<td>Total cases: 208 BAD, 31 SKIPPED, 3 IMPROVEMENT, 1791 OK</td>
</tr>
<tr>
<td>9</td>
<td>Elapsed time: 1 Hours 14 Minutes 33.7384512019 Seconds</td>
</tr>
<tr>
<td>10</td>
<td>Detailed logs are saved in D:/occt/results_2012-06-04T0919</td>
</tr>
</tbody>
</table>

The tests are considered as non-regressive if only OK, BAD (i.e. known problem), and SKIPPED (i.e. not executed, typically because of lack of a data file) statuses are reported. See Interpretation of test results for details.

The results and detailed logs of the tests are saved by default to a new subdirectory of the subdirectory results in the current folder, whose name is generated automatically using the current date and time, prefixed by Git branch name (if Git is available and current sources are managed by Git). If necessary, a non-default output directory can be specified using option -outdir followed by a path to the directory. This directory should be new or empty; use option -overwrite to allow writing results in the existing non-empty directory.

Example:

```
Draw[]> testgrid -outdir d:/occt/last_results -
```
In the output directory, a cumulative HTML report `summary.html` provides links to reports on each test case. An additional report in JUnit-style XML format can be output for use in Jenkins or other continuous integration system.

To re-run the test cases, which were detected as regressions on the previous run, option `-regress dirname` should be used. `dirname` is a path to the directory containing the results of the previous run. Only the test cases with `FAILED` and `IMPROVEMENT` statuses will be tested.

Example:

```
Draw[]> testgrid -regress d:/occt/last_results
```

Type `help testgrid` in DRAW prompt to get help on options supported by `testgrid` command:

```
testgrid: Run all tests, or specified group, or one grid
Use: testgrid [groupmask [gridmask [casemask]]] [options...]
Allowed options are:
-parallel N: run N parallel processes (default is number of CPUs, 0 to disable)
-refresh N: save summary logs every N seconds (default 60, minimal 1, 0 to disable)
-outdir dirname: set log directory (should be empty or non-existing)
-overwrite: force writing logs in existing non-empty directory
-xml filename: write XML report for Jenkins (in JUnit-like format)
-beep: play sound signal at the end of the tests
-regress dirname: re-run only a set of tests that have been detected as regressions on the previous run.
```
Here "dirname" is a path to the directory containing the results of the previous run.
Groups, grids, and test cases to be executed can be specified by the list of file masks separated by spaces or commas; default is all (*).

Running a Single Test

To run a single test, type command test followed by names of group, grid, and test case.

Example:

```
1 Draw[1]> test blend simple A1
2 CASE blend simple A1: OK
3 Draw[2]>
```

Note that normally an intermediate output of the script is not shown. The detailed log of the test can be obtained after the test execution by running command "dlog get".

To see intermediate commands and their output during the test execution, add one more argument "echo" at the end of the command line. Note that with this option the log is not collected and summary is not produced.

Type help test in DRAW prompt to get help on options supported by test command:

```
Draw[3]> help test
test: Run specified test case
    Use: test group grid casename [options...]
    Allowed options are:
    -echo: all commands and results are echoed immediately,
            but log is not saved and summary is not produced
```
It is also possible to use "1" instead of "-echo"
If echo is OFF, log is stored in memory and only summary is output (the log can be obtained with command "dlog get")
-outfile filename: set log file (should be non-existing), it is possible to save log file in text file or in html file(with snapshot), for that "filename"
should have ".html" extension
-overwrite: force writing log in existing file
-beep: play sound signal at the end of the test
-errors: show all lines from the log report that are recognized as errors
This key will be ignored if the "-echo" key is already set.

Creating a New Test

The detailed rules of creation of new tests are given in Creation and modification of tests chapter. The following short description covers the most typical situations:

Use prefix bug followed by Mantis issue ID and, if necessary, additional suffixes, for naming the test script, data files, and DRAW commands specific for this test case.

1. If the test requires C++ code, add it as new DRAW command(s) in one of files in QABugs package.
2. Add script(s) for the test case in the subfolder corresponding to the relevant OCCT module of the group bugs ($CASROOT/tests/bugs). See the correspondence map.
3. In the test script:
   - Load all necessary DRAW modules by command pload.
   - Use command locate_data_file to get a path to data files used by test script. (Make sure to have this command not inside catch
statement if it is used.)

- Use DRAW commands to reproduce the tested situation.
- Make sure that in case of failure the test produces a message containing word "Error" or other recognized by the test system as error (add new error patterns in file parse.rules if necessary).
- If the test case reports error due to an existing problem and the fix is not available, add TODO statement for each error to mark it as a known problem. The TODO statements must be specific so as to match the actually generated messages but not all similar errors.
- To check expected output which should be obtained as the test result, add REQUIRED statement for each line of output to mark it as required.
- If the test case produces error messages (contained in parse.rules), which are expected in that test and should not be considered as its failure (e.g. test for checkshape command), add REQUIRED statement for each error to mark it as required output.

4. If the test uses data file(s) that are not yet present in the test database, it is possible to put them to (sub)directory pointed out by CSF_TestDataPath variable for running test. The files should be attached to the Mantis issue corresponding to the tested modification.

5. Check that the test case runs as expected (test for fix: OK with the fix, FAILED without the fix; test for existing problem: BAD), and integrate it to the Git branch created for the issue.

Example:

- Added files:

```
git status -short
A tests/bugs/heal/data/bug210_a.brep
A tests/bugs/heal/data/bug210_b.brep
A tests/bugs/heal/bug210_1
A tests/bugs/heal/bug210_2
```

- Test script

```
1| puts "OCC210 (case 1): Improve FixShape for
```
touching wires

2 restore [locate_data_file bug210_a.brep] a

4 fixshape result a 0.01 0.01

6 checkshape result
Organization of Test Scripts

General Layout

Standard OCCT tests are located in subdirectory tests of the OCCT root folder ($CASROOT).

Additional test folders can be added to the test system by defining environment variable $CSF_TestScriptsPath. This should be list of paths separated by semicolons (::* on Windows or colons (::* on Linux or Mac. Upon DRAW launch, path to tests subfolder of OCCT is added at the end of this variable automatically.

Each test folder is expected to contain:

- Optional file parse.rules defining patterns for interpretation of test results, common for all groups in this folder
- One or several test group directories.

Each group directory contains:

- File grids.list that identifies this test group and defines list of test grids in it.
- Test grids (sub-directories), each containing set of scripts for test cases, and optional files cases.list, parse.rules, begin and end.
- Optional sub-directory data

By convention, names of test groups, grids, and cases should contain no spaces and be lower-case. The names begin, end, data, parse.rules, grids.list and cases.list are reserved.

General layout of test scripts is shown in Figure 1.
Layout of tests folder
Test Groups

Group Names

The names of directories of test groups containing systematic test grids correspond to the functionality tested by each group.

Example:

caf
mesh
offset

Test group *bugs* is used to collect the tests coming from bug reports. Group *demo* collects tests of the test system, DRAW, samples, etc.

File "grids.list"

This test group contains file *grids.list*, which defines an ordered list of grids in this group in the following format:

001 gridname1
002 gridname2
... gridname2
NNN gridnameN

Example:

001 basic
002 advanced

File "begin"

This file is a Tcl script. It is executed before every test in the current group. Usually it loads necessary Draw commands, sets common parameters and defines additional Tcl functions used in test scripts.

Example:
File "end"

This file is a TCL script. It is executed after every test in the current group. Usually it checks the results of script work, makes a snap-shot of the viewer and writes *TEST COMPLETED* to the output.

Note: *TEST COMPLETED* string should be present in the output to indicate that the test is finished without crash.

See **Creation and modification of tests** chapter for more information.

Example:

```tcl
if { [isdraw result] } {
    checkshape result
} else {
    puts "Error: The result shape can not be built"
}
puts "TEST COMPLETED"
```

File "parse.rules"

The test group may contain *parse.rules* file. This file defines patterns used for analysis of the test execution log and deciding the status of the test run.

Each line in the file should specify a status (single word), followed by a regular expression delimited by slashes (*/*) that will be matched against lines in the test output log to check if it corresponds to this status.

The regular expressions should follow **Tcl syntax**, with a special exception that "\b" is considered as word limit (Perl-style), in addition to "\y" used in Tcl.

The rest of the line can contain a comment message, which will be added to the test report when this status is detected.
Example:

<table>
<thead>
<tr>
<th></th>
<th>FAILED /\b[Ee]xception\b/ exception</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FAILED /\bError\b/ error</td>
</tr>
<tr>
<td></td>
<td>SKIPPED /Cannot open file for reading/ data file is missing</td>
</tr>
<tr>
<td></td>
<td>SKIPPED /Could not read file .*, abandon/ data file is missing</td>
</tr>
</tbody>
</table>

Lines starting with a **##** character and blank lines are ignored to allow comments and spacing.

See **Interpretation of test results** chapter for details.

If a line matches several rules, the first one applies. Rules defined in the grid are checked first, then rules in the group, then rules in the test root directory. This allows defining some rules on the grid level with status **IGNORE** to ignore messages that would otherwise be treated as errors due to the group level rules.

Example:

|   | 1 | FAILED /\bFaulty\b/ bad shape |
|   | 2 | IGNORE /^Error [23]d = \[d.-]+/ debug output of blend command |
|   | 3 | IGNORE /^Tcl Exception: tolerance ang : \[d.-]+/ blend failure |

**Directory "data"**

The test group may contain subdirectory *data*, where test scripts shared by different test grids can be put. See also **Directory data**.
Test Grids

Grid Names

The folder of a test group can have several sub-directories (Grid 1... Grid N) defining test grids. Each directory contains a set of related test cases. The name of a directory should correspond to its contents.

Example:

```
caf
  basic
  bugs
  presentation
```

Here `caf` is the name of the test group and `basic, bugs, presentation`, etc. are the names of grids.

File "begin"

This file is a TCL script executed before every test in the current grid.

Usually it sets variables specific for the current grid.

Example:

```
1 set command bopfuse ;# command tested in this grid
```

File "end"

This file is a TCL script executed after every test in current grid.

Usually it executes a specific sequence of commands common for all tests in the grid.

Example:
File "cases.list"

The grid directory can contain an optional file cases.list defining an alternative location of the test cases. This file should contain a single line defining the relative path to the collection of test cases.

Example:

```plaintext
data/simple
```

This option is used for creation of several grids of tests with the same data files and operations but performed with differing parameters. The common scripts are usually located place in the common subdirectory of the test group, `data/simple` for example.

If file `cases.list` exists, the grid directory should not contain any test cases. The specific parameters and pre- and post-processing commands for test execution in this grid should be defined in the files `begin` and `end`.

Directory "data"

The test grid may contain subdirectory `data`, containing data files used in tests (BREP, IGES, STEP, etc.) of this grid.
Test Cases

The test case is a TCL script, which performs some operations using DRAW commands and produces meaningful messages that can be used to check the validity of the result.

Example:

```
1 pcylinder c1 10 20 ;# create first cylinder
2 pcylinder c2 5 20 ;# create second cylinder
3 ttranslate c2 5 0 10 ;# translate second cylinder to x,y,z
4 bsection result c1 c2 ;# create a section of two cylinders
5 checksection result ;# will output error message if result is bad
```

The test case can have any name (except for the reserved names `begin`, `end`, `data`, `cases.list` and `parse.rules`). For systematic grids it is usually a capital English letter followed by a number.

Example:

```
A1
A2
B1
B2
```

Such naming facilitates compact representation of tests execution results in tabular format within HTML reports.
Creation And Modification Of Tests

This section describes how to add new tests and update existing ones.
Choosing Group, Grid, and Test Case Name

The new tests are usually added in the frame of processing issues in OCCT Mantis tracker. Such tests in general should be added to group bugs, in the grid corresponding to the affected OCCT functionality. See Mapping of OCCT functionality to grid names in group bugs.

New grids can be added as necessary to contain tests for the functionality not yet covered by existing test grids. The test case name in the bugs group should be prefixed by the ID of the corresponding issue in Mantis (without leading zeroes) with prefix `bug`. It is recommended to add a suffix providing a hint on the tested situation. If more than one test is added for a bug, they should be distinguished by suffixes; either meaningful or just ordinal numbers.

Example:

```
1 | bug12345_coaxial
2 | bug12345_orthogonal_1
3 | bug12345_orthogonal_2
```

If the new test corresponds to a functionality already covered by the existing systematic test grid (e.g. group `mesh` for `BRepMesh` issues), this test can be added (or moved later by OCC team) to that grid.
Adding Data Files Required for a Test

It is advisable to make self-contained test scripts whenever possible, so as they could be used in the environments where data files are not available. For that simple geometric objects and shapes can be created using DRAW commands in the test script itself.

If the test requires a data file, it should be put to the directory listed in environment variable `CSF_TestDataPath`. Alternatively, it can be put to subdirectory `data` of the test grid. It is recommended to prefix the data file with the corresponding issue id prefixed by `bug`, e.g. `bug12345_face1.brep`, to avoid possible conflicts with names of existing data files.

Note that when the test is integrated to the master branch, OCC team will move the data file to the data files repository, to keep OCCT sources repository clean from data files.

When you prepare a test script, try to minimize the size of involved data model. For instance, if the problem detected on a big shape can be reproduced on a single face extracted from that shape, use only that face in the test.
Adding new DRAW commands

If the test cannot be implemented using available DRAW commands, consider the following possibilities:

- If the existing DRAW command can be extended to enable possibility required for a test in a natural way (e.g. by adding an option to activate a specific mode of the algorithm), this way is recommended. This change should be appropriately documented in a relevant Mantis issue.
- If the new command is needed to access OCCT functionality not exposed to DRAW previously, and this command can be potentially reused (for other tests), it should be added to the package where similar commands are implemented (use `getsource DRAW` command to get the package name). The name and arguments of the new command should be chosen to keep similarity with the existing commands. This change should be documented in a relevant Mantis issue.
- Otherwise the new command implementing the actions needed for this particular test should be added in `QABugs` package. The command name should be formed by the Mantis issue ID prefixed by `bug`, e.g. `bug12345`.

Note that a DRAW command is expected to return 0 in case of a normal completion, and 1 (Tcl exception) if it is incorrectly used (e.g. a wrong number of input arguments). Thus if the new command needs to report a test error, this should be done by outputting an appropriate error message rather than by returning a non-zero value. File names must be encoded in the script rather than in the DRAW command and passed to the DRAW command as an argument.
**Script Implementation**

The test should run commands necessary to perform the tested operations, in general assuming a clean DRAW session. The required DRAW modules should be loaded by `pload` command, if it is not done by `begin` script. The messages produced by commands in a standard output should include identifiable messages on the discovered problems if any.

Usually the script represents a set of commands that a person would run interactively to perform the operation and see its results, with additional comments to explain what happens.

Example:

```
# Simple test of fusing box and sphere
box b 10 10 10
sphere s 5
bfuse result b s
checkshape result
```

Make sure that file `parse.rules` in the grid or group directory contains a regular expression to catch possible messages indicating the failure of the test.

For instance, for catching errors reported by `checkshape` command relevant grids define a rule to recognize its report by the word `Faulty`:

```
FAILED /\bFaulty\b/ bad shape
```

For the messages generated in the script it is recommended to use the word 'Error' in the error message.

Example:

```bash
set expected_length 11
if { [expr $actual_length - $expected_length] > 0.001 } {
    puts "Error: The length of the edge should be"
} else {
    puts "$actual_length
```
At the end, the test script should output *TEST COMPLETED* string to mark a successful completion of the script. This is often done by the *end* script in the grid.

When the test script requires a data file, use Tcl procedure *locate_data_file* to get a path to it, instead of putting the path explicitly. This will allow easy move of the data file from OCCT sources repository to the data files repository without the need to update the test script.

Example:

```tcl
stepread [locate_data_file CAROSKI_COUPELLE.step] a *
```

When the test needs to produce some snapshots or other artefacts, use Tcl variable *imagedir* as the location where such files should be put.

- Command *testgrid* sets this variable to the subdirectory of the results folder corresponding to the grid.
- Command *test* by default creates a dedicated temporary directory in the system temporary folder (normally the one specified by environment variable *TempDir*, *TEMP*, or *TMP*) for each execution, and sets *imagedir* to that location.

However if variable *imagedir* is defined on the top level of Tcl interpreter, command *test* will use it instead of creating a new directory.

Use Tcl variable *casename* to prefix all files produced by the test. This variable is set to the name of the test case.

The test system can recognize an image file (snapshot) and include it in HTML log and differences if its name starts with the name of the test case (use variable *casename*), optionally followed by underscore or dash and arbitrary suffix.

The image format (defined by extension) should be *png*.

Example:
would produce:

A1.png
A1-axo.png
A1-front.png

Note that OCCT must be built with FreImage support to be able to produce usable images.

Other Tcl variables defined during the test execution are:

- **groupname**: name of the test group;
- **gridname**: name of the test grid;
- **dirname**: path to the root directory of the current set of test scripts.

In order to ensure that the test works as expected in different environments, observe the following additional rules:

- Avoid using external commands such as `grep`, `rm`, etc., as these commands can be absent on another system (e.g. on Windows); use facilities provided by Tcl instead.
- Do not put call to `locate_data_file` in catch statement – this can prevent correct interpretation of the missing data file by the test system.
- Do not use commands `decho` and `dlog` in the test script, to avoid interference with use of these commands by the test system.
Interpretation of test results

The result of the test is evaluated by checking its output against patterns defined in the files `parse.rules` of the grid and group.

The OCCT test system recognizes five statuses of the test execution:

- **SKIPPED**: reported if a line matching SKIPPED pattern is found (prior to any FAILED pattern). This indicates that the test cannot be run in the current environment; the most typical case is the absence of the required data file.
- **FAILED**: reported if a line matching pattern with status FAILED is found (unless it is masked by the preceding IGNORE pattern or a TODO or REQUIRED statement), or if message TEST COMPLETED or at least one of REQUIRED patterns is not found. This indicates that the test has produced a bad or unexpected result, and usually means a regression.
- **BAD**: reported if the test script output contains one or several TODO statements and the corresponding number of matching lines in the log. This indicates a known problem. The lines matching TODO statements are not checked against other patterns and thus will not cause a FAILED status.
- **IMPROVEMENT**: reported if the test script output contains a TODO statement for which no corresponding line is found. This is a possible indication of improvement (a known problem has disappeared).
- **OK**: reported if none of the above statuses have been assigned. This means that the test has passed without problems.

Other statuses can be specified in `parse.rules` files, these will be classified as FAILED.

For integration of the change to OCCT repository, all tests should return either OK or BAD status. The new test created for an unsolved problem should return BAD. The new test created for a fixed problem should return FAILED without the fix, and OK with the fix.
Marking BAD cases

If the test produces an invalid result at a certain moment then the corresponding bug should be created in the OCCT issue tracker located at http://tracker.dev.opencascade.org, and the problem should be marked as TODO in the test script.

The following statement should be added to such a test script:

```puts "TODO BugNumber ListOfPlatforms: RegularExpression"
```

Here:

- `BugNumber` is the bug ID in the tracker. For example: #12345.
- `ListOfPlatforms` is a list of platforms, at which the bug is reproduced (Linux, Windows, MacOS, or All). Note that the platform name is custom for the OCCT test system; Use procedure `checkplatform` to get the platform name.

Example:

```Draw[2]> checkplatform
Windows
```

- `RegularExpression` is a regular expression, which should be matched against the line indicating the problem in the script output.

Example:

```puts "TODO #22622 Mandriva2008: Abort .* an exception was raised"
```

The parser checks the test output and if an output line matches the `RegularExpression` then it will be assigned a BAD status instead of FAILED.

A separate TODO line must be added for each output line matching an error expression to mark the test as BAD. If not all TODO messages are
found in the test log, the test will be considered as possible improvement.

To mark the test as BAD for an incomplete case (when the final TEST COMPLETE message is missing) the expression TEST INCOMPLETE should be used instead of the regular expression.

Example:

```ruby
puts "TODO OCC22817 All: exception.+There are no suitable edges"
puts "TODO OCC22817 All: \*\*\* Exception \*\*\*"
puts "TODO OCC22817 All: TEST INCOMPLETE"
```
Marking required output

To check the obtained test output matches the expected results considered correct, add REQUIRED statement for each specific message. For that, the following statement should be added to the corresponding test script:

```
puts "REQUIRED ListOfPlatforms: RegularExpression"
```

Here `ListOfPlatforms` and `RegularExpression` have the same meaning as in TODO statements described above.

The REQUIRED statement can also be used to mask the message that would normally be interpreted as error (according to the rules defined in `parse.rules`) but should not be considered as such within the current test.

Example:

```
puts "REQUIRED Linux: Faulty shapes in variables faulty_1 to faulty_5"
```

This statement notifies test system that errors reported by `checkshape` command are expected in that test case, and test should be considered as OK if this message appears, despite of presence of general rule stating that 'Faulty' signals failure.

