Tutorial

My First Application

NOTICE FOR USERS:

This User Guide is a general instruction for Open CASCADE Technology study. It may be incomplete and even contain occasional mistakes, particularly in examples, samples, etc.

OPEN CASCADE S.A.S. bears no responsibility for such mistakes. If you find any mistakes or imperfections in this document, or if you have suggestions for improving this document, please, contact us and contribute your share to the development of Open CASCADE Technology: bugmaster@opencascade.com

http://www.opencascade.com/contact/

Table of Contents

1. 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.

1.1. 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.

1.2. The Model

To illustrate the use of classes provided in the 3D geometric modeling toolkits, you will create a bottle as shown:
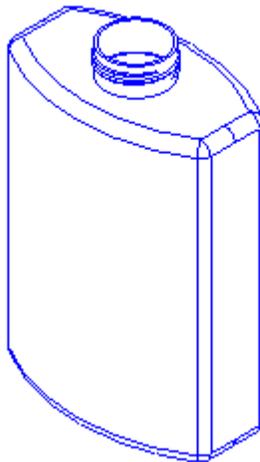
In the tutorial we will create, step-by-step, a function that will model a bottle as shown above. You will find the complete source code of this tutorial, including the very function MakeBottle in the distribution of Open CASCADE Technology. The function body is provided in the file samples/qt/Tutorial/src/MakeBottle.cxx.

1.3. Model Specifications

We first define the bottle specifications as follows:

Object Parameter	Parameter Name	Parameter Value
Bottle height	MyHeight	70mm
Bottle width	MyWidth	50mm
Bottle thickness	MyThickness	30mm

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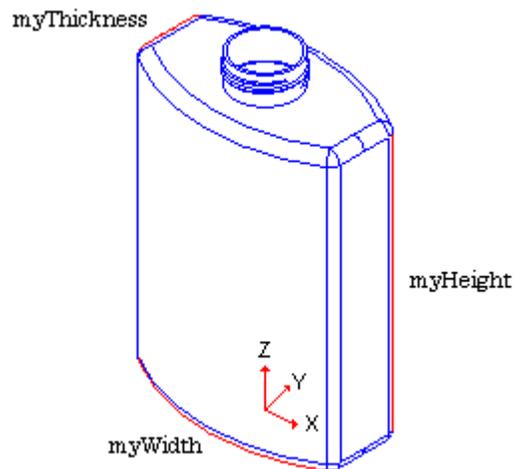
This modeling requires four steps:

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

2. Building the Profile

2.1. 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.
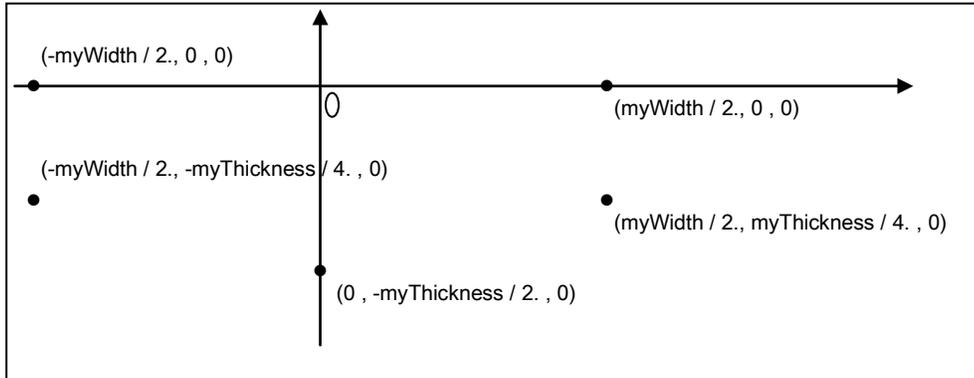
There are two classes to describe a 3D Cartesian point from its X, Y and Z coordinates in Open CASCADE Technology:

- 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:

```
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);
```

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();
```

2.2. *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.
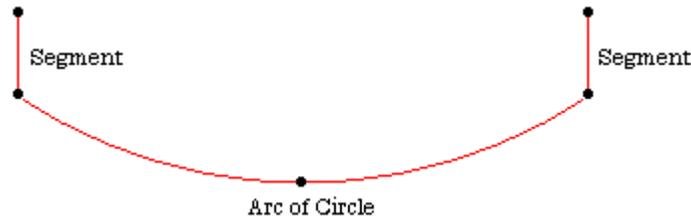
To create such entities, you need a specific data structure, which implements 3D geometric objects. This can be found in the *Geom* package of Open CASCADE Technology.

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 G*eom_TrimmedCurve* on C limited between 0 and M_PI/2.