If output does not contain required statement, test case will be marked as FAILED.
Advanced Use

Running Tests on Older Versions of OCCT

Sometimes it might be necessary to run tests on the previous versions of OCCT (<= 6.5.4) that do not include this test system. This can be done by adding DRAW configuration file DrawAppliInit in the directory, which is current by the moment of DRAW start-up, to load test commands and to define the necessary environment.

Note: in OCCT 6.5.3, file DrawAppliInit already exists in $CASROOT/src/DrawResources, new commands should be added to this file instead of a new one in the current directory.

For example, let us assume that d:/occt contains an up-to-date version of OCCT sources with tests, and the test data archive is unpacked to d:/test-data):

```bash
set env(CASROOT) d:/occt
set env(CSF_TestScriptsPath) $env(CASROOT)/tests
source $env(CASROOT)/src/DrawResources/TestCommands.tcl
set env(CSF_TestDataPath) $env(CASROOT)/data;d:/test-data
return
```

Note that on older versions of OCCT the tests are run in compatibility mode and thus not all output of the test command can be captured; this can lead to absence of some error messages (can be reported as either a failure or an improvement).
Adding custom tests

You can extend the test system by adding your own tests. For that it is necessary to add paths to the directory where these tests are located, and one or more additional data directories, to the environment variables \textit{CSF\_TestScriptsPath} and \textit{CSF\_TestDataPath}. The recommended way for doing this is using DRAW configuration file \textit{DrawAppliInit} located in the directory which is current by the moment of DRAW start-up.

Use Tcl command \_\textit{path\_separator} to insert a platform-dependent separator to the path list.

For example:

\begin{verbatim}
set env(CSF\_TestScriptsPath) \$env(TestScriptsPath) \\
      \_\textit{path\_separator}d:/MyOCCTProject/tests
set env(CSF\_TestDataPath) \\
      d:/occt/test- \\
      data\_\textit{path\_separator}d:/MyOCCTProject/data
return ;# this is to avoid an echo of the last command above in cout
\end{verbatim}
Parallel execution of tests

For better efficiency, on computers with multiple CPUs the tests can be run in parallel mode. This is default behavior for command testgrid: the tests are executed in parallel processes (their number is equal to the number of CPUs available on the system). In order to change this behavior, use option parallel followed by the number of processes to be used (1 or 0 to run sequentially).

Note that the parallel execution is only possible if Tcl extension package Thread is installed. If this package is not available, testgrid command will output a warning message.
Checking non-regression of performance, memory, and visualization

Some test results are very dependent on the characteristics of the workstation, where they are performed, and thus cannot be checked by comparison with some predefined values. These results can be checked for non-regression (after a change in OCCT code) by comparing them with the results produced by the version without this change. The most typical case is comparing the result obtained in a branch created for integration of a fix (CR***) with the results obtained on the master branch before that change is made.

OCCT test system provides a dedicated command *testdiff* for comparing CPU time of execution, memory usage, and images produced by the tests.

```
testdiff dir1 dir2 [groupname [gridname]]
            [options...]
```

Here *dir1* and *dir2* are directories containing logs of two test runs.

Possible options are:

- `-save <filename>` – saves the resulting log in a specified file ($dir1/diff-$dir2.log by default). HTML log is saved with the same name and extension .html;
- `-status {same|ok|all}` – allows filtering compared cases by their status:
  - *same* – only cases with same status are compared (default);
  - *ok* – only cases with OK status in both logs are compared;
  - *all* – results are compared regardless of status;
- `-verbose <level>` – defines the scope of output data:
  - 1 – outputs only differences;
  - 2 – additionally outputs the list of logs and directories present in one of directories only;
  - 3 – (by default) additionally outputs progress messages;
- `-image [filename]` - compare images and save the resulting log in specified file ($dir1/diffimage-$dir2.log by default)
- `cpu [filename]` - compare overall CPU and save the resulting log in specified file ($dir1/diffcpu-$dir2.log by default)
- `memory [filename]` - compare memory delta and save the resulting log in specified file ($dir1/diffmemory-$dir2.log by default)
- `highlight_percent <value>` - highlight considerable (>value in %) deviations of CPU and memory (default value is 5%)

Example:

```
Draw[]> testdiff results/CR12345-2012-10-10T08:00 results/master-2012-10-09T21:20
```

Particular tests can generate additional data that need to be compared by `testdiff` command. For that, for each parameter to be controlled, the test should produce the line containing keyword "COUNTER* followed by arbitrary name of the parameter, then colon and numeric value of the parameter.

Example of test code:

```
puts "COUNTER Memory heap usage at step 5: [meminfo h]"
```
APPENDIX

Test groups

3rdparty

This group allows testing the interaction of OCCT and 3rdparty products.

DRAW module: VISUALIZATION.

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>export</td>
<td>vexport</td>
<td>export of images to different formats</td>
</tr>
<tr>
<td>fonts</td>
<td>vtrihedron, vcolorscale, vdrawtext</td>
<td>display of fonts</td>
</tr>
</tbody>
</table>

blend

This group allows testing blends (fillets) and related operations.

DRAW module: MODELING.

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>blend</td>
<td>fillets on simple shapes</td>
</tr>
<tr>
<td>complex</td>
<td>blend</td>
<td>fillets on complex shapes, non-trivial geometry</td>
</tr>
<tr>
<td>tolblend_simple</td>
<td>tolblend, blend</td>
<td></td>
</tr>
<tr>
<td>buildvol</td>
<td>buildvol</td>
<td></td>
</tr>
<tr>
<td>tolblend_buildvol</td>
<td>tolblend, buildvol</td>
<td>use of additional command tolblend</td>
</tr>
<tr>
<td>bfuseblend</td>
<td>bfuseblend</td>
<td></td>
</tr>
<tr>
<td>encoderegularity</td>
<td>encoderegularity</td>
<td></td>
</tr>
</tbody>
</table>
boolean

This group allows testing Boolean operations.

DRAW module: MODELING (packages BOPTest and BRepTest).

Grids names are based on name of the command used, with suffixes:

- \_2d – for tests operating with 2d objects (wires, wires, 3d objects, etc.);
- \_simple – for tests operating on simple shapes (boxes, cylinders, toruses, etc.);
- \_complex – for tests dealing with complex shapes.

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcommon_2d</td>
<td>bcommon</td>
<td>Common operation (old algorithm), 2d</td>
</tr>
<tr>
<td>bcommon_complex</td>
<td>bcommon</td>
<td>Common operation (old algorithm), complex shapes</td>
</tr>
<tr>
<td>bcommon_simple</td>
<td>bcommon</td>
<td>Common operation (old algorithm), simple shapes</td>
</tr>
<tr>
<td>bcut_2d</td>
<td>bcut</td>
<td>Cut operation (old algorithm), 2d</td>
</tr>
<tr>
<td>bcut_complex</td>
<td>bcut</td>
<td>Cut operation (old algorithm), complex shapes</td>
</tr>
<tr>
<td>bcut_simple</td>
<td>bcut</td>
<td>Cut operation (old algorithm), simple shapes</td>
</tr>
<tr>
<td>bcutblend</td>
<td>bcutblend</td>
<td></td>
</tr>
<tr>
<td>bfuse_2d</td>
<td>bfuse</td>
<td>Fuse operation (old algorithm), 2d</td>
</tr>
<tr>
<td>bfuse_complex</td>
<td>bfuse</td>
<td>Fuse operation (old algorithm), complex shapes</td>
</tr>
<tr>
<td>bfuse_simple</td>
<td>bfuse</td>
<td>Fuse operation (old algorithm), simple shapes</td>
</tr>
<tr>
<td>bopcommon_2d</td>
<td>bopcommon</td>
<td>Common operation, 2d</td>
</tr>
<tr>
<td>bopcommon_complex</td>
<td>bopcommon</td>
<td>Common operation, complex shapes</td>
</tr>
<tr>
<td>Grid</td>
<td>Commands</td>
<td>Functionality</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>basic</td>
<td></td>
<td>Basic attributes</td>
</tr>
<tr>
<td>bugs</td>
<td></td>
<td>Saving and restoring of document</td>
</tr>
<tr>
<td>driver</td>
<td></td>
<td>OCAF drivers</td>
</tr>
<tr>
<td>named_shape</td>
<td></td>
<td>TNaming_NamedShape attribute</td>
</tr>
<tr>
<td>presentation</td>
<td></td>
<td>AISPresentation attributes</td>
</tr>
</tbody>
</table>

**bugs**

This group allows testing cases coming from Mantis issues.

The grids are organized following OCCT module and category set for the issue in the Mantis tracker. See [Mapping of OCCT functionality to grid names in group bugs](#) chapter for details.

**caf**

This group allows testing OCAF functionality.

DRAW module: OCAFKERNEL.
chamfer

This group allows testing chamfer operations.

DRAW module: MODELING.

The test grid name is constructed depending on the type of the tested chamfers. Additional suffix _complex is used for test cases involving complex geometry (e.g. intersections of edges forming a chamfer); suffix _sequence is used for grids where chamfers are computed sequentially.

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal_dist</td>
<td>Equal distances from edge</td>
<td></td>
</tr>
<tr>
<td>equal_dist_complex</td>
<td>Equal distances from edge,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>complex shapes</td>
<td></td>
</tr>
<tr>
<td>equal_dist_sequence</td>
<td>Equal distances from edge,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential operations</td>
<td></td>
</tr>
<tr>
<td>dist_dist</td>
<td>Two distances from edge</td>
<td></td>
</tr>
<tr>
<td>dist_dist_complex</td>
<td>Two distances from edge,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>complex shapes</td>
<td></td>
</tr>
<tr>
<td>dist_dist_sequence</td>
<td>Two distances from edge,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential operations</td>
<td></td>
</tr>
<tr>
<td>dist_angle</td>
<td>Distance from edge and given</td>
<td></td>
</tr>
<tr>
<td></td>
<td>angle</td>
<td></td>
</tr>
<tr>
<td>dist_angle_complex</td>
<td>Distance from edge and given</td>
<td></td>
</tr>
<tr>
<td></td>
<td>angle</td>
<td></td>
</tr>
<tr>
<td>dist_angle_sequence</td>
<td>Distance from edge and given</td>
<td></td>
</tr>
<tr>
<td></td>
<td>angle</td>
<td></td>
</tr>
</tbody>
</table>

demo

This group allows demonstrating how testing cases are created, and testing DRAW commands and the test system as a whole.
<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw</td>
<td>getsource, restore</td>
<td>Basic DRAW commands</td>
</tr>
<tr>
<td>testsystem</td>
<td></td>
<td>Testing system</td>
</tr>
<tr>
<td>samples</td>
<td></td>
<td>OCCT samples</td>
</tr>
</tbody>
</table>

**draft**

This group allows testing draft operations.

**DRAW module:** MODELING.

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle</td>
<td>depouille</td>
<td>Drafts with angle (inclined walls)</td>
</tr>
</tbody>
</table>

**feat**

This group allows testing creation of features on a shape.

**DRAW module:** MODELING (package BRepTest).

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>featdprism</td>
<td></td>
<td></td>
</tr>
<tr>
<td>featlf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>featprism</td>
<td></td>
<td></td>
</tr>
<tr>
<td>featrevol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>featrf</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**heal**

This group allows testing the functionality provided by ShapeHealing toolkit.

**DRAW module:** XSDRAW
<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fix_shape</td>
<td>fixshape</td>
<td>Shape healing</td>
</tr>
<tr>
<td>fix_gaps</td>
<td>fixwgaps</td>
<td>Fixing gaps between edges on a wire</td>
</tr>
<tr>
<td>same_parameter</td>
<td>sameparameter</td>
<td>Fixing non-sameparameter edges</td>
</tr>
<tr>
<td>same_parameter_locked</td>
<td>sameparameter</td>
<td>Fixing non-sameparameter edges</td>
</tr>
<tr>
<td>fix_face_size</td>
<td>DT_ApplySeq</td>
<td>Removal of small faces</td>
</tr>
<tr>
<td>elementary_to_revolution</td>
<td>DT_ApplySeq</td>
<td>Conversion of elementary surfaces to revolution</td>
</tr>
<tr>
<td>direct_faces</td>
<td>directfaces</td>
<td>Correction of axis of elementary surfaces</td>
</tr>
<tr>
<td>drop_small_edges</td>
<td>fixsmall</td>
<td>Removal of small edges</td>
</tr>
<tr>
<td>split_angle</td>
<td>DT_SplitAngle</td>
<td>Splitting periodic surfaces by angle</td>
</tr>
<tr>
<td>split_angle_advanced</td>
<td>DT_SplitAngle</td>
<td>Splitting periodic surfaces by angle</td>
</tr>
<tr>
<td>split_angle_standard</td>
<td>DT_SplitAngle</td>
<td>Splitting periodic surfaces by angle</td>
</tr>
<tr>
<td>split_closed_faces</td>
<td>DT_ClosedSplit</td>
<td>Splitting of closed faces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conversion of</td>
</tr>
<tr>
<td>surface_to_bspline</td>
<td>DT_ToBspl</td>
<td>surfaces to b-splines</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>surface_to_bezier</td>
<td>DT_ShapeConvert</td>
<td>Conversion of surfaces to bezier</td>
</tr>
<tr>
<td>split_continuity</td>
<td>DT_ShapeDivide</td>
<td>Split surfaces by continuity criterion</td>
</tr>
<tr>
<td>split_continuity_advanced</td>
<td>DT_ShapeDivide</td>
<td>Split surfaces by continuity criterion</td>
</tr>
<tr>
<td>split_continuity_standard</td>
<td>DT_ShapeDivide</td>
<td>Split surfaces by continuity criterion</td>
</tr>
<tr>
<td>surface_to_revolution_advanced</td>
<td>DT_ShapeConvertRev</td>
<td>Convert elementary surfaces to revolutions, complex case</td>
</tr>
<tr>
<td>surface_to_revolution_standard</td>
<td>DT_ShapeConvertRev</td>
<td>Convert elementary surfaces to revolutions, simple cases</td>
</tr>
<tr>
<td>update_tolerance_locked</td>
<td>updatetolerance</td>
<td>Update the tolerance of shape so that toler(face) &lt;= toler(edge) &lt;= toler(vertex)</td>
</tr>
</tbody>
</table>

**mesh**

This group allows testing shape tessellation (*BRepMesh*) and shading.

**DRAW modules:** MODELING (package *MeshTest*), VISUALIZATION (package *ViewerTest*)
<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>advanced_shading</td>
<td>vdisplay</td>
<td>Shading, complex shapes</td>
</tr>
<tr>
<td>standard_shading</td>
<td>vdisplay</td>
<td>Shading, simple shapes</td>
</tr>
<tr>
<td>advanced_mesh</td>
<td>mesh</td>
<td>Meshing of complex shapes</td>
</tr>
<tr>
<td>standard_mesh</td>
<td>mesh</td>
<td>Meshing of simple shapes</td>
</tr>
<tr>
<td>advanced_incmesh</td>
<td>incmesh</td>
<td>Meshing of complex shapes</td>
</tr>
<tr>
<td>standard_incmesh</td>
<td>incmesh</td>
<td>Meshing of simple shapes</td>
</tr>
<tr>
<td>advanced_incmesh_parallel</td>
<td>incmesh</td>
<td>Meshing of complex shapes, parallel mode</td>
</tr>
<tr>
<td>standard_incmesh_parallel</td>
<td>incmesh</td>
<td>Meshing of simple shapes, parallel mode</td>
</tr>
</tbody>
</table>

**mkface**

This group allows testing creation of simple surfaces.

**DRAW module: MODELING (package BRepTest)**

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>after_trim</td>
<td>mkface</td>
<td></td>
</tr>
<tr>
<td>after_offset</td>
<td>mkface</td>
<td></td>
</tr>
<tr>
<td>after_extsurf_and_offset</td>
<td>mkface</td>
<td></td>
</tr>
<tr>
<td>after_extsurf_and_trim</td>
<td>mkface</td>
<td></td>
</tr>
<tr>
<td>after_revsurf_and_offset</td>
<td>mkface</td>
<td></td>
</tr>
<tr>
<td>mkplane</td>
<td>mkplane</td>
<td></td>
</tr>
</tbody>
</table>

**nproject**

This group allows testing normal projection of edges and wires onto a face.
OFFSET

This group allows testing offset functionality for curves and surfaces.

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>compshape</td>
<td>offsetcompshape</td>
<td>Offset of shapes with removal of some faces</td>
</tr>
<tr>
<td>faces_type_a</td>
<td>offsetparameter, offsetload, offsetperform</td>
<td>Offset on a subset of faces with a fillet</td>
</tr>
<tr>
<td>faces_type_i</td>
<td>offsetparameter, offsetload, offsetperform</td>
<td>Offset on a subset of faces with a sharp edge</td>
</tr>
<tr>
<td>shape_type_a</td>
<td>offsetparameter, offsetload, offsetperform</td>
<td>Offset on a whole shape with a fillet</td>
</tr>
<tr>
<td>shape_type_i</td>
<td>offsetparameter, offsetload, offsetperform</td>
<td>Offset on a whole shape with a fillet</td>
</tr>
<tr>
<td>shape</td>
<td>offsetshape</td>
<td></td>
</tr>
<tr>
<td>wire_closed_outside_0_005, wire_closed_outside_0_025, wire_closed_outside_0_075,</td>
<td>mkoffset</td>
<td>2d offset of closed and unclosed planar wires with different offset step and directions of</td>
</tr>
<tr>
<td>wire_closed_inside_0_005, wire_closed_inside_0_025, wire_closed_inside_0_075, wire_unclosed_outside_0_005,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wire_unclosed_inside_0_075</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
pipe

This group allows testing construction of pipes (sweeping of a contour along profile).

DRAW module: MODELING (package BRepTest)

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>pipe</td>
<td></td>
</tr>
</tbody>
</table>

prism

This group allows testing construction of prisms.

DRAW module: MODELING (package BRepTest)

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>seminf</td>
<td>prism</td>
<td></td>
</tr>
</tbody>
</table>

sewing

This group allows testing sewing of faces by connecting edges.

DRAW module: MODELING (package BRepTest)

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>tol_0_01</td>
<td>sewing</td>
<td>Sewing faces with tolerance 0.01</td>
</tr>
<tr>
<td>tol_1</td>
<td>sewing</td>
<td>Sewing faces with tolerance 1</td>
</tr>
<tr>
<td>tol_100</td>
<td>sewing</td>
<td>Sewing faces with tolerance 100</td>
</tr>
</tbody>
</table>

thrussection

This group allows testing construction of shell or a solid passing through
a set of sections in a given sequence (loft).

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>solids</td>
<td>thrusection</td>
<td>Lofting with resulting solid</td>
</tr>
<tr>
<td>not_solids</td>
<td>thrusection</td>
<td>Lofting with resulting shell or face</td>
</tr>
</tbody>
</table>

**xcaf**

This group allows testing extended data exchange packages.

<table>
<thead>
<tr>
<th>Grid</th>
<th>Commands</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dxc, dxc_add_ACL, dxc_add_CL,</td>
<td>Subgroups are divided by format of source file, by format of result file and</td>
</tr>
<tr>
<td></td>
<td>igs_to_dxc, igsl_add_ACL, brep_to_igs_add_CL, stp_to_dxc, stp_add_ACL, brep_to_stp_add_CL, brep_to_dxc, add_ACL_brep, brep_add_CL</td>
<td>by type of document modification. For example, brep_to_igs means that the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>source shape in brep format was added to the document, which was saved into</td>
</tr>
<tr>
<td></td>
<td></td>
<td>igs format after that. The postfix add_CL means that colors and layers were</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialized in the document before saving and the postfix add_ACL corresponds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to the creation of assembly and initialization of colors and layers in a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>document before saving.</td>
</tr>
</tbody>
</table>
## Mapping of OCCT functionality to grid names in group *bugs*

<table>
<thead>
<tr>
<th>OCCT Module / Mantis category</th>
<th>Toolkits</th>
<th>Test grid in group bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Framework</td>
<td>PTKernel, TKPShape, TKCDF, TKLCAF, TKCAF, TKBinL, TKXmIL, TKShapeSchema, TKPLCAF, TKBin, TKXml, TKPCAF, FWOSPlugin, TKStdLSchema, TKStdSchema, TKTObj, TKBinTObj, TKXmITObj</td>
<td>caf</td>
</tr>
<tr>
<td>Draw</td>
<td>TKDraw, TKTopTest, TKViewerTest, TKXSDRAW, TKDCAF, TKXDEDRAW, TKTObjDRAW, TKQADraw, DRAWEXE, Problems of testing system</td>
<td>draw</td>
</tr>
<tr>
<td>Shape Healing</td>
<td>TKShHealing</td>
<td>heal</td>
</tr>
<tr>
<td>Mesh</td>
<td>TKMesh, TKXMesh</td>
<td>mesh</td>
</tr>
<tr>
<td>Data Exchange</td>
<td>TKIGES</td>
<td>iges</td>
</tr>
<tr>
<td>Data Exchange</td>
<td>TKSTEPBase, TKSTEPAttr, TKSTEP209, TKSTEP</td>
<td>step</td>
</tr>
<tr>
<td>Data Exchange</td>
<td>TKSTL, TKVRML</td>
<td>stlvrml</td>
</tr>
<tr>
<td>Data Exchange</td>
<td>TKXSBase, TKXCAF, TKXCAFSchema, TKXDEIGES, TKXDSTEP, TKXmIXCAF, TKBinXCAF</td>
<td>xde</td>
</tr>
<tr>
<td>Foundation Classes</td>
<td>TKernel, TKMath</td>
<td>fclasses</td>
</tr>
<tr>
<td>Modeling_algorithms</td>
<td>TKGeomAlgo, TKTopAlgo, TKPrim, TKBO, TKBool, TKHLR, TKFillet, TKOffset, TKFeat, TKXMesh</td>
<td>modalg</td>
</tr>
<tr>
<td>Modeling Data</td>
<td>TKG2d, TKG3d, TKGeomBase, TKBRep</td>
<td>moddata</td>
</tr>
<tr>
<td>Visualization</td>
<td>TKService, TKV2d, TKV3d, TKOpenGL, TKMeshVS, TKNIS</td>
<td>vis</td>
</tr>
</tbody>
</table>
Recommended approaches to checking test results

Shape validity

Run command `checkshape` on the result (or intermediate) shape and make sure that `parse.rules` of the test grid or group reports bad shapes (usually recognized by word "Faulty") as error.