```
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:

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

2.3. 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:

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

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.
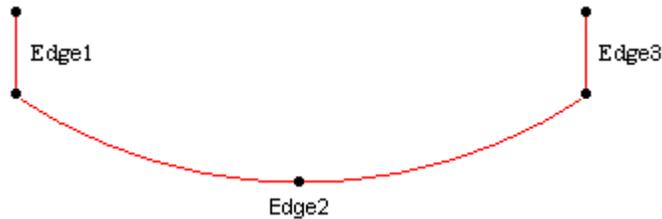
However, the *TopoDS* package provides only the data structure of the topological entities. Algorithm classes available to compute standard topological objects can be found in the *BRepBuilderAPI* package.

To create an edge, you use the *BRepBuilderAPI_MakeEdge* class with the previously computed curves:

```
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:

```
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);
```

2.4. 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.
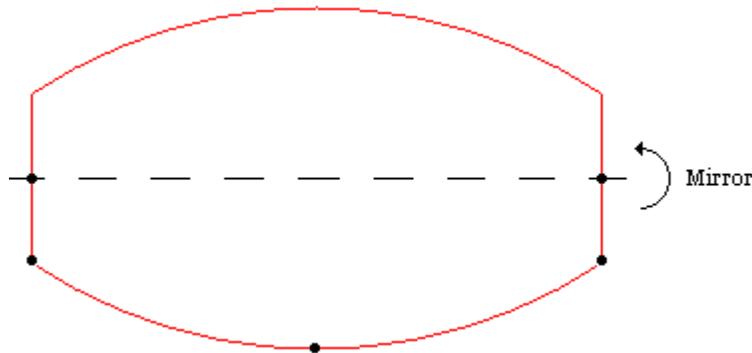
To apply a transformation on shapes (including wires), you first need to define the properties of a 3D geometric transformation by using the *gp_Trsf* class. This transformation can be a translation, a rotation, a scale, a reflection, or a combination of these.

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.

```
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:

```
gp_Ax1 xAxis = gp::OX();
```

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.

```
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

```
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:

```
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.

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

The bottle's profile is almost finished. You have created two wires: *aWire* and *aMirroredWire*. You need to concatenate them to compute a single shape. To do this, you use the *BRepBuilderAPI_MakeWir*e class as follows:

- create an instance of *BRepBuilderAPI_MakeWire*.

- add all edges of the two wires by using the *Add* method on this object.

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

3. Building the Body

3.1. 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:

Shape	Generates
Vertex	Edge
Edge	Face
Wire	Shell
Face	Solid
Shell	Compound of Solids

myHeight
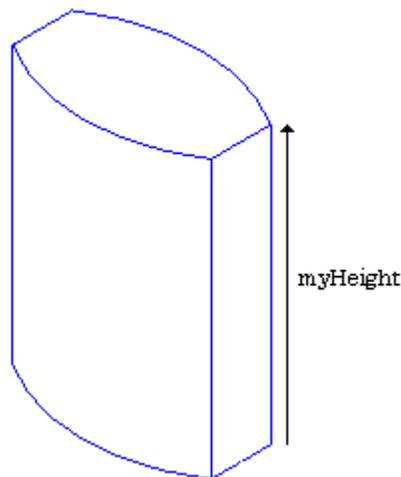
Your current profile is a wire. Referring to the Shape/Generates table, you need to compute a face out of its wire to generate a solid.

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);
```

3.2. *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 *myThickness* / 12
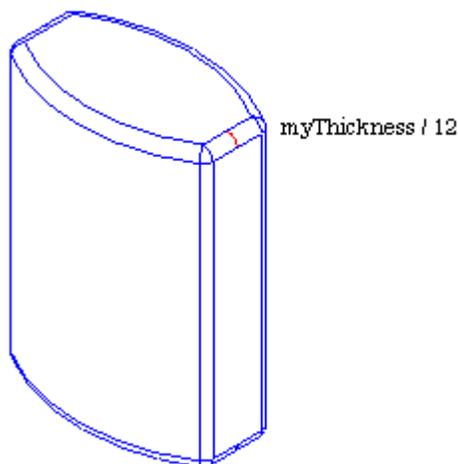
To apply fillets on the edges of a shape, you use the *BRepFilletAPI_MakeFillet* class. This class is normally used as follows:

- 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.

```
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.