Example

```
checkshape result
```

To check the number of faults in the shape command `checkfaults` can be used.

Use: `checkfaults shape source_shape [ref_value=0]`

The default syntax of `checkfaults` command:

```
checkfaults results a_1
```

The command will check the number of faults in the source shape (a_1) and compare it with number of faults in the resulting shape (result). If shape `result` contains more faults, you will get an error:

```
checkfaults results a_1
Error : Number of faults is 5
```

It is possible to set the reference value for comparison (reference value is 4):

```
checkfaults results a_1 4
```

If number of faults in the resulting shape is unstable, reference value should be set to "-1". As a result command `checkfaults` will return the following error:
Shape tolerance

The maximal tolerance of sub-shapes of each kind of the resulting shape can be extracted from output of tolerance command as follows:

```
set tolerance [tolerance result]
regexp { *FACE +: +MAX=([-0-9.+eE]+)} $tolerance
dummy max_face
regexp { *EDGE +: +MAX=([-0-9.+eE]+)} $tolerance
dummy max_edgee
regexp { *VERTEX +: +MAX=([-0-9.+eE]+)} $tolerance
dummy max_vertex
```

It is possible to use command `checkmaxtol` to check maximal tolerance of shape and compare it with reference value.

Use: `checkmaxtol` shape [options...]

Allowed options are:

- `-ref` – reference value of maximum tolerance;
- `-source` – list of shapes to compare with;
- `-min_tol` – minimum tolerance for comparison;
- `-multi_tol` – tolerance multiplier.

The default syntax of `checkmaxtol` command for comparison with the reference value:

```
checkmaxtol result -ref 0.00001
```

There is an opportunity to compare max tolerance of resulting shape with max tolerance of source shape. In the following example command `checkmaxtol` gets max tolerance among objects `a_1` and `a_2`. Then it chooses the maximum value between founded tolerance and value `-min_tol` (0.000001) and multiply it on the coefficient `-multi_tol` (i.e. 2):

```
checkmaxtol result -source {a_1 a_2} -min_tol
```
If the value of maximum tolerance more than founded tolerance for comparison, the command will return an error.

Also, command `checkmaxtol` can be used to get max tolerance of the shape:

```
set maxtol [checkmaxtol result]
```

**Shape volume, area, or length**

Use command `vprops`, `sprops`, or `lprops` to correspondingly measure volume, area, or length of the shape produced by the test. The value can be extracted from the result of the command by `regexp`.

Example:

```
# check area of shape result with 1% tolerance
regexp {Mass += +([-0-9.]+eE)+} [sprops result] dummy
if { abs($area - $expected) > 0.1 + 0.01 * abs($area) } {
  puts "Error: The area of result shape is $area, while expected $expected"
}
```

**Memory leaks**

The test system measures the amount of memory used by each test case. Considerable deviations (as well as the overall difference) in comparison with reference results can be reported by command `testdiff` (see Checking non-regression of performance, memory, and visualization).

To check memory leak on a particular operation, run it in a cycle, measure the memory consumption at each step and compare it with a threshold value. The command `checktrend` (defined in tests/bugs/begin) can be used to analyze a sequence of memory measurements and to get
a statistically based evaluation of the leak presence.

Example:

```
set listmem {}
for {set i 1} {$i < 100} {incr i} {
    # run suspect operation
    ...
    # check memory usage (with tolerance equal to half page size)
    lappend listmem [expr [meminfo w] / 1024]
    if { [checktrend $listmem 0 256 "Memory leak detected"] } {
        puts "No memory leak, $i iterations"
        break
    }
}
```

**Visualization**

The following command sequence allows you to take a snapshot of the viewer, give it the name of the test case, and save in the directory indicated by Tcl variable `imagedir`.

```
vinit
vclear
vdisplay result
vsetdispmode 1
vfit
vzfit
vdump $imagedir/${casename}_shading.png
```

This image will be included in the HTML log produced by `testgrid` command and will be checked for non-regression through comparison of images by command `testdiff`.

Also it is possible to use command `checkview` to make a snapshot of the viewer.
Use: checkview [options...] Allowed options are:

- **-display shapename** – displays shape with name *shapename*;
- **-3d** – displays shape in 3d viewer;
- **-2d [ v2d / smallview ]** - displays shape in 2d viewer (the default viewer is *smallview*);
- **-path PATH** – sets the location of the saved viewer screenshot;
- **-vdispmode N** – sets *vdispmode* for 3d viewer (default value is 1)
- **-screenshot** – makes a screenshot of already created viewer
- The procedure can check a property of shape (length, area or volume) and compare it with value *N*:
  - **-l [N]**
  - **-s [N]**
  - **-v [N]**
  - If the current property is equal to value *N*, the shape is marked as valid in the procedure.
  - If value *N* is not given, the procedure will mark the shape as valid if the current property is non-zero.
- **-with \{a b c\}** – displays shapes *a*, *b* and *c* together with the shape (if the shape is valid)
- **-otherwise \{d e f\}** – displays shapes *d*, *e* and *f* instead of the shape (if the shape is NOT valid)

Note that it is required to use either option -2d or option -3d.

Examples:

```
checkview -display result -2d -path
     ${imagedir}/${test_image}.png
checkview -display result -3d -path
     ${imagedir}/${test_image}.png
checkview -display result_2d -2d v2d -path
     ${imagedir}/${test_image}.png
box a 10 10 10
box b 5 5 5 10 10 10
bcut result b a
set result_vertices [explode result v]
checkview -display result -2d -with
     ${result_vertices} -otherwise \{a b\} -l -path
```
Number of free edges

Procedure checkfreebounds compares the number of free edges with a reference value.

Use: checkfreebounds shape ref_value [options...]

Allowed options are:

- `-tol N` – used tolerance (default -0.01);
- `-type N` – used type, possible values are "closed" and "opened" (default "closed").

checkfreebounds result 13

Option `-tol N` defines tolerance for command freebounds, which is used within command checkfreebounds.

Option `-type N` is used to select the type of counted free edges: closed or open.

If the number of free edges in the resulting shape is unstable, the reference value should be set to "-1". As a result command checkfreebounds will return the following error:

checkfreebounds result -1
Error : Number of free edges is UNSTABLE

Compare numbers
Procedure *checkreal* checks the equality of two reals with a tolerance (relative and absolute).

Use: `checkreal name value expected tol_abs tol_rel`

```
checkreal "Some important value" $value 5 0.0001 0.01
```

**Check number of sub-shapes**

Procedure *checknbshapes* compares the number of sub-shapes in "shape" with the given reference data.

Use: `checknbshapes shape [options...]`

Allowed options are:

- `-vertex N`
- `-edge N`
- `-wire N`
- `-face N`
- `-shell N`
- `-solid N`
- `-compsolid N`
- `-compound N`
- `-shape N`
- `-t` – compares the number of sub-shapes in "shape" counting the same sub-shapes with different location as different sub-shapes.
- `-m msg` – prints "msg" in case of error

```
checknbshapes result -vertex 8 -edge 4
```

**Check pixel color**

Command *checkcolor* can be used to check pixel color.

Use: `checkcolor x y red green blue`

where:

- `x, y` – pixel coordinates;
- *red green blue* – expected pixel color (values from 0 to 1).

This procedure checks color with tolerance (5x5 area).

Next example will compare color of point with coordinates x=100 y=100 with RGB color R=1 G=0 B=0. If colors are not equal, procedure will check the nearest ones points (5x5 area)

```
checkcolor 100 100 1 0 0
```

### Compute length, area and volume of input shape

Procedure *checkprops* computes length, area and volume of the input shape.

Use: checkprops shapename [options...]

Allowed options are:

- `-l LENGTH` – command *lprops*, computes the mass properties of all edges in the shape with a linear density of 1;
- `-s AREA` – command *sprops*, computes the mass properties of all faces with a surface density of 1;
- `-v VOLUME` – command *vprops*, computes the mass properties of all solids with a density of 1;
- `-eps EPSILON` – the epsilon defines relative precision of computation;
- `-deps DEPSILON` – the epsilon defines relative precision to compare corresponding values;
- `-equal SHAPE` – compares area, volume and length of input shapes. Puts error if they are not equal;
- `-notequal SHAPE` – compares area, volume and length of input shapes. Puts error if they are equal.

Options `-l`, `-s` and `-v` are independent and can be used in any order. Tolerance *epsilon* is the same for all options.

```
checkprops result -s 6265.68
checkprops result -s -equal FaceBrep
```
Parse output dump and compare it with reference values

Procedure `checkdump` is used to parse output dump and compare it with reference values.

Use: `checkdump shapename [options...]`

Allowed options are:

- `-name NAME` – list of parsing parameters (e.g. Center, Axis, etc.);
- `-ref VALUE` – list of reference values for each parameter in `NAME`;
- `-eps EPSILON` – the epsilon defines relative precision of computation.

```
checkdump result -name {Center Axis XAxis YAxis Radii} -ref {{-70 0} {-1 -0} {-1 -0} {0 -1} {20 10}} -eps 0.01
```

Compute length of input curve

Procedure `checklength` computes length of the input curve.

Use: `checklength curvename [options...]`

Allowed options are:

- `-l LENGTH` – command `length`, computes the length of the input curve with precision of computation;
- `-eps EPSILON` – the epsilon defines a relative precision of computation;
- `-equal CURVE` – compares the length of input curves. Puts error if they are not equal;
- `-notequal CURVE` – compares the length of input curves. Puts error if they are equal.

```
checklength cp1 -l 7.278
checklength res -l -equal ext_1
```

Check maximum deflection, number of triangles and nodes in mesh
Command *checktrinfo* can be used to check the maximum deflection, as well as the number of nodes and triangles in mesh.

Use: checktrinfo shapename [options...]

Allowed options are:

- **-tri [N]** – compares the current number of triangles in *shapename* mesh with the given reference data. If reference value N is not given and the current number of triangles is equal to 0, procedure *checktrinfo* will print an error.
- **-nod [N]** – compares the current number of nodes in *shapename* mesh with the given reference data. If reference value N is not given and the current number of nodes is equal to 0, procedure *checktrinfo* will print an error.
- **-defl [N]** – compares the current value of maximum deflection in *shapename* mesh with the given reference data. If reference value N is not given and current maximum deflection is equal to 0, procedure *checktrinfo* will print an error.
- **-max_defl N** – compares the current value of maximum deflection in *shapename* mesh with the max possible value.
- **-tol_abs_tri N** – absolute tolerance for comparison of number of triangles (default value 0).
- **-tol_rel_tri N** – relative tolerance for comparison of number of triangles (default value 0).
- **-tol_abs_nod N** – absolute tolerance for comparison of number of nodes (default value 0).
- **-tol_rel_nod N** – relative tolerance for comparison of number of nodes (default value 0).
- **-tol_abs_defl N** – absolute tolerance for deflection comparison (default value 0).
- **-tol_rel_defl N** – relative tolerance for deflection comparison (default value 0).
- **-ref [trinfo a]** – compares deflection, number of triangles and nodes in *shapename* and a.

Note that options -tri, -nod and -defl do not work together with option -ref.

Examples:
Comparison with some reference values:

`checktrinfo result -tri 129 -nod 131 -defl 0.01`

Comparison with another mesh:

`checktrinfo result -ref [tringo a]`

Comparison of deflection with the max possible value:

`checktrinfo result -max_defl 1`

Check that the current values are not equal to zero:

`checktrinfo result -tri -nod -defl`

Check that the number of triangles and the number of nodes are not equal to some specific values:

`checktrinfo result -tri !10 -nod !8`

It is possible to compare current values with reference values with some tolerances. Use options `-tol_*` for that:

`checktrinfo result -defl 1 -tol_abs_defl 0.001`
Debugging tools and hints

Table of Contents

- Introduction
- Compiler macro to enable extended debug messages
- Calling JIT debugger on exception
- Self-diagnostics in Boolean operations algorithm
- Functions for calling from debugger
  - Interacting with DRAW
  - Saving and dumping shapes and geometric objects
- Using Visual Studio debugger
  - Command window
  - Customized display of variables content
- Performance measurement tools
Introduction

This manual describes facilities included in OCCT to support debugging, and provides some hints for more efficient debug.
Compiler macro to enable extended debug messages

Many OCCT algorithms can produce extended debug messages, usually printed to cout. These include messages on internal errors and special cases encountered, timing etc. In OCCT versions prior to 6.8.0 most of these messages were activated by compiler macro `DEB`, enabled by default in debug builds. Since version 6.8.0 this is disabled by default but can be enabled by defining compiler macro `OCCT_DEBUG`.

To enable this macro on Windows when building with Visual Studio projects, edit file custom.bat and add the line:

```bash
set CSF_DEFINES=OCCT_DEBUG
```

Some algorithms use specific macros for yet more verbose messages, usually started with `OCCT_DEBUG_`. These messages can be enabled in the same way, by defining corresponding macro.

Note that some header files are modified when `OCCT_DEBUG` is enabled, hence binaries built with it enabled are not compatible with client code built without this option; this is not intended for production use.
Calling JIT debugger on exception

On Windows platform when using Visual Studio compiler there is a possibility to start the debugger automatically if an exception is caught in a program running OCCT. For this, set environment variable `CSF_DEBUG` to any value. Note that this feature works only if you enable OCCT exception handler in your application by calling `OSD::SetSignal()`. 
Self-diagnostics in Boolean operations algorithm

In real-world applications modeling operations are often performed in a long sequence, while the user sees only the final result of the whole sequence. If the final result is wrong, the first debug step is to identify the offending operation to be debugged further. Boolean operation algorithm in OCCT provides a self-diagnostic feature which can help to do that step.

This feature can be activated by defining environment variable `CSF_DEBUG_BOP`, which should specify an existing writeable directory.

The diagnostic code checks validity of the input arguments and the result of each Boolean operation. When an invalid situation is detected, the report consisting of argument shapes and a DRAW script to reproduce the problematic operation is saved to the directory pointed by `CSF_DEBUG_BOP`.

Note that this feature does not applicable for UWP build.
Functions for calling from debugger

Modern interactive debuggers provide the possibility to execute application code at a program break point. This feature can be used to analyse the temporary objects available only in the context of the debugged code. OCCT provides several global functions that can be used in this way.

Note that all these functions accept pointer to variable as $void^*$ to allow calling the function even when debugger does not recognize type equivalence or can not perform necessary type cast automatically. It is responsibility of the developer to provide the correct pointer. In general these functions are not guaranteed to work, thus use them with caution and at your own risk.
Interacting with DRAW

Open CASCADE Test Harness or DRAW provides an extensive set of tools for inspection and analysis of OCCT shapes and geometric objects and is mostly used as environment for prototyping and debugging OCCT-based algorithms.

In some cases the objects to be inspected are available in DRAW as results of DRAW commands. In other cases, however, it is necessary to inspect intermediate objects created by the debugged algorithm. To support this, DRAW provides a set of commands allowing the developer to store intermediate objects directly from the debugger stopped at some point during the program execution (usually at a breakpoint).

```c
const char* Draw_Eval (const char *theCommandStr)
```

Evaluates a DRAW command or script. A command is passed as a string parameter.

```c
const char* DBRep_Set (const char* theNameStr, void* theShapePtr)
```

Sets the specified shape as a value of DRAW interpreter variable with the given name.

- `theNameStr` – the DRAW interpreter variable name to set.
- `theShapePtr` – a pointer to `TopoDS_Shape` variable.

```c
const char* DBRep_SetComp (const char* theNameStr, void* theListPtr)
```

Makes a compound from the specified list of shapes and sets it as a value of DRAW interpreter variable with the given name.

- `theNameStr` – the DRAW interpreter variable name to set.
- `theListPtr` – a pointer to `TopTools_ListOfShape` variable.

```c
const char* DrawTrSurf_Set (const char* theNameStr, void* theHandlePtr)
```
Sets the specified geometric object as a value of DRAW interpreter variable with the given name.

- `theNameStr` – the DRAW interpreter variable name to set.
- `theHandlePtr` – a pointer to the geometric variable (Handle to `Geom_Geometry` or `Geom2d_Curve` or descendant) to be set.
- `thePntPtr` – a pointer to the variable of type `gp_Pnt` to be set.
- `thePnt2dPtr` – a pointer to the variable of type `gp_Pnt2d` to be set.

All these functions are defined in `TKDraw` toolkit and return a string indicating the result of execution.
Saving and dumping shapes and geometric objects

The following functions are provided by *TKBRep* toolkit and can be used from debugger prompt:

```c
const char* BRepTools_Write (const char* theFileNameStr, void* theShapePtr)
```

Saves the specified shape to a file with the given name.

- `theFileNameStr` – the name of the file where the shape is saved.
- `theShapePtr` – a pointer to `TopoDS_Shape` variable.

```c
const char* BRepTools_Dump (void* theShapePtr)
const char* BRepTools_DumpLoc (void* theShapePtr)
```

Dumps shape or its location to cout.

- `theShapePtr` – a pointer to `TopoDS_Shape` variable.

The following function is provided by *TKMesh* toolkit:

```c
const char* BRepMesh_Dump (void* theMeshHandlePtr, const char* theFileNameStr)
```

Stores mesh produced in parametric space to BREP file.

- `theMeshHandlePtr` – a pointer to `Handle(BRepMesh_DataStructureOfDelaun)` variable.
- `theFileNameStr` – the name of the file where the mesh is stored.

The following functions are provided by *TKTopTest* toolkit:

```c
const char* MeshTest_DrawLinks(const char* theNameStr, void* theFaceAttr)
const char* MeshTest_DrawTriangles(const char* theNameStr, void* theFaceAttr)
```
Sets the edges or triangles from mesh data structure of type Handle(BRepMesh_FaceAttribute) as DRAW interpreter variables, assigning a unique name in the form "<theNameStr>_<index>" to each object.

- **theNameStr** – the prefix to use in names of objects.
- **theFaceAttr** – a pointer to Handle(BRepMesh_FaceAttribute) variable.

The following additional function is provided by *TKGeomBase* toolkit:

```cpp
const char* GeomTools_Dump (void* theHandlePtr)
```

Dump geometric object to cout.

- **theHandlePtr** – a pointer to the geometric variable (Handle to Geom_Geometry or Geom2d_Curve or descendant) to be set.
Using Visual Studio debugger

Command window

Visual Studio debugger provides the Command Window (can be activated from menu View / Other Windows / Command Window), which can be used to evaluate variables and expressions interactively in a debug session (see http://msdn.microsoft.com/en-us/library/c785s0kz.aspx). Note that the Immediate Window can also be used but it has some limitations, e.g. does not support aliases.

When the execution is interrupted by a breakpoint, you can use this window to call the above described functions in context of the currently debugged function. Note that in most cases you will need to specify explicitly context of the function by indicating the name of the DLL where it is defined.

For example, assume that you are debugging a function, where local variable TopoDS_Edge anEdge1 is of interest. The following set of commands in the Command window will save this edge to file edge1.brep, then put it to DRAW variable e1 and show it maximized in the axonometric DRAW view:

```plaintext
>?
({,,TKBRep.dll}BRepTools_Write)("d:/edge1.brep", 
(void*)&anEdge1)
0x04a2f234 "d:/edge1.brep"
>?
({,,TKDraw.dll}DBRep_Set)("e1",(void*)&anEdge1)
0x0369eba8 "e1"
>?
({,,TKDraw.dll}Draw_Eval)("donly e1; axo; fit")
0x029a48f0 ""
```

For convenience it is possible to define aliases to commands in this window, for instance (here ">" is prompt provided by the command window; in the Immediate Window this symbol should be entered manually):

```plaintext
>alias deval
> ?
```

```plaintext
({,,TKDraw.dll}Draw_Eval)
```
> alias dsetshape  ? ({{,,TKDraw}DBRep_Set})
> alias dsetcomp  ? ({{,,TKDraw}DBRep_SetComp})
> alias dsetgeom  ? ({{,,TKDraw}DrawTrSurf_Set})
> alias dsetpnt   ? ({{,,TKDraw}DrawTrSurf_SetPnt})
> alias dsetpnt2d ? ({{,,TKDraw}DrawTrSurf_SetPnt2d})
> alias saveshape ? ({{,,TKBRep}BRepTools_Write})
> alias dumpshape ? ({{,,TKBRep}BRepTools_Dump})
> alias dumploc   ? ({{,,TKBRep}BRepTools_DumpLoc})
> alias dumpmesh  ? ({{,,TKMesh}BRepMesh_Dump})
> alias dumpgeom  ? ({{,,TKGeomBase}GeomTools_Dump})

Note that aliases are stored in the Visual Studio user's preferences and it is sufficient to define them once on a workstation. With these aliases, the above example can be reproduced easier (note the space symbol after alias name!):

```
> saveshape ("d:/edge1.brep",(void*)&anEdge1)
0x04a2f234  "d:/edge1.brep"
> dsetshape ("e1",(void*)&anEdge1)
0x0369eba8  "e1"
> deval ("donly e1; axo; fit")
0x029a48f0  ""
```

Note that there is no guarantee that the call will succeed and will not affect the program execution, thus use this feature at your own risk. In particular, the commands interacting with window system (such as axo, vinit, etc.) are known to cause application crash when the program is built in 64-bit mode. To avoid this, it is recommended to prepare all necessary view windows in advance, and arrange these windows to avoid overlapping with the Visual Studio window, to ensure that they are visible during debug.
Customized display of variables content

Visual Studio provides a way to customize display of variables of different types in debugger windows (Watch, Autos, Locals, etc.).

In Visual Studio 2005-2010 the rules for this display are defined in file `autoexp.dat` located in subfolder `Common7\Packages\Debugger` of the Visual Studio installation folder (hint: the path to that folder is given in the corresponding environment variable, e.g. `VS100COMNTOOLS` for vc10). This file contains two sections: `AutoExpand` and `Visualizer`. The following rules can be added to these sections to provide more convenient display of some OCCT data types.

**[AutoExpand] section**

```plaintext
; Open CASCADE classes
Standard_Transient=<,t> count=<count,d>
Handle_Standard_Transient=<entity,x> count=<entity->count,d> <,t>
TCollection_AsciiString=<mylength,d> <mystring,s>
TCollection_HAsciiString=<myString.mylength,d> <myString.mystring,s>
TCollection_ExtendedString=<mylength,d> <mystring,su>
TCollection_HExtendedString=<myString.mylength,d> <myString.mystring,su>
TCollection_BaseSequence=size=<Size,d> curr=<CurrentIndex,d>
TCollection_BasicMap=size=<mySize,d>
NCollection_BaseSequence=size=<mySize,d> curr=<myCurrentIndex,d>
NCollection_BaseList=length=<myLength,d>
NCollection_BaseMap=size=<mySize,d> buckets=<myNbBuckets>
NCollection_BaseVector=length=<myLength,d>
TDF_Label=<myLabelNode,x> tag=<myLabelNode->myTag>
TDF_LabelNode=tag=<myTag,d>
TDocStd_Document=format=<myStorageFormat.mystring,su>
```
count=<count,d>  <,t>
TopoDS_Shape=<myTShape.entity,x>  <myOrient>
gp_XYZ=<x,g>,  <y,g>,  <z,g>
gp_Pnt=<coord.x,g>,  <coord.y,g>,  <coord.z,g>
gp_Vec=<coord.x,g>,  <coord.y,g>,  <coord.z,g>
gp.Dir=<coord.x,g>,  <coord.y,g>,  <coord.z,g>
gp_XY=<x,g>,  <y,g>
gp_Pnt2d=<coord.x,g>,  <coord.y,g>
gp.Dir2d=<coord.x,g>,  <coord.y,g>
gp.Vec2d=<coord.x,g>,  <coord.y,g>
gp_Mat2d={<matrix[0][0],g>,<matrix[0][1],g>},
   {<matrix[1][0],g>,<matrix[1][1],g>}
gp.Ax1=loc={<loc.coord.x,g>,  <loc.coord.y,g>,
   <loc.coord.z,g>}  vdir={<vdir.coord.x,g>,
   <vdir.coord.y,g>,  <vdir.coord.z,g>}

[Visualizer] section

; Open CASCADE classes
NCollection_Handle<*> {  
   preview ( *(($T0::Ptr*)$e.entity)->myPtr) )  
   children ( (($T0::Ptr*)$e.entity)->myPtr )
}

NCollection_List<*> {  
   preview ( #( "NCollection_List [", $e.myLength, "]" ) )  
   children ( #list( head: $c.myFirst, next: myNext )  
        : #(*($T1*)($e+1)) )
}

NCollection_Array1<*> {  
   preview ( #( "NCollection_Array1 [",  
           $e.myLowerBound, ",", $e.myUpperBound, "]") )  
   children ( #array( expr: $c.myData[$i], size:  
       1+$c.myUpperBound ) )
math_Vector {
    preview ( #( "math_Vector [", $e.LowerIndex, ", "$e.UpperIndex, "]" ) )
    children ( #array ( expr: ((double*) ($c.Array.Addr))[i], size: 1+$$c.UpperIndex ) )
}

TColStd_Array1OfReal {
    preview ( #( "Array1OfReal [", $e.myLowerBound, ", "$e.myUpperBound, "]" ) )
    children ( #array ( expr: ((double*)($c.myStart))[i], size: 1+$$c.myUpperBound ) )
}

Handle_TColStd_HArray1OfReal {
    preview ( #( "HArray1ofReal [", 
        ((TColStd_HArray1OfReal*)$e.entity)->myArray.myLowerBound, ", " 
        ((TColStd_HArray1OfReal*)$e.entity)->myArray.myUpperBound, "]" ) )
    children ( #array ( expr: ((double*)((TColStd_HArray1OfReal*)$e.entity)->myArray.myStart))[$i], size: 1+$$c.myUpperBound ) )
}

TColStd_Array1OfInteger {
    preview ( #( "Array1ofInteger [", $e.myLowerBound, ", "$e.myUpperBound, "]" ) )
    children ( #array ( expr: ((int*)($c.myStart))[i], size: 1+$$c.myUpperBound ) )
}
Handle_TColStd_HArray1OfInteger {
    preview ( #( "HArray1OfInteger [",
                ((TColStd_HArray1OfInteger*)$e.entity)->myArray.myLowerBound, "..",
                ((TColStd_HArray1OfInteger*)$e.entity)->myArray.myUpperBound, "]",
                [$e.entity,x], " count="+$e.entity->count )
    children ( #array ( expr: ((int*)
                        (((TColStd_HArray1OfInteger*)$e.entity)->myArray.myStart))[$i],
                        size: 1+
                        ((TColStd_HArray1OfInteger*)$e.entity)->myArray.myUpperBound ) )
}

Handle_TCollection_HExtendedString {
    preview ( #( "HExtendedString ", [$e.entity,x], " count="+$e.entity->count,
                 " ",
                 (TCollection_HExtendedString*$e.entity)->myString )
    children ( #([actual members]: [$e,!] ) )
}

Handle_TCollection_HAsciiString {
    preview ( #( "HAsciiString ", [$e.entity,x], " count="+$e.entity->count,
                 " ",
                 (TCollection_HAsciiString*$e.entity)->myString )
    children ( #([actual members]: [$e,!] ,
                 #array( expr:
In Visual Studio 2012 and later, visualizers can be put in a separate file in subdirectory *Visualizers*. See file *occt.natvis* for example.
Performance measurement tools

It is recommended to use specialized performance analysis tools to profile OCCT and application code. However, when such tools are not available or cannot be used for some reason, tools provided by OSD package can be used: low-level C functions and macros defined in `OSD_PerfMeter.h` and `OSD_PerfMeter` class.

This tool maintains an array of 100 global performance counters that can be started and stopped independently. Adding a performance counter to a function of interest allows to get statistics on the number of calls and the total execution time of the function.

- In C++ code, this can be achieved by creating local variable `OSD_PerfMeter` in each block of code to be measured.
- In C or Fortran code, use functions `perf_start_meter` and `perf_stop_meter` to start and stop the counter.

Note that this instrumentation is intended to be removed when the profiling is completed.

Macros provided in `OSD_PerfMeter.h` can be used to keep instrumentation code permanently but enable it only when macro `PERF_ENABLE_METERS` is defined. Each counter has its name shown when the collected statistics are printed.

In DRAW, use command `dperf` to print all performance statistics.

Note that performance counters are not thread-safe.
# Upgrade from older OCCT versions

## Table of Contents

- Introduction
  - Precautions
  - Disclaimer
- Upgrade to OCCT 6.5.0
- Upgrade to OCCT 6.5.1
- Upgrade to OCCT 6.5.2
- Upgrade to OCCT 6.5.3
- Upgrade to OCCT 6.5.4
- Upgrade to OCCT 6.6.0
- Upgrade to OCCT 6.7.0
  - Object-level clipping and capping algorithm.
  - Redesign of markers presentation
  - Default views are not created automatically
  - Improved dimensions implementation
  - NCollection_Set replaced by List collection
- Upgrade to OCCT 6.8.0
  - Changes in NCollection classes
  - 3D View Camera
  - Redesign of Connected Interactive Objects
- Support of UNICODE Characters
- Elimination of Projection Shift Concept
- Upgrade to OCCT 6.9.0
  - 3D Viewer initialization
  - Changes in Selection
  - Changes in Adaptor3d_Curve class
  - Changes in V3d_View class
- Upgrade to OCCT 7.0.0
  - Removal of legacy persistence
  - Removal of CDL and WOK
    - Automatic upgrade
    - Possible compiler errors
    - Possible runtime problems
    - Option to avoid cast of handle to reference to base type
    - Preserving compatibility with OCCT 6.x
    - Applications based on CDL and WOK
- Separation of BSpline cache
- Structural result of Boolean operations
- BRepExtrema_ExtCC finds one solution only
- Removal of SortTools package
- On-screen objects and ColorScale
- UserDraw and Visual3d
- Deprecation of Local Context
- Separation of visualization part from TKCAF
- Correction of interpretation of Euler angles in gp_Quaternion
- Zoom Persistent Selection
- Texture mapping of objects
- Shape presentation builders
- Upgrade to OCCT 7.1.0
  - Presentation attributes
  - Typedefs
  - Programmable Pipeline
  - Transformation persistence
  - Dynamic highlight and selection properties
  - Correction in TObj_Model class
- Redundant environment variables
- Removed features
- Other changes
- Upgrade to OCCT 7.2.0
- Removed features
 Corrections in BRepOffset API

 Corrections in BRepOffset API

 Highlight style

 Elimination of implicit 3D Viewer updates

 Elimination of Quantity_NameOfColor from TKV3d interface classes

 Result of Boolean operations on containers

 Other changes

 BOP - Pairs of interfering indices

 Removal of the Draw commands based on old Boolean operations

 Change of Face/Face intersection in Boolean operations

 Restore OCCT 6.9.1 persistence

 Change in BRepLib_MakeFace algorithm

 Change in BRepFill_OffsetWire algorithm

 Change in Geom(2d)Adaptor_Curve::IsPeriodic

 Change in algorithm ShapeUpgrade_UnifySame

 Changes in STL Reader / Writer

 Refactoring of the Error/Warning reporting system in Boolean Component
Introduction

This document provides technical details on changes made in particular versions of OCCT. It can help to upgrade user applications based on previous versions of OCCT to newer ones.
Precautions

Back-up your code before the upgrade. We strongly recommend using version control system during the upgrade process and saving one or several commits at each step of upgrade, until the overall result is verified. This will facilitate identification and correction of possible problems that can occur at the intermediate steps of upgrade. It is advisable to document each step carefully to be able to repeat it if necessary.
Disclaimer

This document describes known issues that have been encountered during porting of OCCT and some applications and approaches that have helped to resolve these issues in known cases. It does not pretend to cover all possible migration issues that can appear in your application. Take this document with discretion; apply your expertise and knowledge of your application to ensure the correct result.

The automatic upgrade tool is provided as is, without warranty of any kind, and we explicitly disclaim any liability for possible errors that may appear due to use of this tool. It is your responsibility to ensure that the changes you made in your code are correct. When you upgrade the code by an automatic script, make sure to carefully review the introduced changes at each step before committing them.
Upgrade to OCCT 6.5.0

Porting of user applications from an earlier OCCT version to version 6.5 requires taking into account the following major changes:

- If you are not comfortable with dependence on Intel TBB, FreeImage, or Gl2Ps libraries, you will need to (re)build OCCT with these dependencies disabled.
- The low-level format version of OCAF binary and XML persistence has been incremented. Hence, the files saved by OCCT 6.5 to OCAF binary or XML format will not be readable by previous versions of OCCT.
- The BRepMesh triangulation algorithm has been seriously revised and now tries hard to fulfill the requested deflection and angular tolerance parameters. If you experience any problems with performance or triangulation quality (in particular, display of shapes in shading mode), consider revising the values of these parameters used in your application.
- If you were using method ToPixMap() of class V3d_View to get a buffer for passing to Windows API functions (e.g. BitBlt), this will not work anymore. You will need to use method Image_PixMap::AccessBuffer() to get the raw buffer data that can be further passed to WinAPI functions.
- As the processing of message gravity parameter in Message package has been improved, some application messages (especially the ones generated by IGES or STEP translators) can be suppressed or new messages appear in the application. Use relevant message level parameter to tune this behavior.
Upgrade to OCCT 6.5.1

Porting of user applications from an earlier OCCT version to version 6.5.1 requires taking into account the following major changes:

- Method `Graphic3d_Structure::Groups()` now returns `Graphic3d_SequenceOfGroup`. If this method has been used, the application code should be updated to iterate another collection type or, if `Graphic3d_HSetOfGroup` is required, to fill its own collection:

```cpp
const Graphic3d_SequenceOfGroup& aGroupsSeq = theStructure.Groups();
Handle(Graphic3d_HSetOfGroup) aGroupSet = new Graphic3d_HSetOfGroup();
Standard_Integer aLen = aGroupsSeq.Length();
for (Standard_Integer aGr = 1; aGr <= aLen; ++aGr)
{
    aGroupSet->Add (aGroupsSeq.Value (aGr));
}
```

- All occurrences of `Select3D_Projector` in the application code (if any) should be replaced with `Handle(Select3D_Projector)`.
- The code of inheritors of `Select3D_SensitiveEntity` should be updated if they override `Matches()` (this is probable, if clipping planes are used).
- Constructor for `V3d_Plane` has been changed, so the extra argument should be removed if used in the application. It is necessary to add a new plane using method `V3d_Viewer::AddPlane()` if `V3d_Viewer` has been used to manage clipping planes list (this does not affect clipping planes representation). Please, have a look at the source code for new DRAWEXE `vclipplane` command in `ViewerTest_ObjectsCommands.hxx`, `VClipPlane` to see how clipping planes can be managed in the application.
Upgrade to OCCT 6.5.2

Porting of user applications from an earlier OCCT version to version 6.5.2 requires taking into account the following major changes:

- Any code that has been generated by WOK from CDL generic classes `Tcollection_DataMap` and `Tcollection_IndexedDataMap` needs to be regenerated by WOK to take into account the change in the interface of these classes.
- The enumerations `CDF_StoreStatus` and `CDF_RetrievableStatus` have been replaced by the enumerations `PCDM_StoreStatus` and `PCDM_ReaderStatus`. Correspondingly, the methods `Open`, `Save` and `SaveAs` of the class `TDocStd_Application` have changed their return value. Any code, which uses these enumerations, needs to be updated.
- `BRepLib_MakeFace` has been modified to receive tolerance value for resolution of degenerated edges. This tolerance parameter has no default value to ensure that the client code takes care of passing a meaningful value, not just `Precision::Confusion`, so some porting overheads are expected.
- If the callback mechanism in `call_togl_redraw` function was used in the application code, it is necessary to revise it to take into account the new callback execution and provide a check of reason value of `Aspect_GraphicCallbackStruct` in callback methods to confirm that the callback code is executed at the right moment. Now the callbacks are executed before redrawing the underlayer, before redrawing the overlayer and at the end of redrawing. The information about the moment when the callback is invoked is provided with the reason value in form of an additional bit flag (`OCC_PRE_REDRAW`, `OCC_PRE_OVERLAY`). The state of OpenGL changed in callback methods will not be restored automatically, which might lead to unwanted behavior in redrawing procedure.
- The print method used in the application code might need to be revised to take into account the ability to choose between print algorithms: tile and stretch. The stretch algorithm will be selected by default during porting.
- It is recommended to `BRepMesh_DiscretFactory` users, to check `BRepMesh_DiscretFactory::SetDefault()` return value to determine
plugin availability / validity. \textit{BRepMesh\_DiscretFactory\::\:Discret()} method now returns handle instead of pointer. The code should be updated in the following manner:

\begin{verbatim}
Handle(BRepMesh\_DiscretRoot) aMeshAlgo =
   BRepMesh\_DiscretFactory\::\:Get().Discret
   (theShape, theDeflection, theAngularToler);
if (!aMeshAlgo.IsNull()) {} ...
\end{verbatim}

- The default state of \textit{BRepMesh} parallelization has been turned off. The user should switch this flag explicitly:
  - by using methods \textit{BRepMesh\_IncrementalMesh\::\:SetParallel(Standard\_True)} for each \textit{BRepMesh\_IncrementalMesh} instance before \textit{Perform()};
  - by calling \textit{BRepMesh\_IncrementalMesh\::\:SetParallelDefault(Standard\_True)} when \textit{BRepMesh\_DiscretFactory} is used to retrieve the meshing tool (this also affects auto-triangulation in \textit{AIS}).
Upgrade to OCCT 6.5.3

Porting of user applications from an earlier OCCT version to version 6.5.3 requires taking into account the following major changes:

- As a result of code clean-up and redesign of TKOpenGl driver, some obsolete functions and rendering primitives (TriangleMesh, TriangleSet, Bezier, Polyline, Polygon, PolygonHoles, QuadrangleMesh and QuadrangleSet) have been removed. Instead, the application developers should use primitive arrays that provide the same functionality but are hardware-accelerated. The details can be found in OCCT Visualization User's Guide, “Primitive Arrays” chapter.
- Applications should not call AIS_InteractiveObject::SetPolygonOffsets() method for an instance of AIS_TexturedShape class after it has been added to AIS_InteractiveContext. More generally, modification of Graphic3d_AspectFillArea3d parameters for the computed groups of any AIS_InteractiveObject subclass that uses texture mapping should be avoided, because this results in broken texture mapping (see issue 23118). It is still possible to apply non-default polygon offsets to AIS_TexturedShape by calling SetPolygonOffsets() before displaying the shape.
- The applications that might have used internal functions provided by TKOpenGl or removed primitives will need to be updated.
- In connection with the implementation of Z-layers it might be necessary to revise the application code or revise the custom direct descendant classes of Graphic3d_GraphicDriver and Graphic3d_StructureManager to use the Z-layer feature.
- Global variables Standard_PI and PI have been eliminated (use macro M_PI instead).
- Method HashCode() has been removed from class Standard_Transient. It is advisable to use global function HashCode() for Handle objects instead.
- Declaration of operators new/delete for classes has become consistent and is encapsulated in macros.
- Memory management has been changed to use standard heap (MMGT_OPT=0) and reentrant mode (MMGT_REENTRANT=1) by
default.
- Map classes in NCollection package now receive one more argument defining a hash tool.
Upgrade to OCCT 6.5.4

Porting of user applications from an earlier OCCT version to version 6.5.4 requires taking into account the following major changes:

- The code using obsolete classes `Aspect_PixMap`, `Xw_PixMap` and `WNT_PixMap` should be rewritten implementing class `Image_PixMap`, which is now retrieved by `ToPixMap` methods as argument. A sample code using `ToPixMap` is given below:

```cpp
#include <Image_AlienPixMap.hxx>
void dump (Handle(V3d_View)& theView3D)
{
    Standard_Integer aWndSizeX = 0;
    Standard_Integer aWndSizeY = 0;
    theView3D->Window()->Size (aWndSizeX, aWndSizeY);
    Image_AlienPixMap aPixMap;
    theView3D->ToPixMap (aPixMap, aWndSizeX, aWndSizeY);
    aPixMap.Save ("c:\image.png");
}
```

- Now OpenGL resources related to Interactive Objects are automatically freed when the last view (window) is removed from graphical driver. To avoid presentation data loss, the application should replace an old view with a new one in the proper order: first the new view is created and activated and only then the old one is detached and removed.