```
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.

```
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.

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

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

```
myBody = mkFillet.Shape();
```

3.3. 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.
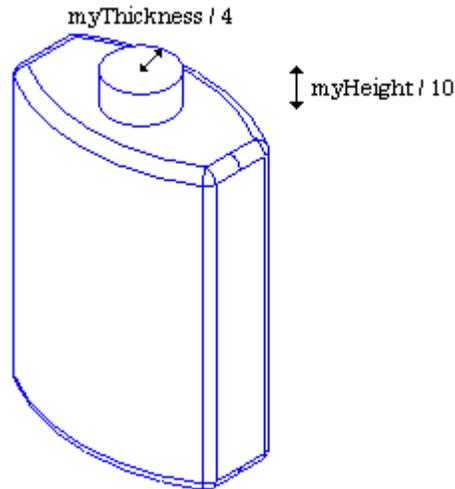
To position the cylinder, you need to define a coordinate system with the *gp_Ax2* class defining a right-handed coordinate system from a point and two directions - the main (Z) axis direction and the X direction (the Y direction is computed from these two).

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:

```
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

```
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 *BRepAlgoAPI* package provides services to perform Boolean operations between shapes, and especially: common (Boolean intersection), cut (Boolean subtraction) and fuse (Boolean union).

Use *BRepAlgoAPI_Fuse* to fuse the two shapes:

```
myBody = BRepAlgoAPI_Fuse(myBody, myNeck);
```

3.4. 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.
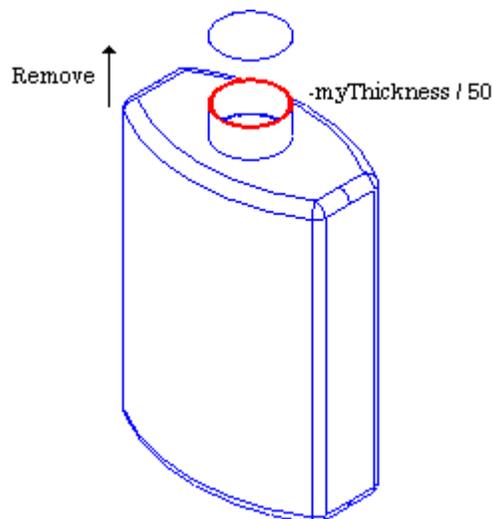
To compute a thick solid, you create an instance of the *BRepOffsetAPI_MakeThickSolid* class by giving the following information:

- 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.

```
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

```
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(aSurface->DynamicType()   ==   STANDARD_TYPE(Geom_Plane)){
...
}
```

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:

```
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:

```
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:

```
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.

```
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* constructor:

```
MyBody = BRepOffsetAPI_MakeThickSolid(myBody, facesToRemove, -
myThickness / 50, 1.e-3);
```

4. Building the Threading

4.1. 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:
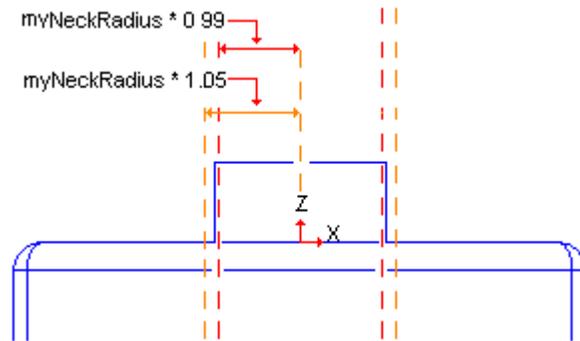
Notice that one of the cylindrical surfaces is smaller than the neck. There is a good reason for this: after the thread creation, you will fuse it with the neck. So, we must make sure that the two shapes remain in contact.