- It is recommended to use `NCollection` containers with hasher parameter (introduced in 6.5.3) instead of global definition `IsEqual()/HashCode()` as well as to use explicit namespaces to avoid name collision.
Upgrade to OCCT 6.6.0

Porting of user applications from an earlier OCCT version to version 6.6.0 requires taking into account the following major changes:

- Due to the changes in the implementation of Boolean Operations, the order of sub-shapes resulting from the same operation performed with OCCT 6.5.x and OCCT 6.6.0 can be different. It is necessary to introduce the corresponding changes in the applications for which the order of sub-shapes resulting from a Boolean operation is important. It is strongly recommended to use identification methods not relying on the order of sub-shapes (e.g. OCAF naming).
- If you need to use OCCT on Mac OS X with X11 (without Cocoa), build OCCT with defined pre-processor macro `CSF_MAC_USE_GLX11`. XLib front-end (previously the only way for unofficial OCCT builds on Mac OS X) is now disabled by default on this platform. If your application has no support for Cocoa framework you may build OCCT with XLib front-end adding `MACOSX_USE_GLX` macro to compiler options (you may check the appropriate option in WOK configuration GUI and in CMake configuration). Notice that XQuartz (XLib implementation for Mac OS X) now is an optional component and does not provide a sufficient level of integrity with native (Cocoa-based) applications in the system. It is not possible to build OCCT with both XLib and Cocoa at the same time due to symbols conflict in OpenGL functions.
- Animation mode and degeneration presentation mode (simplified presentation for animation) and associated methods have been removed from 3D viewer functionality. Correspondingly, the code using methods `SetAnimationModeOn()`, `SetAnimationModeOff()`, `AnimationModeIsOn()`, `AnimationMode()`, `Tumble()`, `SetDegenerateModeOn()`, `SetDegenerateModeOff()` and `DegenerateModeIsOn()` of classes `V3d_View` and `Visual3d_View` will need to be removed or redesigned. Please, notice that Hidden Line Removal presentation was not affected; however, the old code that used methods `V3d_View::SetDegenerateModeOn` or `V3d_View::SetDegenerateModeOff` to control HLR presentation should be updated to use `V3d_View::SetComputedMode` method
instead.

- Calls of `Graphic3d_Group::BeginPrimitives()` and `Graphic3d_Group::EndPrimitives()` should be removed from the application code.
- Application functionality for drawing 2D graphics that was formerly based on `TKV2d` API should be migrated to `TKV3d` API. The following changes are recommended for this migration:
  - A 2D view can be implemented as a `V3d_View` instance belonging to `V3d_Viewer` managed by `AIS_InteractiveContext` instance. To turn `V3d_View` into a 2D view, the necessary view orientation should be set up at the view initialization stage using `V3d_View::SetProj()` method, and view rotation methods simply should not be called.
  - Any 2D graphic entity (formerly represented with `AIS2D_InteractiveObject`) should become a class derived from `AIS_InteractiveObject` base. These entities should be manipulated in a view using `AIS_InteractiveContext` class API.
  - All drawing code should be put into `Compute()` virtual method of a custom interactive object class and use API of `Graphic3d` package. In particular, all geometry should be drawn using class hierarchy derived from `Graphic3d_ArrayOfPrimitives`. Normally, the Z coordinate for 2D geometry should be constant, unless the application implements some advanced 2D drawing techniques like e.g. multiple "Z layers" of drawings.
  - Interactive selection of 2D presentations should be set up inside `ComputeSelection()` virtual method of a custom interactive object class, using standard sensitive entities from `Select3D` package and standard or custom entity owners derived from `SelectMgr_EntityOwner` base. Please refer to the Visualization User's Guide for further details concerning OCCT 3D visualization and selection classes. See also `Viewer2D` OCCT sample application, which shows how 2D drawing can be implemented using TKV3d API.
- Run-time graphic driver library loading mechanism based on `CSF_GraphicShr` environment variable usage has been replaced by explicit linking against `TKOpenGl` library. The code sample below shows how the graphic driver should be created and initialized in the application code:

```c++
// initialize a new viewer with OpenGl graphic driver
```
Handle(Graphic3d_GraphicDriver) aGraphicDriver =
new OpenGL_GraphicDriver ("TKOpenGL);
aGraphicDriver->Begin (new
   Aspect_DisplayConnection());
TCollection_ExtendedString aNameOfViewer
   ("Visu3D");
Handle(V3d_Viewer) aViewer
   = new V3d_Viewer (aGraphicDriver,
   aNameOfViewer.ToExtString());
aViewer->Init();

   // create a new window or a wrapper over the
   // existing window,
   // provided by a 3rd-party framework (Qt, MFC, C#
   // or Cocoa)
#if defined(_WIN32) || defined(__WIN32__)
   Aspect_Handle aWindowHandle = (Aspect_Handle
   )winId();
   Handle(WNT_Window) aWindow = new WNT_Window
   (winId());
#elif defined(__APPLE__) &&
   !defined(MACOSX_USE_GLX)
   NSView* aViewHandle = (NSView* )winId();
   Handle(Cocoa_Window) aWindow = new Cocoa_Window
   (aViewHandle);
#else
   Aspect_Handle aWindowHandle = (Aspect_Handle
   )winId();
   Handle(Xw_Window) aWindow =
   new Xw_Window (aGraphicDriver-
   >GetDisplayConnection(), aWindowHandle);
#endif // WNT

   // setup the window for a new view
   Handle(V3d_View) aView = aViewer->CreateView();
aView->SetWindow (aWindow);

• The following changes should be made in the application-specific
implementations of texture aspect:

- Graphic3d_TextureRoot inheritors now should return texture image by overloading of Graphic3d_TextureRoot::GetImage() method instead of the old logic.
- Now you can decide if the application should store the image copy as a field of property or reload it dynamically each time (to optimize the memory usage). The default implementation (which loads the image content from the provided file path) does not hold an extra copy since it will be uploaded to the graphic memory when first used.
- Notice that the image itself should be created within Image_PixMap class from AlienImage package, while Image_Image class is no more supported and will be removed in the next OCCT release.
Upgrade to OCCT 6.7.0

Porting of user applications from an earlier OCCT version to version 6.7.0 requires taking into account the following major changes.
Object-level clipping and capping algorithm.

- It might be necessary to revise and port code related to management of view-level clipping to use Graphic3d_ClipPlane instead of V3d_Plane instances. Please note that V3d_Plane class has been preserved – as previously, it can be used as plane representation. Another approach to represent Graphic3d_ClipPlane in a view is to use custom presentable object.

- The list of arguments of Select3D_SensitiveEntity::Matches() method for picking detection has changed. Since now, for correct selection clipping, the implementations should perform a depth clipping check and return (as output argument) minimum depth value found at the detected part of sensitive. Please refer to CDL / Doxygen documentation to find descriptive hints and snippets.

- Select3D_SensitiveEntity::ComputeDepth() abstract method has been removed. Custom implementations should provide depth checks by method Matches() instead – all data required for it is available within a scope of single method.

- It might be necessary to revise the code of custom sensitive entities and port Matches() and ComputeDepth() methods to ensure proper selection clipping. Please note that obsolete signature of Matches is not used anymore by the selector. If your class inheriting Select3D_SensitiveEntity redefines the method with old signature the code should not compile as the return type has been changed. This is done to prevent override of removed methods.
Redesign of markers presentation

- Due to the redesign of Graphic3d_AspectMarker3d class the code of custom markers initialization should be updated. Notice that you can reuse old markers definition code as TColStd_HArray1OfByte; however, Image_PixMap is now the preferred way (and supports full-color images on modern hardware).
- Logics and arguments of methods AIS_InteractiveContext::Erase() and AIS_InteractiveContext::EraseAll() have been changed. Now these methods do not remove resources from Graphic3d_Structure; they simply change the visibility flag in it. Therefore, the code that deletes and recomputes resources should be revised.
- Graphic3d_Group::MarkerSet() has been removed. Graphic3d_Group::AddPrimitiveArray() should be used instead to specify marker(s) array.
Default views are not created automatically

As the obsolete methods Init(), DefaultOrthographicView() and DefaultPerspectiveView() have been removed from V3d_Viewer class, the two default views are no longer created automatically. It is obligatory to create V3d_View instances explicitly, either directly by operator new or by calling V3d_Viewer::CreateView().

The call V3d_Viewer::SetDefaultLights() should also be done explicitly at the application level, if the application prefers to use the default light source configuration. Otherwise, the application itself should set up the light sources to obtain a correct 3D scene.
**Improved dimensions implementation**

- It might be necessary to revise and port code related to management of `AIS_LengthDimension`, `AIS_AngleDimension` and `AIS_DiameterDimension` presentations. There is no more need to compute value of dimension and pass it as string to constructor argument. The value is computed internally. The custom value can be set with `SetCustomValue()` method.
- The definition of units and general aspect properties is now provided by `Prs3d_DimensionUnits` and `Prs3d_DimensionAspect` classes.
- It might be also necessary to revise code of your application related to usage of `AIS_DimensionDisplayMode` enumeration. If it used for specifying the selection mode, then it should be replaced by a more appropriate enumeration `AIS_DimensionSelectionMode`. 
NCollection_Set replaced by List collection

It might be necessary to revise your application code, which uses non-ordered Graphic3d_SetOfHClipPlane collection type and replace its occurrences by ordered Graphic3d_SequenceOfHClipPlane collection type.
Upgrade to OCCT 6.8.0

Porting of user applications from an earlier OCCT version to version 6.8.0 requires taking into account the following major changes.
Changes in NCollection classes

Method Assign() in NCollection classes does not allow any more copying between different collection types. Such copying should be done manually.

List and map classes in NCollection package now require that their items be copy-constructible, but do not require items to have default constructor. Thus the code using NCollection classes for non-copy-constructible objects needs be updated. One option is to provide copy constructor; another possibility is to use Handle or other smart pointer.
3D View Camera

If ViewMapping and ViewOrientation were used directly, this functionality has to be ported to the new camera model. The following methods should be considered as an alternative to the obsolete Visual3d services (all points and directions are supposed to be in world coordinates):

- `Graphic3d_Camera::ViewDimensions()` or `V3d_View::Size()/ZSize()` – returns view width, height and depth (or "Z size"). Since the view is symmetric now, you can easily compute top, bottom, left and right limits. `Graphic3d_Camera::ZNear()/ZFar()` can be used to obtain the near and far clipping distances with respect to the eye.
- `Graphic3d_Camera::Up()` or `V3d_View::Up()` – returns Y direction of the view.
- `Graphic3d_Camera::Direction()` returns the reverse view normal directed from the eye, `V3d_View::Proj()` returns the old-style view normal.
- `Graphic3d_Camera::Eye()` or `V3d_View::Eye()` – returns the camera position (same as projection reference point in old implementation).
- `Graphic3d_Camera::Center()` or `V3d_View::At()` – returns the point the camera looks at (or view reference point according to old terminology).

The current perspective model is not fully backward compatible, so the old perspective-related functionality needs to be reviewed.

Please revise application-specific custom presentations to provide proper bounding box. Otherwise object might become erroneously clipped by automatic ZFit or frustum culling algorithms enabled by default.
Redesign of Connected Interactive Objects

The new implementation of connected Interactive Objects makes it necessary to take the following steps if you use connected Interactive Objects in your application.

- Use new `PrsMgr_PresentableObject` transformation API.
- Call `RemoveChild()` from the original object after connect if you need the original object and `AIS_ConnectedInteractive` to move independently.
- Access instances of objects connected to `AIS_MultiplyConnectedInteractive` with `Children()` method.
- For `PrsMgr_PresentableObject` transformation:
  - `SetLocation (TopLoc_Location) -> SetLocalTransformation (gp_Trsf)`
  - `Location -> LocalTransformation`
  - `HasLocation -> HasTransformation`
  - `ResetLocation -> ResetTransformation`
Support of UNICODE Characters

Support of UNICODE characters introduced in OCCT breaks backward compatibility with applications, which currently use filenames in extended ASCII encoding bound to the current locale. Such applications should be updated to convert such strings to UTF-8 format.

The conversion from UTF-8 to wchar_t is made using little-endian approach. Thus, this code will not work correctly on big-endian platforms. It is needed to complete this in the way similar as it is done for binary persistence (see the macro DO_INVERSE in FSD_FileHeader.hxx).
Elimination of Projection Shift Concept

It might be necessary to revise the application code, which deals with `Center()` method of `V3d_View`.

This method was used to pan a `V3d` view by virtually moving the screen center with respect to the projection ray passed through Eye and At points. There is no more need to derive the panning from the Center parameter to get a camera-like eye position and look at the coordinates. `Eye()` and `At()` now return these coordinates directly. When porting code dealing with `Center()`, the parameters `Eye()` and `At()` can be adjusted instead. Also `V3d_View::SetCenter(Xpix, Ypix)` method can be used instead of `V3d_View::Center(X, Y)` to center the view at the given point. However, if the center coordinates X and Y come from older OCCT releases, calling `V3d_View::Panning(-X, -Y)` can be recommended to compensate missing projection shift effect.

There are several changes introduced to `Graphic3d_Camera`. The internal data structure of the camera is based on `Standard_Real` data types to avoid redundant application-level conversions and precision errors. The transformation matrices now can be evaluated both for `Standard_Real` and `Standard_ShortReal` value types. `ZNear` and `ZFar` planes can be either negative or positive for orthographic camera projection, providing a trade-off between the camera distance and the range of `ZNear` or `ZFar` to reduce difference of exponents of values composing the orientation matrix - to avoid calculation errors. The negative values can be specified to avoid Z-clipping if the reference system of camera goes inside of the model when decreasing camera distance.

The auto z fit mode, since now, has a parameter defining Z-range margin (the one which is usually passed as argument to `ZFitAll()` method). The methods `SetAutoZFitMode()`, `AutoZFitScaleFactor()` and `ZFitAll()` from class `V3d_View` deal with the new parameter.

The class `Select3D_Projector` now supports both orientation and projection transformation matrices, which can be naturally set for the projector. The definition of projector was revised in `StdSelect_VisualSelector3d`: perspective and orthographic projection
parameters are handled properly. Orthographic projector is based only on direction of projection - no more Center property. This makes it possible to avoid unnecessary re-projection of sensitive while panning, zooming or moving along the projection ray of the view. These operations do not affect the orthographic projection.
Upgrade to OCCT 6.9.0

Porting of user applications from an earlier OCCT version to version 6.9.0 requires taking into account the following major changes.
3D Viewer initialization

3D Viewer now uses GLSL programs for managing frame buffer and stereoscopic output. For proper initialization, application should configure `CSF_ShadersDirectory` environment variable pointing to a folder with GLSL resources - files from folder `CASROOT/src/Shaders`. Note that `CSF_ShadersDirectory` become optional since OCCT 7.1.0 release.
Changes in Selection

Selection mechanism of 3D Viewer has been redesigned to use 3-level BVH tree traverse directly in 3D space instead of projection onto 2D screen space (updated on each rotation). This architectural redesign may require appropriate changes at application level in case if custom Interactive Objects are used.

Standard selection

Usage of standard OCCT selection entities would require only minor updates.

Custom Interactive Objects should implement new virtual method SelectMgr_SelectableObject::BoundingBox().

Now the method SelectMgr_Selection::Sensitive() does not return SelectBasics_SensitiveEntity. It returns an instance of SelectMgr_SensitiveEntity, which belongs to a different class hierarchy (thus DownCast() will fail). To access base sensitive it is necessary to use method SelectMgr_SensitiveEntity::BaseSensitive(). For example:

```cpp
Handle(SelectMgr_Selection) aSelection =
    anInteractiveObject->Selection (aMode);
for (aSelection->Init(); aSelection->More();
    aSelection->Next())
{
    Handle(SelectBasics_SensitiveEntity) anEntity =
        aSelection->Sensitive()->BaseSensitive();
}
```

Custom sensitive entities

Custom sensitive entities require more complex changes, since the selection algorithm has been redesigned and requires different output from the entities.
The method `SelectBasics_SensitiveEntity::Matches()` of the base class should be overridden following the new signature:

```
Standard_Boolean Matches (SelectBasics_SelectingVolumeManager& theMgr, SelectBasics_PickResult& thePickResult)
```

where `theMgr` contains information about the currently selected frustum or set of frustums (see `SelectMgr_RectangularFrustum`, `SelectMgr_TriangularFrustum`, `SelectMgr_TriangularFrustumSet`) and `SelectBasics_PickResult` is an output parameter, containing information about the depth of the detected entity and distance to its center of geometry.

In the overridden method it is necessary to implement an algorithm of overlap and inclusion detection (the active mode is returned by `theMgr.IsOverlapAllowed()`) with triangular and rectangular frustums.

The depth and distance to the center of geometry must be calculated for the 3D projection of user-picked screen point in the world space. You may use already implemented overlap and inclusion detection methods for different primitives from `SelectMgr_RectangularFrustum` and `SelectMgr_TriangularFrustum`, including triangle, point, axis-aligned box, line segment and planar polygon.

Here is an example of overlap/inclusion test for a box:

```cpp
if (!theMgr.IsOverlapAllowed()) // check for inclusion
{
  Standard_Boolean isInside = Standard_True;
  return theMgr.Overlaps (myBox.CornerMin(),
                         myBox.CornerMax(), &isInside) && isInside;
}

Standard_Real aDepth;
if (!theMgr.Overlaps (myBox, aDepth)) // check for overlap
{
  return Standard_False;
}
```
thePickResult =
SelectBasics_PickResult (aDepth,
    theMgr.DistToGeometryCenter (myCenter3d));

The interface of *SelectBasics_SensitiveEntity* now contains four new pure virtual functions that should be implemented by each custom sensitive:

- *BoundingBox()* – returns a bounding box of the entity;
- *Clear()* – clears up all the resources and memory allocated for complex sensitive entities;
- *BVH()* – builds a BVH tree for complex sensitive entities, if it is needed;
- *NbSubElements()* – returns atomic sub-entities of a complex sensitive entity, which will be used as primitives for BVH building. If the entity is simple and no BVH is required, this method returns 1.

Each sensitive entity now has its own tolerance, which can be overridden by method *SelectBasics_SensitiveEntity::SetSensitivityFactor()* called from constructor.
Changes in Adaptor3d_Curve class

All classes inheriting *Adaptor3d_Curve* (directly or indirectly) must be updated in application code to use new signature of methods *Intervals()* and *NbIntervals()*. Note that no compiler warning will be generated if this is not done.
Changes in V3d_View class

The methods V3d_View::Convert and V3d_View::ConvertWithProj() have ceased to return point on the active grid. It might be necessary to revise the code of your application so that V3d_View::ConvertToGrid() was called explicitly for the values returned by V3d_View::Convert to get analogous coordinates on the grid. The methods V3d_View::Convert and V3d_View::ConvertWithProj convert point into reference plane of the view corresponding to the intersection with the projection plane of the eye/view point vector.
Upgrade to OCCT 7.0.0

Porting of user applications from an earlier OCCT version to version 7.0.0 requires taking into account the following major changes.

Building OCCT now requires compiler supporting some C++11 features. The supported compilers are:

- MSVC: version 10 (Visual Studio 2010) or later
- GCC: version 4.3 or later
- CLang: version 3.6 or later
- ICC: version XE 2013 SP 1 or later

When compiling code that uses OCCT with GCC and CLang compilers, it is necessary to use compiler option -std=c++0x (or its siblings) to enable C++11 features.
Removal of legacy persistence

Legacy persistence for shapes and OCAF data based on Storage_Schema (toolkits TKPShape, TKPLCAF, TKPCAF, TKShapeSchema, TLStdLSchema, TKStdSchema, and TKXCAFSchema) has been removed in OCCT 7.0.0. The applications that used these data persistence tools need to be updated to use other persistence mechanisms.

Note
For compatibility with previous versions, the possibility to read standard OCAF data (TKLCAF and TKCAF) from files stored in the old format is preserved (toolkits TKStdL and TKStd).

The existing data files in standard formats can be converted using OCCT 6.9.1 or a previous version, as follows.

Note
Reading / writing custom files capability from OCCT 6.9.1 is restored in OCCT 7.2.0. See details in Restore OCCT 6.9.1 persistence section.

CSFDB files

Files in CSFDB format (usually with extension .csfdb) contain OCCT shape data that can be converted to BRep format. The easiest way to do that is to use ImportExport sample provided with OCCT 6.9.0 (or earlier):

- Start ImportExport sample;
- Select File / New;
- Select File / Import / CSFDB... and specify the file to be converted;
- Drag the mouse with the right button pressed across the view to select all shapes by the rectangle;
- Select File / Export / BREP... and specify the location and name for the resulting file

OCAF and XCAF documents
Files containing OCAF data saved in the old format usually have extensions .std, .sgd or .dxc (XDE documents). These files can be converted to XML or binary OCAF formats using DRAW Test Harness commands. Note that if the file contains only attributes defined in TKLCAF and TKCAF, this action can be performed in OCCT 7.0; otherwise OCCT 6.9.1 or earlier should be used.