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

Handle(Geom_CylindricalSurface) aCyl2 = new
Geom_CylindricalSurface(neckAx2, myNeckRadius * 1.05);
```

4.2. 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 * (cos(U) * xDir + sin(U) * yDir) + V * 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, 2PI] and V is infinite.
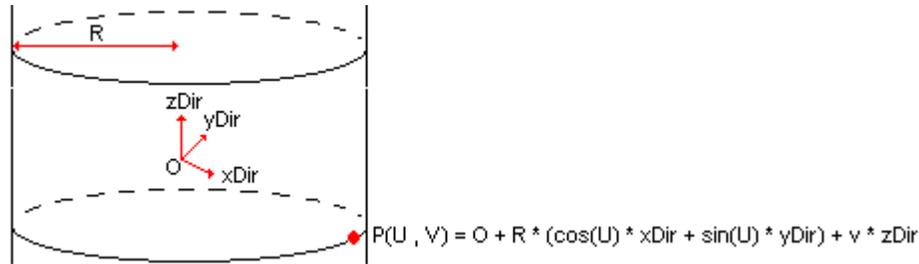
The advantage of having such parameterized geometries is that you can compute, for any (U, V) parameters of the surface:

- 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:
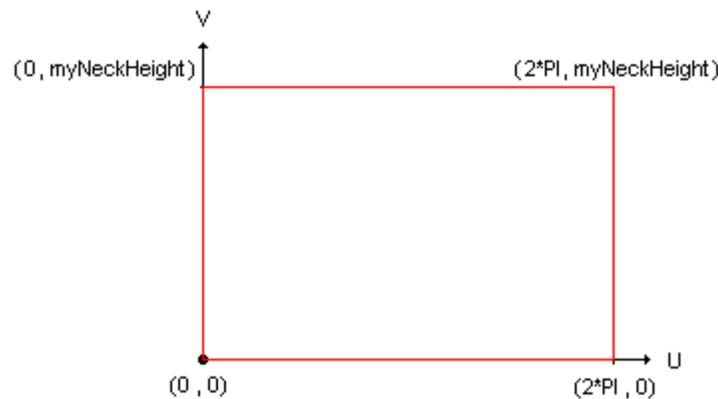
Suppose that you create a 2D line on this parametric (U, V) space and compute its 3D parametric curve. Depending on the line definition, results are as follows:

Case	Parametric Equation	Parametric Curve
U = 0	P(V) = O + V * zDir	Line parallel to the Z direction
V = 0	P(U) = O + R * (cos(U) * xDir + sin(U) * yDir)	Circle parallel to the (O, X, Y) plane
U != 0 V != 0	P(U, V) = O + R * (cos(U) * xDir + sin(U) * yDir) + V * zDir	Helicoidal curve describing the evolution of height and angle on the cylinder

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:
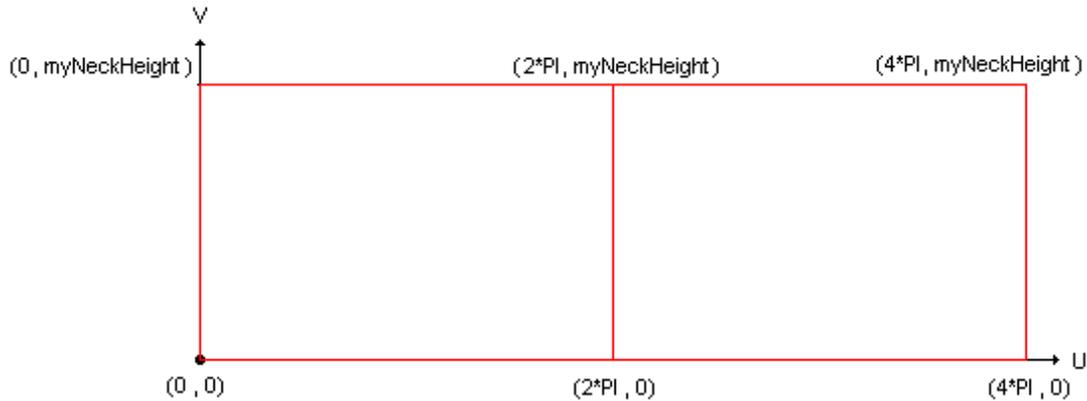
In this (U, V) parametric space, you will create a local (X, Y) coordinate system to position the curves to be created. This coordinate system will be defined with:

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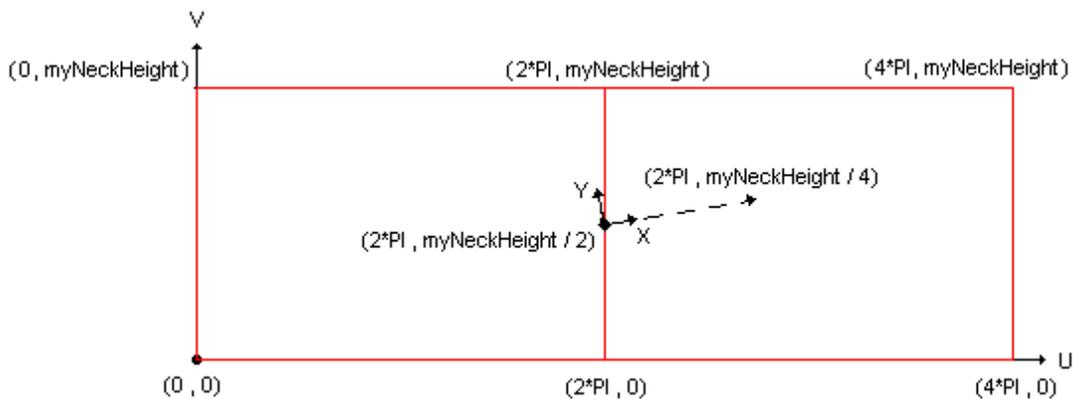
To use 2D primitive geometry types of Open CASCADE Technology for defining a point and a coordinate system, you will once again instantiate classes from *gp*:

- 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.

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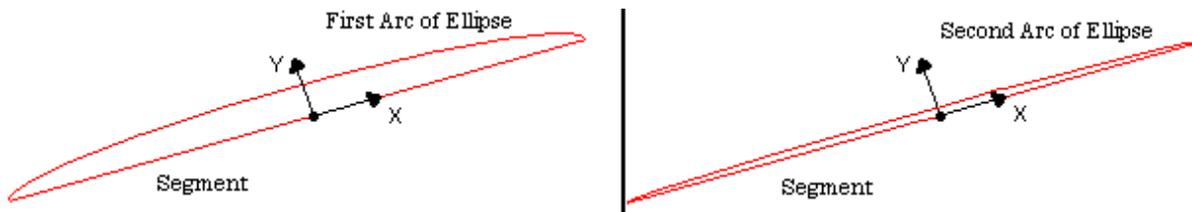
You have already used the *Geom* package to define 3D geometric entities. For 2D, you will use the *Geom2d* package. As for *Geom*, all geometries are parameterized. For example, a *Geom2d_Ellipse* ellipse is defined from:

- 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:

```
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 + (MajorRadius * cos(U) * XDirection) + (MinorRadius * sin(U) * 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);
```

4.3. 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.
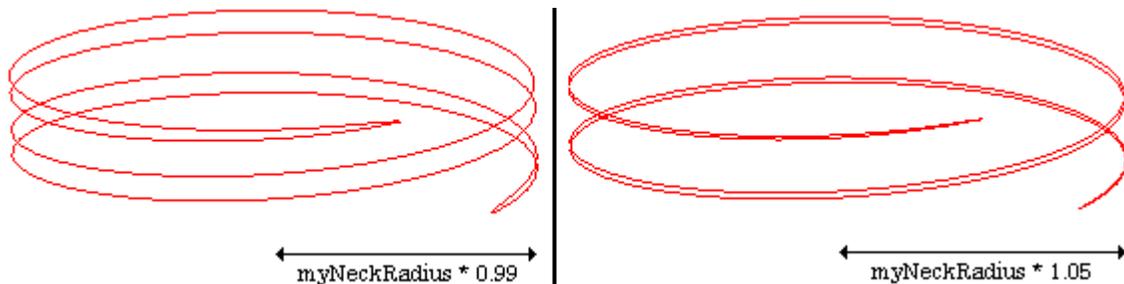
Previously, you have built:

- 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.

```
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.

```
TopoDS_Wire  threadingWire1  =  BRepBuilderAPI_MakeWire(anEdge1OnSurf1,
anEdge2OnSurf1);
TopoDS_Wire  threadingWire2  =  BRepBuilderAPI_MakeWire(anEdge1OnSurf2,
anEdge2OnSurf2);
```

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.

```
BRepLib::BuildCurves3d(threadingWire1);
BRepLib::BuildCurves3d(threadingWire2);
```

4.4. 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:
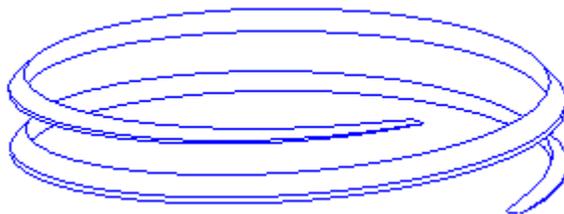
- *Initialize* the algorithm by creating an instance of the class. The first parameter of this constructor must be specified if you want to create a solid. By default, *BRepOffsetAPI_ThruSections* builds a shell.