For that, start **DRAWEXE** and perform the following commands:

- To convert *.std and *.sgd file formats to binary format *.cbf (The created document should be in BinOcaf format instead of MDTV-Standard):

  ```
  Draw[]> pload ALL
  Draw[]> Open [path to *.std or *.sgd file] Doc
  Draw[]> Format Doc BinOcaf
  Draw[]> SaveAs Doc [path to the new file]
  ```

- To convert *.dxc file format to binary format *.xbf (The created document should be in BinXCAF format instead of MDTV-XCAF):

  ```
  Draw[]> pload ALL
  Draw[]> XOpen [path to *.dxc file] Doc
  Draw[]> Format Doc BinXCAF
  Draw[]> XSave Doc [path to the new file]
  ```

On Windows, it is necessary to replace back slashes in the file path by direct slashes or pairs of back slashes.

Use **XmlOcaf** or **XmlXCAF** instead of **BinOcaf** and **BinXCAF**, respectively, to save in XML format instead of binary one.
Removal of CDL and WOK

OCCT code has been completely refactored in version 7.0 to get rid of obsolete technologies used since its inception: CDL (Cas.Cade Definition Language) and WOK (Workshop Organization Kit).

C++ code previously generated by WOK from CDL declarations is now included directly in OCCT sources.

This modification did not change names, API, and behavior of existing OCCT classes, thus in general the code based on OCCT 6.x should compile and work fine with OCCT 7.0. However, due to redesign of basic mechanisms (CDL generic classes, Handles and RTTI) using C++ templates, some changes may be necessary in the code when porting to OCCT 7.0, as described below.

WOK is not necessary anymore for building OCCT from sources, though it still can be used in a traditional way – auxiliary files required for that are preserved. The recommended method for building OCCT 7.x is CMake, see Building with CMake. The alternative solution is to use project files generated by OCCT legacy tool genproj, see Building with MS Visual C++, Building with Code::Blocks, and Building with Xcode.

Automatic upgrade

Most of typical changes required for upgrading code for OCCT 7.0 can be done automatically using the upgrade tool included in OCCT 7.0. This tool is a Tcl script, thus Tcl should be available on your workstation to run it.

Example:

$ tclsh
% source <path_to_occt>/adm/upgrade.tcl
% upgrade -recurse -all -src=<path_to_your_sources>

On Windows, the helper batch script upgrade.bat can be used, provided that Tcl is either available in PATH, or configured via custom.bat script (for instance, if you use OCCT installed from Windows installer package).
Start it from the command prompt:

```
cmd> <path_to_occt>\upgrade.bat -recurse -all -inc=<path_to_occt>\inc -src=<path_to_your_sources> [options]
```

Run the upgrade tool without arguments to see the list of available options.

The upgrade tool performs the following changes in the code.

1. Replaces macro `DEFINE_STANDARD_RTTI` by `DEFINE_STANDARD_RTTI_EXT`, with second argument indicating base class for the main argument class (if inheritance is recognized by the script):

   \[
   \text{DEFINE_STANDARD_RTTI(Class)} \rightarrow \text{DEFINE_STANDARD_RTTI_EXT(Class, Base)}
   \]

   **Note**

   If macro `DEFINE_STANDARD_RTTI` with two arguments (used in intermediate development versions of OCCT 7.0) is found, the script will convert it to either `DEFINE_STANDARD_RTTI_EXT` or `DEFINE_STANDARD_RTTI_INLINE`. The former case is used if current file is header and source file with the same name is found in the same folder. In this case, macro `IMPLEMENT_STANDARD_RTTI` is injected in the corresponding source file. The latter variant defines all methods for RTTI as inline, and does not require `IMPLEMENT_STANDARD_RTTI_EXT` macro.

2. Replaces forward declarations of collection classes previously generated from CDL generics (defined in `TCollection` package) by inclusion of the corresponding header:

   \[
   \text{class TColStd_Array1OfReal;} \rightarrow \text{#include <TColStd_Array1OfReal.hxx>}
   \]

3. Replaces underscored names of `Handle` classes by usage of a macro:

   \[
   \text{Handle\_Class} \rightarrow \text{Handle(Class)}
   \]

   This change is not applied if the source or header file is recognized as containing the definition of Qt class with signals or slots, to avoid
possible compilation errors of MOC files caused by inability of MOC to recognize macros (see http://doc.qt.io/qt-4.8/signalsandslots.html). The file is considered as defining a Qt object if it contains strings Q_OBJECT and either slots: or signals:.

4. Removes forward declarations of classes with names Handle(C) or Handle_C, replacing them either by forward declaration of its argument class, or (for files defining Qt objects) #include statement for a header with the name of the argument class and extension .hxx:

```cpp
class Handle(TColStd_HArray1OfReal); -> #include <TColStd_HArray1OfReal.hxx>
```

5. Removes #includes of files Handle_.hxx that have disappeared in OCCT 7.0:

```cpp
#include <Handle_Geom_Curve.hxx> ->
```

6. Removes typedef statements that use Handle macro to generate the name:

```cpp
typedef NCollection_Handle<Message_Msg>
Handle(Message_Msg); ->
```

7. Converts C-style casts applied to Handles into calls to DownCast() method:

```cpp
((Handle(A)&)b) -> Handle(A)::DownCast(b)
(Handle(A)&)b -> Handle(A)::DownCast(b)
(*((Handle(A)*)&b)) -> Handle(A)::DownCast(b)
*(((Handle(A)*)&b)) -> Handle(A)::DownCast(b)
(*((Handle(A)*)&b)) -> Handle(A)::DownCast(b)
```

8. Moves Handle() macro out of namespace scope:

```cpp
Namespace::Handle(Class) -> 
Handle(Namespace::Class)
```

9. Converts local variables of reference type, which are initialized by a temporary object returned by call to DownCast(), to the variables of non-reference type (to avoid using references to destroyed memory):

```cpp
const Handle(A)& a = Handle(B)::DownCast (b); -> 
Handle(A) a (Handle(B)::DownCast (b));
```

10. Adds #include for all classes used as argument to macro STANDARD_TYPE(), except for already included ones;

11. Removes uses of obsolete macros IMPLEMENT_DOWNCAST and
IMPLEMENT_STANDARD_..., except
IMPLEMENT_STANDARD_RTTIEXT.

Note
If you plan to keep compatibility of your code with older versions of OCCT, add option -compat to avoid this change. See also Preserving compatibility with OCCT 6.x.

As long as the upgrade routine runs, some information messages are sent to the standard output. In some cases the warnings or errors like the following may appear:

```
Error in {HEADER_FILE}: Macro DEFINE_STANDARD_RTTI used for class {CLASS_NAME} whose declaration is not found in this file, cannot fix
```

Be sure to check carefully all reported errors and warnings, as the corresponding code will likely require manual corrections. In some cases these messages may help you to detect errors in your code, for instance, cases where DEFINE_STANDARD_RTTI macro is used with incorrect class name as an argument.

Possible compiler errors

Some situations requiring upgrade cannot be detected and / or handled by the automatic procedure. If you get compiler errors or warnings when trying to build the upgraded code, you will need to fix them manually. The following paragraphs list known situations of this kind.

Missing header files

The use of handle objects (construction, comparison using operators == or !=, use of function STANDARD_TYPE() and method DownCast()) now requires the type of the object pointed by Handle to be completely known at compile time. Thus it may be necessary to include header of the corresponding class to make the code compilable.

For example, the following lines will fail to compile if Geom_Line.hxx is not included:
Handle(Geom_Line) aLine = 0;
if (aLine != aCurve) {...}
if (aCurve->IsKind(STANDARD_TYPE(Geom_Line))) {...}
aLine = Handle(Geom_Line)::DownCast(aCurve);

Note that it is not necessary to include header of the class to declare Handle to it. However, if you define a class $B$ that uses Handle($A$) in its fields, or contains a method returning Handle($A$), it is advisable to have header defining $A$ included in the header of $B$. This will eliminate the need to include the header $A$ in each source file where class $B$ is used.

**Ambiguity of calls to overloaded functions**

This issue appears in the compilers that do not support default arguments in template functions (known cases are Visual C++ 10 and 11): the compiler reports an ambiguity error if a handle is used in the argument of a call to the function that has two or more overloaded versions, receiving handles to different types. The problem is that operator `const handle<T2>&` is defined for any type $T2$, thus the compiler cannot make the right choice.

Example:

```cpp
void func (const Handle(Geom_Curve)&);
void func (const Handle(Geom_Surface)&);

Handle(Geom_TrimmedCurve) aCurve = new Geom_TrimmedCurve (...);
func (aCurve); // ambiguity error in VC++ 10
```

Note that this problem can be avoided in many cases if macro `OCCT_HANDLE_NOCAST` is used, see [below](#).

To resolve this ambiguity, change your code so that argument type should correspond exactly to the function signature. In some cases this can be done by using the relevant type for the corresponding variable, like in the example above:

```cpp
Handle(Geom_Curve) aCurve = new Geom_TrimmedCurve
```
Other variants consist in assigning the argument to a local variable of the correct type and using the direct cast or constructor:

```cpp
const Handle(Geom_Curve)& aGCurve (aTrimmedCurve);
func (aGCurve);  // OK - argument has exact type
func (static_cast(aCurve));  // OK - direct cast
func (Handle(Geom_Curve)(aCurve));  // OK - temporary handle is constructed
```

Another possibility consists in defining additional template variant of the overloaded function causing ambiguity, and using SFINAE to resolve the ambiguity. This technique can be illustrated by the definition of the template variant of method `IGESData_IGESWriter::Send()`.

**Lack of implicit cast to base type**

As the cast of a handle to the reference to another handle to the base type has become a user-defined operation, the conversions that require this cast together with another user-defined cast will not be resolved automatically by the compiler.

For example:

```cpp
Handle(Geom_Geometry) aC = GC_MakeLine (p, v);  // compiler error
```

The problem is that the class `GC_MakeLine` has a user-defined conversion to `const Handle(Geom_TrimmedCurve)&`, which is not the same as the type of the local variable `aC`.

To resolve this, use method `Value()`:

```cpp
Handle(Geom_Geometry) aC = GC_MakeLine (p, v).Value();  // ok
```

or use variable of the appropriate type:

```cpp
Handle(Geom_TrimmedCurve) aC = GC_MakeLine (p, v);  // ok
```
A similar problem appears with GCC compiler, when \textit{const} handle to derived type is used to construct handle to base type via assignment (and in some cases in return statement), for instance:

```cpp
const Handle(Geom_Line) aLine;
Handle(Geom_Curve) c1 = aLine; // GCC error
Handle(Geom_Curve) c2 (aLine); // ok
```

This problem is specific to GCC and it does not appear if macro \textit{OCCT\_HANDLE\_NOCAST} is used, see \textit{below}.

\section*{Incorrect use of \texttt{STANDARD\_TYPE} and \texttt{Handle} macros}

You might need to clean your code from incorrect use of macros \texttt{STANDARD\_TYPE() and Handle()}. 

1. Explicit definitions of static functions with names generated by macro \texttt{STANDARD\_TYPE()}, which are artifacts of old implementation of RTTI, should be removed.

   Example:

   ```cpp
   const Handle(Standard_Type)&
   STANDARD_TYPE(math\_GlobOptMin)
   {
     static Handle(Standard_Type) _atyp = new
     Standard_Type ("math\_GlobOptMin", sizeof
     (math\_GlobOptMin));
     return _atyp;
   }
   ```

2. Incorrect location of closing parenthesis of \texttt{Handle()} macro that was not detectable in OCCT 6.x will cause a compiler error and must be corrected.

   Example (note misplaced closing parenthesis):

   ```cpp
   aBSpline = Handle(
   ```
Use of class Standard_AncestorIterator

Class Standard_AncestorIterator has been removed; use method Parent() of Standard_Type class to parse the inheritance chain.

Absence of cast to Standard_Transient*

Handles in OCCT 7.0 do not have the operator of conversion to Standard_Transient*, which was present in earlier versions. This is done to prevent possible unintended errors like this:

```cpp
Handle(Geom_Line) aLine = ...;
Handle(Geom_Surface) aSurf = ...;
... if (aLine == aSurf) {...} // will cause a compiler error in OCCT 7.0, but not OCCT 6.x
```

The places where this implicit cast has been used should be corrected manually. The typical situation is when Handle is passed to stream:

```cpp
Handle(Geom_Line) aLine = ...;
os << aLine; // in OCCT 6.9.0, resolves to operator << (void*)
```

Call method get() explicitly to output the address of the Handle.

Method DownCast for non-base types

Method DownCast() in OCCT 7.0 is made templated; if its argument is not a base class, "deprecated" compiler warning is generated. This is done to prevent possible unintended errors like this:

```cpp
Handle(Geom_Surface) aSurf = ;
Handle(Geom_Line) aLine =
    Handle(Geom_Line)::DownCast (aSurf); // will cause a compiler warning in OCCT 7.0, but not OCCT 6.x
```
The places where this cast has been used should be corrected manually.

If down casting is used in a template context where the argument can have the same or unrelated type so that `DownCast()` may be not available in all cases, use C++ `dynamic_cast<>` instead, e.g.:

```cpp
template <class T>
bool CheckLine (const Handle(T) theArg)
{
    Handle(Geom_Line) aLine = dynamic_cast<Geom_Line>(theArg.get());
    ...
}
```

### Possible runtime problems

Here is the list of known possible problems at run time after the upgrade to OCCT 7.0.

#### References to temporary objects

In previous versions, the compiler was able to detect the situation when a local variable of a "reference to a Handle" type is initialized by temporary object, and ensured that lifetime of that object is longer than that of the variable. In OCCT 7.0 with default options, it will not work if types of the temporary object and variable are different (due to involvement of user-defined type cast), thus such temporary object will be destroyed immediately.

This problem does not appear if macro `OCCT_HANDLE_NOCAST` is used during compilation, see below.

**Example:**

```cpp
// note that DownCast() returns new temporary object!
const Handle(Geom_BoundedCurve)& aBC = Handle(Geom_TrimmedCurve)::DownCast(aCurve);
aBC->Transform (T); // access violation in OCCT 7.0
```
Option to avoid cast of handle to reference to base type

In OCCT 6.x and earlier versions the handle classes formed a hierarchy echoing the hierarchy of the corresponding object classes. This automatically enabled the possibility to use the handle to a derived class in all contexts where the handle to a base class was needed, e.g. to pass it in a function by reference without copying:

```cpp
Standard_Boolean GetCurve (Handle(Geom_Curve)& theCurve);
....
Handle(Geom_Line) aLine;
if (GetCurve (aLine)) {
    // use aLine, unsafe
}
```

This feature was used in multiple places in OCCT and dependent projects. However it is potentially unsafe: in the above example no checks are done at compile time or at run time to ensure that the type assigned to the argument handle is compatible with the type of the handle passed as argument. If an object of incompatible type (e.g. Geom_Circle) is assigned to `theCurve`, the behavior will be unpredictable.

For compatibility with the existing code, OCCT 7.0 keeps this possibility by default, providing operators of type cast to the handle to a base type. However, this feature is unsafe and in specific situations it may cause compile-time or run-time errors as described above.

To provide a safer behavior, this feature can be disabled by a compile-time macro `OCCT_HANDLE_NOCAST`. When it is used, constructors and assignment operators are defined (instead of type cast operators) to convert handles to a derived type into handles to a base type. This implies creation of temporary objects and hence may be more expensive at run time in some circumstances, however this way is more standard, safer, and in general recommended.

The code that relies on the possibility of casting to base should be amended to always use the handle of argument type in function call and to use `DownCast()` to safely convert the result to the desired type. For
instance, the code from the example below can be changed as follows:

```cpp
Handle(Geom_Line) aLine;
Handle(Geom_Curve) aCurve;
if (GetCurve (aCurve) && !(aLine =
    Handle(Geom_Line)::DownCast (aCurve)).IsNull())
{
    // use aLine safely
}
```

Preserving compatibility with OCCT 6.x

If you like to preserve the compatibility of your application code with OCCT versions 6.x even after the upgrade to 7.0, consider the following suggestions:

1. If your code used sequences of macros
   `IMPLEMENT_STANDARD_...` generated by WOK, replace them by single macro `IMPLEMENT_STANDARD_RTTIEXT`
2. When running automatic upgrade tool, add option `-compat`.
3. Define macros `DEFINE_STANDARD_RTTIEXT` and `DEFINE_STANDARD_RTTI_INLINE` when building with previous versions of OCCT, resolving to `DEFINE_STANDARD_RTTI` with single argument

   Example:

   ```
   #if OCC_VERSION_HEX < 0x070000
   #define DEFINE_STANDARD_RTTIEXT(C1,C2)
       DEFINE_STANDARD_RTTI(C1)
   #endif
   #define DEFINE_STANDARD_RTTI_INLINE(C1,C2)
       DEFINE_STANDARD_RTTI(C1)
   ```

Applications based on CDL and WOK

If your application is essentially based on CDL, and you need to upgrade it to OCCT 7.0, you will very likely need to convert your application code to non-CDL form. This is a non-trivial effort; the required actions would
depend strongly on the structure of the code and used CDL features.

The upgrade script and sources of a specialized WOK version used for OCCT code upgrade can be found in WOK Git repository in branch CR0_700_2.

Contact us if you need more help.
Separation of B-Spline cache

Implementation of NURBS curves and surfaces has been revised: the cache of polynomial coefficients, which is used to accelerate the calculation of values of a B-spline, has been separated from data objects `Geom2d_BSplineCurve`, `Geom_BSplineCurve` and `Geom_BSplineSurface` into the dedicated classes `BSplCLib_Cache` and `BSplSLib_Cache`.

The benefits of this change are:

- Reduced memory footprint of OCCT shapes (up to 20% on some cases)
- Possibility to evaluate the same B-Spline concurrently in parallel threads without data races and mutex locks

The drawback is that direct evaluation of B-Splines using methods of curves and surfaces becomes slower due to the absence of cache. The slow-down can be avoided by using adaptor classes `Geom2dAdaptor_Curve`, `GeomAdaptor_Curve` and `GeomAdaptor_Surface`, which now use cache when the curve or surface is a B-spline.

OCCT algorithms have been changed to use adaptors for B-spline calculations instead of direct methods for curves and surfaces. The same changes (use of adaptors instead of direct call to curve and surface methods) should be implemented in relevant places in the applications based on OCCT to get the maximum performance.
Structural result of Boolean operations

The result of Boolean operations became structured according to the structure of the input shapes. Therefore it may impact old applications that always iterate on direct children of the result compound assuming to obtain solids as iteration items, regardless of the structure of the input shapes. In order to get always solids as iteration items it is recommended to use TopExp_Explorer instead of TopoDS_Iterator.
BRepExtrema_ExtCC finds one solution only

Extrema computation between non-analytical curves in shape-shape distance calculation algorithm has been changed in order to return only one solution. So, if e.g. two edges are created on parallel b-spline curves the algorithm BRepExtrema_DistShapeShape will return only one solution instead of enormous number of solutions. There is no way to get algorithm working in old manner.
Removal of SortTools package

Package SortTools has been removed. The code that used the tools provided by that package should be corrected manually. The recommended approach is to use sorting algorithms provided by STL.

For instance:

```cpp
#include <SortTools_StraightInsertionSortOfReal.hxx>
#include <SortTools_ShellSortOfReal.hxx>
#include <TCollection_CompareOfReal.hxx>
...
TCollection_Array1OfReal aValues = ...;
...
TCollection_CompareOfReal aCompReal;
SortTools_StraightInsertionSortOfReal::Sort(aValues, aCompReal);
```

can be replaced by:

```cpp
#include <algorithm>
...
TCollection_Array1OfReal aValues = ...;
...
std::stable_sort (aValues.begin(), aValues.end());
```
On-screen objects and ColorScale

The old mechanism for rendering Underlay and Overlay on-screen 2D objects based on Visual3d_Layer and immediate drawing model (uncached and thus slow) has been removed. Classes Aspect_Clayer2d, OpenGL_GraphicDriver_Layer, Visual3d_Layer, Visual3d_LayerItem, V3d_LayerMgr and V3d_LayerMgrPointer have been deleted.

General AIS interactive objects with transformation persistence flag Graphic3d_TMF_2d can be used as a replacement of Visual3d_LayerItem. The anchor point specified for transformation persistence defines the window corner of (or center in case of (0, 0) point). To keep on-screen 2D objects on top of the main screen, they can be assigned to the appropriate Z-layer. Predefined Z-layers Graphic3d_ZLayerId_TopOSD and Graphic3d_ZLayerId_BotOSD are intended to replace Underlay and Overlay layers within the old API.

ColorScale object previously implemented using Visual3d_LayerItem has been moved to a new class AIS_ColorScale, with width and height specified explicitly. The property of V3d_View storing the global ColorScale object has been removed with associated methods V3d_View::ColorScaleDisplay(), V3d_View::ColorScaleErase(), V3d_View::ColorScaleIsDisplayed() and V3d_View::ColorScale() as well as the classes V3d_ColorScale, V3d_ColorScaleLayerItem and Aspect_ColorScale. Here is an example of creating ColorScale using the updated API:

```cpp
Handle(AIS_ColorScale) aCS = new AIS_ColorScale(); // configuring
Standard_Integer aWidth, aHeight;
aView->Window()->Size (aWidth, aHeight);
aCS->SetSize (aWidth, aHeight);
aCS->SetRange (0.0, 10.0);
aCS->SetNumberOfIntervals (10);
// displaying
aCS->SetZLayer (Graphic3d_ZLayerId_TopOSD);
aCS->SetTransformPersistence (Graphic3d_TMF_2d, gp_Pnt (-1,-1,0));
```
To see how 2d objects are implemented in OCCT you can call Draw commands `vcolorscale`, `vlayerline` or `vdrawtext` (with -2d option). Draw command `vcolorscale` now requires the name of ColorScale object as argument. To display this object use command `vdisplay`. For example:

```
pload VISUALIZATION
vinit
vcolorscale cs -demo
pload MODELING
box b 100 100 100
vdisplay b
vsetdispmode 1
vfit
vlayerline 0 300 300 300 10
vdrawtext t "2D-TEXT" -2d -pos 0 150 0 -color red
```

Here is a small example in C++ illustrating how to display a custom AIS object in 2d:

```
Handle(AIS_InteractiveContext) aContext = ...;
Handle(AIS_InteractiveObject) anObj = ...; // create an AIS object
anObj->SetZLayer(Graphic3d_ZLayerId_TopOSD); // display object in overlay
anObj->SetTransformPersistence (Graphic3d_TMF_2d,
    gp_Pnt (-1,-1,0)); // set 2d flag, coordinate origin is set to down-left corner
aContext->Display (anObj); // display the object
```
UserDraw and Visual3d

Visual3d package

Package Visual3d implementing the intermediate layer between high-level V3d classes and low-level OpenGL classes for views and graphic structures management has been dropped.

The OpenGL_View inherits from the new class Graphic3d_CView. Graphic3d_CView is an interface class that declares abstract methods for managing displayed structures, display properties and a base layer code that implements computation and management of HLR (or more broadly speaking view-dependent) structures.

In the new implementation it takes place of the eliminated Visual3d_View. As before the instance of Graphic3d_CView is still completely managed by V3d_View classes. It can be accessed through V3d_View interface but normally it should not be required as all its methods are completely wrapped.

In more details, a concrete specialization of Graphic3d_CView is created and returned by the graphical driver on request. Right after the creation the views are directly used for setting rendering properties and adding graphical structures to be displayed.

The rendering of graphics is possible after mapping a window and activating the view. The direct setting of properties obsoletes the use of intermediate structures with display parameter like Visual3d_ContextView, etc. This means that the whole package Visual3d becomes redundant.

The functionality previously provided by Visual3d package has been redesigned in the following way:

- The management of display of structures has been moved from Visual3d_ViewManager into Graphic3d_StructureManager.
- The class Visual3d_View has been removed. The management of computed structures has been moved into the base layer of Graphic3d_CView.
All intermediate structures for storing view parameters, e.g. `Visual3d_ContextView`, have been removed. The settings are now kept by instances of `Graphic3d_CView`.

The intermediate class `Visual3d_Light` has been removed. All light properties are stored in `Graphic3d_CLight` structure, which is directly accessed by instances of `V3d_Light` classes.

All necessary enumerations have been moved into `Graphic3d` package.

### Custom OpenGL rendering and UserDraw

Old APIs based on global callback functions for creating `UserDraw` objects and for performing custom OpenGL rendering within the view have been dropped. `UserDraw` callbacks are no more required since `OpenGL_Group` now inherits `Graphic3d_Group` and thus can be accessed directly from `AIS_InteractiveObject`:

```cpp
/// Class implementing custom OpenGL element.
class UserDrawElement : public OpenGl_Element {};

/// Implementation of virtual method
AIS_InteractiveObject::Compute().
void UserDrawObject::Compute (const
   Handle(PrsMgr_PresentationManager3d)& thePrsMgr,
   const
   Handle(Prs3d_Presentation)& thePrs,
   const Standard_Integer
   theMode)
{
   Graphic3d_Vec4 aBndMin (myCoords[0], myCoords[1],
     myCoords[2], 1.0f);
   Graphic3d_Vec4 aBndMax (myCoords[3], myCoords[4],
     myCoords[5], 1.0f);

   // casting to OpenGL_Group should be always true as
   // far as application uses OpenGL_GraphicDriver for
   // rendering
   Handle(OpenGl_Group) aGroup =
```
To perform a custom OpenGL code within the view, it is necessary to inherit from class `OpenGl_View`. See the following code sample:

```cpp
// Custom view.
class UserView : public OpenGl_View
{
public:
    // Override rendering into the view.
    virtual void render (Graphic3d_Camera::Projection theProjection,
                        OpenGL_FrameBuffer* theReadDrawFbo,
                        const Standard_Boolean theToDrawImmediate)
    {
        OpenGl_View::render (theProjection,
                             theReadDrawFbo,
                             theToDrawImmediate);
        if (theToDrawImmediate)
        {
            return;
        }

        // perform custom drawing
    }
};
```
const Handle(OpenGL_Context)& aCtx = myWorkspace->GetGlContext();
GLfloat aVerts[3] = { 0.0f, 0.0f, 0.0f };
aCtx->core20->glEnableClientState(GL_VERTEX_ARRAY);
aCtx->core20->glVertexPointer(3, GL_FLOAT, 0, aVerts);
aCtx->core20->glDrawArrays(GL_POINTS, 0, 1);
aCtx->core20->glDisableClientState(GL_VERTEX_ARRAY);
}
};

//! Custom driver for creating UserView.
class UserDriver : public OpenGL_GraphicDriver
{
public:
    //! Create instance of own view.
    virtual Handle(Graphic3d_CView) CreateView (const Handle(Graphic3d_StructureManager)& theMgr)
    Standard_OVERRIDE
    {
        Handle(UserView) aView = new UserView (theMgr, this, myCaps, myDeviceLostFlag, &myStateCounter);
        myMapOfView.Add (aView);
        for (TColStd_SequenceOfInteger::Iterator aLayerIt (myLayerSeq); aLayerIt.More(); aLayerIt.Next())
        {
            const Graphic3d_ZLayerId aLayerId = aLayerIt.Value();
            const Graphic3d_ZLayerSettings& aSettings = myMapOfZLayerSettings.Find (aLayerId);
            aView->AddZLayer (aLayerId);
            aView->SetZLayerSettings (aLayerId, aSettings);
        }
    }
}
return aView;
}
Deprecation of Local Context

The conception of Local Context has been deprecated. The related classes, e.g. AIS_LocalContext, and methods (AIS_InteractiveContext::OpenLocalContext() and others) will be removed in a future OCCT release.

The main functionality provided by Local Context - selection of object subparts - can be now used within Neutral Point without opening any Local Context.

The property SelectionMode() has been removed from the class AIS_InteractiveObject. This property contradicts to selection logic, since it is allowed to activate several Selection modes at once. Therefore keeping one selection mode as object field makes no sense. Applications that used this method should implement selection mode caching at application level, if it is necessary for some reason.
Separation of visualization part from TKCAF

Visualization CAF attributes have been moved into a new toolkit TKVCAF. If your application uses the classes from TPrsStd package then add link to TKVCAF library.

Version numbers of BinOCAF and XmlOCAF formats are incremented; new files cannot be read by earlier versions of OCCT.

Before loading the OCAF files saved by previous versions and containing TPrsStd_AISPresentation attribute it is necessary to define the environment variable CSF_MIGRATION_TYPES, pointing to file src/StdResources/MigrationSheet.txt. When using documents loaded from a file, make sure to call method TPrsStd_AISViewer::New() prior to accessing TPrsStd_AISPresentation attributes in this document as that method creates them.
Correction of interpretation of Euler angles in gp_Quaternion

Conversion of gp_Quaternion to and from intrinsic Tait-Bryan angles (including gp_YawPitchRoll) is fixed.

Before that fix the sequence of rotation axes was opposite to the intended; e.g. gp_YawPitchRoll (equivalent to gp_Intrinsic_ZYX) actually defined intrinsic rotations around X, then Y, then Z. Now the rotations are made in the correct order.

The applications that use gp_Quaternion to convert Yaw-Pitch-Roll angles (or other intrinsic Tait-Bryan sequences) may need to be updated to take this change into account.
Zoom Persistent Selection

Zoom persistent selection introduces a new structure \texttt{Graphic3d\_TransformPers} to transform persistence methods and parameters and a new class \texttt{Graphic3d\_WorldViewProjState} to refer to the camera transformation state. You might need to update your code to deal with the new classes if you were using the related features. Please, keep in mind the following:

- \texttt{Graphic3d\_Camera::ModelViewState} has been renamed to \texttt{Graphic3d\_Camera::WorldViewState}.
- Transformation matrix utilities from \texttt{OpenGl\_Utils} namespace have been moved to \texttt{Graphic3d\_TransformUtils} and \texttt{Graphic3d\_TransformUtils.hxx} header respectively.
- Matrix stack utilities from \texttt{OpenGl\_Utils} namespace have been moved to \texttt{OpenGl\_MatrixStack} class and \texttt{OpenGl\_MatrixStack.hxx} header respectively.
- \texttt{OpenGl\_View} methods \texttt{Begin/EndTransformPersistence} have been removed. Please, use \texttt{Graphic3d\_TransformPers::Apply()} instead to apply persistence to perspective and world-view projection matrices.
Texture mapping of objects

Textured objects now have the priority over the environment mapping.

Redundant enumerations `V3d_TypeOfSurface` and `Graphic3d_TypeOfSurface`, class `OpenGL_SurfaceDetailState`, the corresponding methods from `Graphic3d_CView`, `OpenGL_ShaderManager`, `OpenGL_View`, `V3d_View` and `V3d_Viewer` have been deleted. Draw command `VSetTextureMode` has been deleted.
Shape presentation builders

Presentation tools for building Wireframe presentation have been refactored to eliminate duplicated code and interfaces. Therefore, the following classes have been modified:

- *StdPrs_WDeflectionShape* and *Prs3d_WFShape* have been removed. *StdPrs_WFShape* should be used instead.
- *StdPrs_ToolShadedShape* has been renamed to *StdPrs_ToolTriangulatedShape.*
Upgrade to OCCT 7.1.0

Presentation attributes

This section should be considered if application defines custom presentations, i.e. inherited from \textit{AIS\_InteractiveObject}. The previous versions of OCCT have three levels for defining presentation properties (e.g. colors, materials, etc.):

1. For the entire structure - \textit{Graphic3d\_Structure / Prs3d\_Presentation}.
2. For a specific group of primitives - \textit{Graphic3d\_Group::SetGroupPrimitivesAspect()} overriding structure aspects.
3. For a specific primitive array within the graphic group - \textit{Graphic3d\_Group::SetPrimitivesAspect()}.

The structure level has de facto not been used for a long time since OCCT presentations always define aspects at the graphic group level (overriding any structure aspects). Within this OCCT release, structure level of aspects has been completely removed. In most cases the application code should just remove missing methods. In those rare cases, when this functionality was intentionally used, the application should explicitly define aspects to the appropriate graphic groups.

Note that defining several different aspects within the same graphic group should also be avoided in the application code since it is a deprecated functionality which can be removed in further releases. \textit{Graphic3d\_Group::SetGroupPrimitivesAspect()} should be the main method defining presentation attributes.

The implementation of \textit{Graphic3d\_Group::SetGroupPrimitivesAspect()} has been changed from copying aspect values to keeping the passed object. Although it was not documented, previously it was possible to modify a single aspect instance, like \textit{Graphic3d\_AspectFillArea3d} and set it to multiple groups. Now such code would produce an unexpected result and therefore should be updated to create the dedicated aspect instance.
Typedefs

The following type definitions in OCCT has been modified to use C++11 types:

- `Standard_Boolean` is now `bool` (previously `unsigned int`).
- `Standard_ExtCharacter` is now `char16_t` (previously `short`).
- `Standard_ExtString` is now `const char16_t` (previously `const short`).
- `Standard_Utf16Char` is now `char16_t` (previously `uint16_t` for compatibility with old compilers).
- `Standard_Utf32Char` is now `char32_t` (previously `uint32_t` for compatibility with old compilers).

For most applications this change should be transparent on the level of source code. Binary compatibility is not maintained, as `bool` has a different size in comparison with `unsigned int`. 
Programmable Pipeline

Fixed-function pipeline has been already deprecated since OCCT 7.0.0. Release 7.1.0 disables this functionality by default in favor of Programmable Pipeline (based on GLSL programs).

Method `V3d_View::Export()`, based on `gl2ps` library, requires fixed pipeline and will return error if used with default settings. Applications should explicitly enable fixed pipeline by setting `OpenGL_Caps::ffpEnable` flag to TRUE within `OpenGL_GraphicDriver::ChangeOptions()` before creating the viewer to use `V3d_View::Export()`. This method is declared as deprecated and will be removed in one of the the next OCCT releases. The recommended way to generate a vector image of a 3D model or scene is to use an application-level solution independent from OpenGL.
Transformation persistence

The behavior of transformation persistence flags `Graphic3d_TMF_ZoomPers` and `Graphic3d_TMF_TriedronPers` has been changed for consistency with a textured fixed-size 2D text. An object with these flags is considered as defined in pixel units, and the presentation is no more scaled depending on the view height. The applications that need to scale such objects depending on viewport size should update them manually.

Flags `Graphic3d_TMF_PanPers` and `Graphic3d_TMF_FullPers` have been removed. `Graphic3d_TMF_TriedronPers` or `Graphic3d_TMF_2d` can be used instead depending on the context.

`Graphic3d_TransModeFlags` is not an integer bitmask anymore - enumeration values should be specified instead. Several transformation persistence methods in `PrsMgr_PresentableObject` have been marked deprecated. Transformation persistence should be defined using `Graphic3d_TransformPers` constructor directly and passed by a handle, not value.
Dynamic highlight and selection properties

Release 7.1.0 introduces Graphic3d_HighlightStyle - an entity that allows flexible customization of highlighting parameters (such as highlighting method, color, and transparency). Therefore, the signatures of the following methods related to highlighting:

- AIS_InteractiveContext::Hilight();
- AIS_InteractiveContext::HilightWithColor();
- PrsMgr_PresentationManager::Color();
- SelectMgr_EntityOwner::HilightWithColor(); have been changed to receive Graphic3d_HighlightStyle instead of Quantity_Color.

Method AIS_InteractiveContext::Hilight is now deprecated and highlights the interactive object with selection style.

A group of methods AIS_InteractiveContext::IsHilighted has changed its behavior - now they only check highlight flags of the object or the owner in the global status. If the highlight color is required on the application level, it is necessary to use overloaded methods AIS_InteractiveContext::HighlightStyle for the owner and the object.

The following methods have been replaced in AIS_InteractiveContext class:

- HilightColor and SetHilightColor by HighlightStyle and SetHighlightStyle;
- SelectionColor setter and getter by SelectionStyle and SetSelectionStyle.

The API of Prs3d_Drawer has been extended to allow setting up styles for both dynamic selection and highlighting. Therefore, it is possible to change the highlight style of a particular object on the application level via SelectMgr_SelectableObject::HilightAttributes() and process it in the entity owner.
Correction in TObj_Model class

Methods TObj_Model::SaveAs and TObj_Model::Load now receive TCollection_ExtendedString filename arguments instead of char*. UTF-16 encoding can be used to pass file names containing Unicode symbols.
Redundant environment variables

The following environment variables have become redundant:

- `CSF_UnitsLexicon` and `CSF_UnitsDefinition` are no more used. Units definition (`UnitsAPI/Lexi_Expr.dat` and `UnitsAPI/Units.dat`) is now embedded into source code.
- `CSF_XSMessages` and `CSF_XHMessage` are now optional. English messages (`XSMmessage/*XSTEP.us` and `SHMessage/*SHAPE.us`) are now embedded into source code and automatically loaded when environment variables are not set.
- `CSF_ShadersDirectory` is not required any more, though it still can be used to load custom shaders. Mandatory GLSL resources are now embedded into source code.
- `CSF_PluginDefaults` and other variables pointing to OCAF plugin resources (`CSF_StandardDefaults`, `CSF_XCAFDefaults`, `CSF_StandardLiteDefaults` and `CSF_XmlOcafResource`) are not necessary if method `TDocStd_Application::DefineFormat()` is used to enable persistence of OCAF documents.

Other environment variables still can be used to customize behavior of relevant algorithms but are not necessary any more (all required resources are embedded).
Removed features

The following obsolete features have been removed:

- **Anti-aliasing API** `V3d_View::SetAntialiasingOn()`. This method was intended to activate deprecated OpenGL functionality `GL_POLYGON_SMOOTH`, `GL_LINE_SMOOTH` and `GL_POINT_SMOOTH`. Instead of the old API, the application should request MSAA buffers for anti-aliasing by assigning `Graphic3d_RenderingParams::NbMsaaSamples` property of the structure returned by `V3d_View::ChangeRenderingParams()`.
- **`Prs3d_Drawer::ShadingAspectGlobal()`** flag has been removed as not used. The corresponding calls can be removed safely from the application code.
- The methods managing ZClipping planes and ZCueing: `V3d_View::SetZClippingType()`, `V3d_View::SetZCueingOn()`, etc. have been removed. ZClipping planes can be replaced by general-purpose clipping planes (the application should update plane definition manually).
- The 3D viewer printing API `V3d_View::Print()` has been removed. This functionality was available on Windows platforms only. The applications should use the general image dump API `V3d_View::ToPixMap()` and manage printing using a platform-specific API at the application level. Text resolution can be managed by rendering parameter `Graphic3d_RenderingParams::Resolution`, returned by `V3d_View::ChangeRenderingParams()`.
- Methods `PrsMgr_PresentationManager::BoundBox`, `PrsMgr_PresentationManager::Hilight` and `SelectMgr_EntityOwner::Hilight` have been removed as not used. The corresponding method in custom implementations of `SelectMgr_EntityOwner` can be removed safely.
- **Class** `NCollection_QuickSort` has been removed. The code that used the tools provided by that class should be corrected manually. The recommended approach is to use sorting algorithms provided by STL (std::sort). See also Removal of SortTools package above.
- **Package** `Dico`. The code that used the tools provided by that package should be corrected manually. The recommended approach
is to use *NCollection_DataMap* and *NCollection_IndexedDataMap* classes.
Other changes

The following classes have been changed:

- **BVH_Sorter** class has become abstract. The list of arguments of both *Perform* methods has been changed and the methods became pure virtual.
- **Extrema_FuncExtPS** has been renamed to **Extrema_FuncPSNorm**.
- The default constructor and the constructor taking a point and a surface have been removed from class **Extrema_GenLocateExtPS**. Now the only constructor takes the surface and optional tolerances in U and V directions. The new method *Perform* takes the point with the start solution and processes it. The class has become not assignable and not copy-constructable.
- Constructors with arguments *(const gp_Ax22d& D, const gp_Pnt2d& F)* have been removed from **GCE2d_MakeParabola**, **gce_MakeParab2d** and **gp_Parab2d**. The objects created with some constructors of class **gp_Parab2d** may differ from the previous version (see the comments in **gp_Parab2d.hxx**). The result returned by **gp_Parab2d::Directrix()** method has an opposite direction in comparison with the previous OCCT versions.
- **BRepTools_Modifier** class now has two modes of work. They are defined by the boolean parameter *MutableInput*, which is turned off by default. This means that the algorithm always makes a copy of a sub-shape (e.g. vertex) if its tolerance is to be increased in the output shape. The old mode corresponds to *MutableInput* turned on. This change may impact an application if it implements a class derived from **BRepTools_Modifier**.
- The second parameter *theIsOuterWire* of method **ShapeAnalysis_Wire::CheckSmallArea** has been removed.
- In class **GeomPlate_CurveConstraint**, two constructors taking boundary curves of different types have been replaced with one constructor taking the curve of an abstract type.
- The last optional argument *RemoveInvalidFaces* has been removed from the constructor of class **BRepOffset_MakeOffset** and method *Initialize*.
- The public method **BOPDS_DS::VerticesOnIn** has been renamed into **SubShapesOnIn** and the new output parameter *theCommonPB* has been added.
Upgrade to OCCT 7.2.0

Removed features

The following obsolete features have been removed:

- `AIS_InteractiveContext::PreSelectionColor()`, `DefaultColor()`, `WasCurrentTouched()`, `ZDetection()`. These properties were unused, and therefore application should remove occurrences of these methods.
- `AIS_InteractiveObject::SelectionPriority()`. These property was not implemented.
- The class `LocOpe_HBuilder` has been removed as obsolete.
- The package `TestTopOpe` has been removed;
- The package `TestTopOpeDraw` has been removed;
- The package `TestTopOpeTools` has been removed.
- The packages `QANewModTopOpe`, `QANewBRepNaming` and `QANewDBRepNaming` have been removed as containing obsolete features.
- The following methods of the `IntPolyh_Triangle` class have been removed as unused:
  - `CheckCommonEdge`
  - `SetEdgeandOrientation`
  - `MultipleMiddleRefinement2`.
- The method `IntPolyh_Triangle::TriangleDeflection` has been renamed to `IntPolyh_Triangle::ComputeDeflection`.
- The following methods of the `IntPolyh_MaillageAffinage` class have been removed as unused:
  - `LinkEdges2Triangles`;
  - `TriangleEdgeContact2`;
  - `StartingPointsResearch2`;
  - `NextStartingPointsResearch2`;
  - `TriangleComparePSP`;
  - `StartPointsCalcul`.
- The method `PerformAdvanced` of the `ShapeConstruct_ProjectCurveOnSurface` class has been removed as unused.
- The method `Perform` of the `ShapeConstruct_ProjectCurveOnSurface`
class is modified:
  ◦ input arguments continuity, maxdeg, nbinterval have been removed as unused;
  ◦ input arguments TolFirst, TolLast have been added at the end of arguments' list.
- Typedefs Quantity_Factor, Quantity_Parameter, Quantity_Ratio, Quantity_Coefficient, Quantity_PlaneAngle, Quantity_Length, V3d_Parameter and V3d_Coordinate have been removed; Standard_Real should be used instead.
Corrections in BRepOffset API

In classes BRepTools_ReShape and ShapeBuild_ReShape, the possibility to process shapes different only by orientation in different ways has been removed. Thus methods Remove() and Replace() do not have any more the last argument 'oriented'; they work always as if Standard_False was passed before (default behavior). Methods ModeConsiderLo() and Apply() with three arguments have been removed.
Corrections in BRepOffset API

Class BRepOffsetAPI_MakeOffsetShape:

- `BRepOffsetAPI_MakeOffsetShape::BRepOffsetAPI_MakeOffsetShape()` - constructor with parameters has been deleted.
- `BRepOffsetAPI_MakeOffsetShape::PerformByJoin()` - method has been added. This method is old algorithm behaviour.

The code below shows new calling procedure:

```cpp
BRepOffsetAPI_MakeOffsetShape OffsetMaker;
OffsetMaker.PerformByJoin(Shape, OffsetValue, Tolerance);
NewShape = OffsetMaker.Shape();
```

Class BRepOffsetAPI_MakeThickSolid:

- `BRepOffsetAPI_MakeThickSolid::BRepOffsetAPI_MakeThickSolid()` - constructor with parameters has been deleted.
- `BRepOffsetAPI_MakeThickSolid::MakeThickSolidByJoin()` - method has been added. This method is old algorithm behaviour.

The code below shows new calling procedure:

```cpp
BRepOffsetAPI_MakeThickSolid BodyMaker;
BodyMaker.MakeThickSolidByJoin(myBody, facesToRemove, -myThickness / 50, 1.e-3);
myBody = BodyMaker.Shape();
```
Highlight style

Management of highlight attributes has been revised and might require modifications from application side:

- New class `Graphic3d_PresentationAttributes` defining basic presentation attributes has been introduced. It's definition includes properties previously defined by class `Graphic3d_HighlightStyle` (`Color`, `Transparency`), and new properties (`Display mode`, `ZLayer`, optional `FillArea aspect`).
- Class `Prs3d_Drawer` now inherits class `Graphic3d_PresentationAttributes`. So that overall presentation attributes are now split into two parts - Basic attributes and Detailed attributes.
- Class `Graphic3d_HighlightStyle` has been dropped. It is now defined as a typedef to `Prs3d_Drawer`. Therefore, highlight style now also includes not only Basic presentation attributes, but also Detailed attributes which can be used by custom presentation builders.
- Highlighting style defined by class `Graphic3d_PresentationAttributes` now provides more options:
  - `Graphic3d_PresentationAttributes::BasicFillAreaAspect()` property providing complete Material definition. This option, when defined, can be used instead of the pair Object Material + Highlight Color.
  - `Graphic3d_PresentationAttributes::ZLayer()` property specifying the Layer where highlighted presentation should be shown. This property can be set to `Graphic3d_ZLayerId_UNKNOWN`, which means that ZLayer of main presentation should be used instead.
  - `Graphic3d_PresentationAttributes::DisplayMode()` property specifying Display Mode for highlight presentation.
- Since Highlight and Selection styles within `AIS_InteractiveContext` are now defined by `Prs3d_Drawer` inheriting from `Graphic3d_PresentationAttributes`, it is now possible to customize default highlight attributes like `Display Mode` and `ZLayer`, which previously could be defined only on Object level.
- Properties `Prs3d_Drawer::HighlightStyle()` and `Prs3d_Drawer::SelectionStyle()` have been removed. Instead, `AIS_InteractiveObject` now defines `DynamicHilightAttributes()` for dynamic highlighting in addition to `HilightAttributes()` used for
highlighting in selected state. Note that
\textit{AIS\_InteractiveObject::HilightAttributes()} and
\textit{AIS\_InteractiveObject::DynamicHilightAttributes()} override
highlighting properties for both - entire object and for part coming
from decomposition. This includes Z-layer settings, which will be the
same when overriding properties through \textit{AIS\_InteractiveObject},
while \textit{AIS\_InteractiveContext::HighlightStyle()} allows customizing
properties for local and global selection independently (with
\texttt{Graphic3d\_ZLayerId\_Top} used for dynamic highlighting of entire
object and \texttt{Graphic3d\_ZLayerId\_Topmost} for dynamic highlighting of
object part by default).

- The following protected fields have been removed from class
\textit{AIS\_InteractiveObject}:
  - \texttt{myOwnColor}, replaced by \texttt{myDrawer->Color()}
  - \texttt{myTransparency}, replaced by \texttt{myDrawer->Transparency()}
  - \texttt{myZLayer}, replaced by \texttt{myDrawer->ZLayer()}
- The method \textit{PrsMgr\_PresentationManager::Unhighlight()} taking
  Display Mode as an argument has been marked deprecated.
  Implementation now performs unhighlighting of all highlighted
  presentation mode.
- The methods taking/returning \textit{Quantity\_NameOfColor} (predefined list
  of colors) and duplicating methods operating with \textit{Quantity\_Color}
  (definition of arbitrary RGB color) in AIS have been removed.
  \textit{Quantity\_Color} should be now used instead.
Elimination of implicit 3D Viewer updates

Most AIS_InteractiveContext methods are defined with a flag to update viewer immediately or not. Within previous version of OCCT, this argument had default value TRUE. While immediate viewer updates are useful for beginners (the result is displayed as soon as possible), this approach is inefficient for batch viewer updates, and having default value as TRUE leaded to non-intended accidental updates which are difficult to find.

To avoid such issues, the interface has been modified and default value has been removed. Therefore, old application code should be updated to set the flag theToUpdateViewer explicitly to desired value (TRUE to preserve old previous behavior), if it was not already set.

The following AIS_InteractiveContext methods have been changed: Display, Erase, EraseAll, DisplayAll, EraseSelected, DisplaySelected, ClearPrs, Remove, RemoveAll, Hilight, HilightWithColor, Unhilight, Redisplay, RecomputePrsOnly, Update, SetDisplayMode, UnsetDisplayMode, SetColor, UnsetColor, SetWidth, UnsetWidth, SetMaterial, UnsetMaterial, SetTransparency, UnsetTransparency, SetLocalAttributes, UnsetLocalAttributes, SetPolygonOffsets, SetTrihedronSize, SetPlaneSize, SetPlaneSize, SetDeviationCoefficient, SetDeviationAngle, SetAngleAndDeviation, SetHLRDeviationCoefficient, SetHLRDeviationAngle, SetHLRAngleAndDeviation, SetSelectedAspect, MoveTo, Select, ShiftSelect, SetSelected, UpdateSelected, AddOrRemoveSelected, HilightSelected, UnhilightSelected, ClearSelected, ResetOriginalState, SubIntensityOn, SubIntensityOff, FitSelected, EraseGlobal, ClearGlobal, ClearGlobalPrs.

In addition, the API for immediate viewer update has been removed from V3d_View and Graphic3d_StructureManager classes (enumerations Aspect_TypeOfUpdate and V3d_TypeOfUpdate): V3d::SetUpdateMode(), V3d::UpdateMode(), Graphic3d_StructureManager::SetUpdateMode(), Graphic3d_StructureManager::UpdateMode().

The argument theUpdateMode has been removed from methods Graphic3d_CView::Display(), Erase(), Update(). Method Graphic3d_CView::Update() does not redraw the view and does not recompute structures anymore.
The following Grid management methods within class V3d_Viewer do not implicitly redraw the viewer: ActivateGrid, DeactivateGrid, SetRectangularGridValues, SetCircularGridValues, RectangularGridGraphicValues, CircularGridGraphicValues, SetPrivilegedPlane, DisplayPrivilegedPlane.
Elimination of Quantity_NameOfColor from TKV3d interface classes

The duplicating interface methods accepting Quantity_NameOfColor (in addition to methods accepting Quantity_Color) of TKV3d toolkit have been removed. In most cases this change should be transparent, however applications implementing such interface methods should also remove them (compiler will automatically highlight this issue for methods marked with Standard_OVERRIDE keyword).
Result of Boolean operations on containers

- The result of Boolean operations on arguments of collection types (WIRE/SHELL/COMPSOLID) is now filtered from duplicating containers.
Other changes

- **MMgt_TShared** class definition has been removed - Standard_TTransient should be used instead (MMgt_TShared is marked as deprecated typedef of Standard_TTransient for smooth migration).
- Class GeomPlate_BuildPlateSurface accepts base class Adaptor3d_HCurve (instead of inherited Adaptor3d_HCurveOnSurface accepted earlier).
- Types GeomPlate_Array1OfHCurveOnSurface and GeomPlate_HArray1OfHCurveOnSurface have been replaced with GeomPlate_Array1OfHCurve and GeomPlate_HArray1OfHCurve correspondingly (accept base class Adaptor3d_HCurve instead of Adaptor3d_HCurveOnSurface).
- Enumeration **Image_PixMap::ImgFormat**, previously declared as nested enumeration within class **Image_PixMap**, has been moved to global namespace as Image_Format following OCCT coding rules. The enumeration values have suffix Image_Format_ and preserve previous name scheme for easy renaming of old values - e.g. Image_PixMap::ImgGray become Image_Format_Gray. Old definitions are preserved as deprecated aliases to the new ones; 
- Methods **Image_PixMap::PixelColor()** and **Image_PixMap::SetPixelColor()** now take/return Quantity_ColorRGBA instead of Quantity_Color/NCollection_Vec4. 
- The method BOPAlgo_Builder::Origins() returns BOPCol_DataMapOfShapeListOfShape instead of BOPCol_DataMapOfShapeShape.
- The methods BOPDS_DS::IsToSort(const Handle(BOPDS_CommonBlock)&, Standard_Integer&) and BOPDS_DS::SortPaveBlocks(const Handle(BOPDS_CommonBlock)&) have been removed. The sorting is now performed during the addition of the Pave Blocks into Common Block.
- The methods BOPAlgo_Tools::MakeBlocks() and BOPAlgo_Tools::MakeBlocksCnx() have been replaced with the single template method BOPAlgo_Tools::MakeBlocks(). The chains of connected elements are now stored into the list of list instead of data map.
- The methods BOPAlgo_Tools::FillMap() have been replaced with the
single template method BOPAlgo_Tools::FillMap().

- Package BVH now uses opencascade::handle instead of NCollection Handle (for classes BVH_Properties, BVH_Builder, BVH_Tree, BVH_Object). Application code using BVH package directly should be updated accordingly.

- AIS Shape now computes UV texture coordinates for AIS_Shaded presentation in case if texture mapping is enabled within Shaded Attributes. Therefore, redundant class AIS_TexturedShape is now deprecated - applications can use AIS_Shape directly (texture mapping should be defined through AIS_Shape::Attributes()).

- Methods for managing active texture within OpenGL_Workspace class (ActiveTexture(), DisableTexture(), EnableTexture()) have been moved to OpenGL_Context::BindTextures().
The classes `BOPDS_PassKey` and `BOPDS_PassKeyBoolean` are too excessive and not used any more in Boolean Operations. To replace them the new `BOPDS_Pair` class has been implemented. Thus:

- The method `BOPDS_DS::Interferences()` now returns the `BOPDS_MapOfPair`;
- The method `BOPDS_Iterator::Value()` takes now only two parameters - the indices of interfering sub-shapes.
Removal of the Draw commands based on old Boolean operations

- The commands \texttt{fubl} and \texttt{cubl} have been removed. The alternative for these commands are the commands \texttt{bfuseblend} and \texttt{bcutblend} respectively.
- The command \texttt{ksection} has been removed. The alternative for this command is the command \texttt{bsection}. 

Change of Face/Face intersection in Boolean operations

- Previously, the intersection tolerance for all section curves between pair of faces has been calculated as the maximal tolerance among all curves. Now, each curve has its own valid tolerance calculated as the maximal deviation of the 3D curve from its 2D curves or surfaces in case there are no 2D curves.
- The methods `IntTools_FaceFace::TolReached3d()`, `IntTools_FaceFace::TolReal()` and `IntTools_FaceFace::TolReached2d()` have been removed.
- Intersection tolerances of the curve can be obtained from the curve itself:
  - `IntTools_Curve::Tolerance()` - returns the valid tolerance for the curve;
  - `IntTools_Curve::TangentialTolerance()` - returns the tangential tolerance, which reflects the size of the common between faces.
- 2d tolerance (`IntTools_FaceFace::TolReached2d()`) has been completely removed from the algorithm as unused.
Restore OCCT 6.9.1 persistence

The capability of reading / writing files in old format using Storage_ShapeSchema functionality from OCCT 6.9.1 has been restored in OCCT 7.2.0.

One can use this functionality in two ways:

- invoke DRAW Test Harness commands fsdread / fsdwrite for shapes;
- call StdStorage class Read / Write functions in custom code.

The code example below demonstrates how to read shapes from a storage driver using StdStorage class.

```cpp
// aDriver should be created and opened for reading
Handle(StdStorage_Data) aData;

// Read data from the driver
// StdStorage::Read creates aData instance
// automatically if it is null
Storage_Error anError = StdStorage::Read(*aDriver, aData);
if (anError != Storage_VSOk) {
    // Error processing
}

// Get root objects
Handle(StdStorage_RootData) aRootData = aData->RootData();
Handle(StdStorage_HSequenceOfRoots) aRoots = aRootData->Roots();
if (!aRoots.IsNull()) {
    // Iterator over the sequence of root objects
    for (StdStorage_HSequenceOfRoots::Iterator
```
anIt(*aRoots); anIt.More(); anIt.Next()
{
  Handle(StdStorage_Root)& aRoot = anIt.ChangeValue();
  // Get a persistent root's object
  Handle(StdObjMgt_Persistent) aPObject = aRoot->Object();
  if (!aPObject.IsNull())
  {
    Handle(ShapePersistent_TopoDS::HShape) aHShape =
    Handle(ShapePersistent_TopoDS::HShape)::DownCast(aPObject);
    if (aHShape) // Downcast to an expected type to import transient data
      {
        TopoDS_Shape aShape = aHShape->Import();
        shapes.Append(aShape);
      }
  }
}

The following code demonstrates how to write shapes in OCCT 7.2.0 using StdStorage class.

// Create a file driver
NCollection_Handle<Storage_BaseDriver> aFileDriver(new FSD_File());

// Try to open the file driver for writing
try
{
  OCC_CATCH_SIGNALS
  PCDM_ReadWriter::Open (*aFileDriver,
    TCollection_ExtendedString(theFilename),
    Storage_VSWrite);
catch (Standard_Failure& e) {
    // Error processing
}

// Create a storage data instance
Handle(StdStorage_Data) aData = new StdStorage_Data();
// Set an axiliary application name (optional)
aData->HeaderData()->SetApplicationName(TCollection_ExtendedString("Application"));

// Provide a map to track sharing
StdObjMgt_TransientPersistentMap aMap;
// Iterator over a collection of shapes
for (Standard_Integer i = 1; i <= shapes.Length(); ++i) {
    TopoDS_Shape aShape = shapes.Value(i);
    // Translate a shape to a persistent object
    Handle(ShapePersistent_TopoDS::HShape) aPShape = ShapePersistent_TopoDS::Translate(aShape, aMap, ShapePersistent_WithTriangle);
    if (aPShape.IsNull()) {
        // Error processing
    }

    // Construct a root name
    TCollection_AsciiString aName = TCollection_AsciiString("Shape_") + i;

    // Add a root to storage data
    Handle(StdStorage_Root) aRoot = new StdStorage_Root(aName, aPShape);
aData->RootData()->AddRoot(aRoot);
}

// Write storage data to the driver
Storage_Error anError =
    StdStorage::Write(*aFileDriver, aData);
if (anError != Storage_VSOk)
{
    // Error processing
}
Change in BRepLib_MakeFace algorithm

Previously, BRepLib_MakeFace algorithm changed orientation of the source wire in order to avoid creation of face as a hole (i.e. it is impossible to create the entire face as a hole; the hole can be created in context of another face only). New algorithm does not reverse the wire if it is open. Material of the face for the open wire will be located on the left side from the source wire.
Change in BRepFill_OffsetWire algorithm

From now on, the offset will always be directed to the outer region in case of positive offset value and to the inner region in case of negative offset value. Inner/Outer region for an open wire is defined by the following rule: when we go along the wire (taking into account edges orientation) the outer region will be on the right side, the inner region will be on the left side. In case of a closed wire, the inner region will always be inside the wire (at that, the edges orientation is not taken into account).
Change in Geom(2d)Adaptor_Curve::IsPeriodic

Since 7.2.0 version, method IsPeriodic() returns the corresponding status of periodicity of the basis curve regardless of closure status of the adaptor curve (see method IsClosed()). Method IsClosed() for adaptor can return false even on periodic curve, in the case if its parametric range is not full period, e.g. for adaptor on circle in range \([0, \pi]\). In previous versions, IsPeriodic() always returned false if IsClosed() returned false.
Change in algorithm
ShapeUpgrade_UnifySameDomain

The history of the changing of the initial shape was corrected:

- all shapes created by the algorithm are considered as modified shapes instead of generated ones;
- method Generated was removed and its calls should be replaced by calls of method History()\rightarrow Modified.
Changes in STL Reader / Writer

Class RWStl now uses class Poly_Triangulation for storing triangular mesh instead of StlMesh data classes; the latter have been removed.
Refactoring of the Error/Warning reporting system in Boolean Component

The Error/Warning reporting system of the algorithms in Boolean Component (in all BOPAlgo_* and BRepAlgoAPI_* algorithms) has been refactored. The methods returning the status of errors and warnings of the algorithms (ErrorStatus() and WarningStatus()) have been removed. Instead use methods HasErrors() and HasWarnings() to check for presence of errors and warnings, respectively. The full list of errors and warnings, with associated data such as problematic sub-shapes, can be obtained by method GetReport().
<table>
<thead>
<tr>
<th>License</th>
</tr>
</thead>
</table>

Open CASCADE Technology is available under GNU Lesser General Public License (LGPL) version 2.1 with additional exception.
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts
as the successor of the GNU Library Public License, version 2, hence
the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom
to share and change it. By contrast, the GNU General Public Licenses
are intended to guarantee your freedom to share and change free
software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some
specially designated software packages—typically libraries—of the Free
Software Foundation and other authors who decide to use it. You can use
it too, but we suggest you first think carefully about whether this license
or the ordinary General Public License is the better strategy to use in any
particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not
price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free
When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

- 0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any
warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

1. The modified work must itself be a software library.
2. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
3. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
4. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.
Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked
with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License. Also, you must do one of these things:

1. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

2. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

3. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

4. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

5. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:
   1. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
   2. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.
11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time.
Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!
Open CASCADE Exception (version 1.0) to GNU LGPL version 2.1.

The object code (i.e. not a source) form of a "work that uses the Library" can incorporate material from a header file that is part of the Library. As a special exception to the GNU Lesser General Public License version 2.1, you may distribute such object code incorporating material from header files provided with the Open CASCADE Technology libraries (including code of CDL generic classes) under terms of your choice, provided that you give prominent notice in supporting documentation to this code that it makes use of or is based on facilities provided by the Open CASCADE Technology software.