- 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.

```
BRepOffsetAPI_ThruSections aTool(Standard_True);
aTool.AddWire(threadingWire1);
aTool.AddWire(threadingWire2);
aTool.CheckCompatibility(Standard_False);
TopoDS_Shape myThreading = aTool.Shape();
```

5. 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*:

```
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:
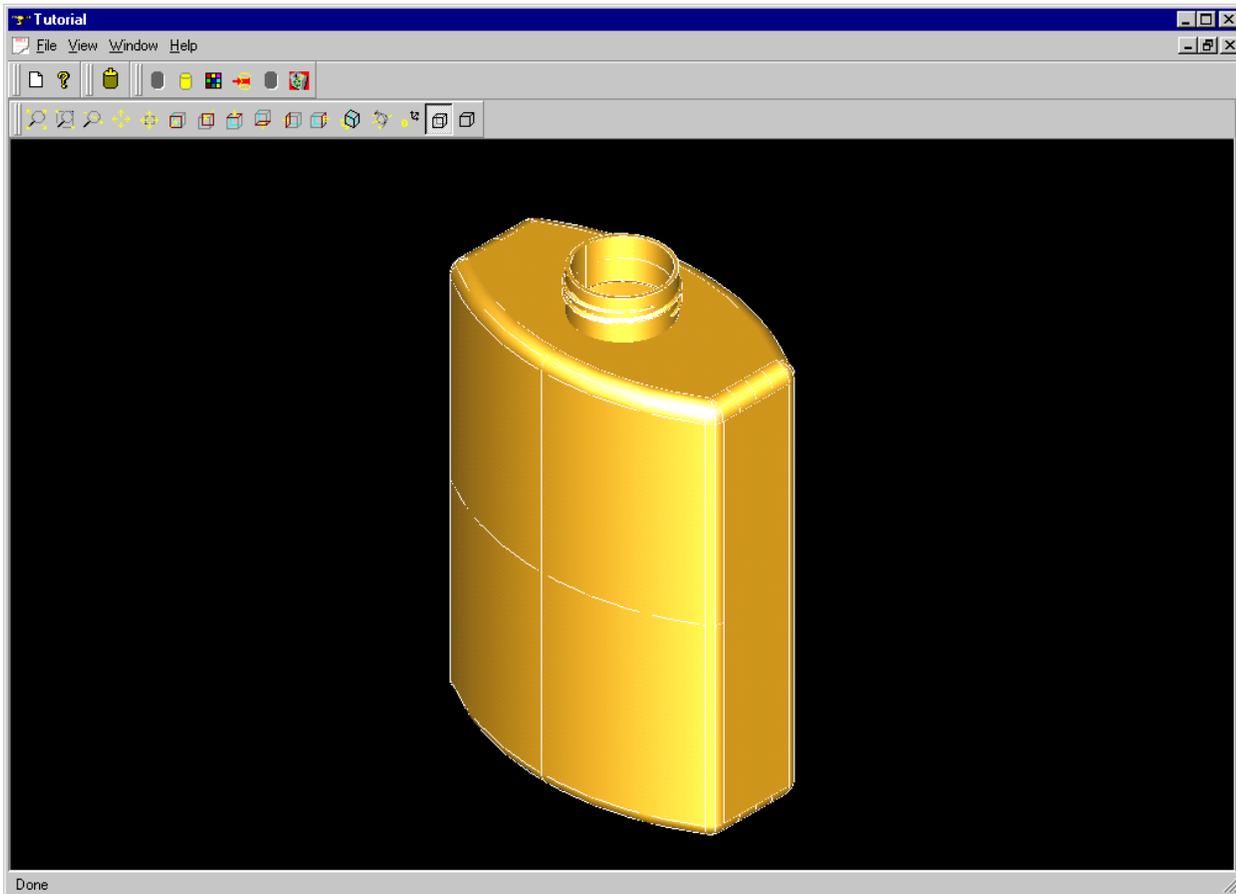
We hope that this tutorial has provided you with a feel for the industrial strength power of Open CASCADE Technology.

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.org/support. Our professional services can maximize the power of your Open CASCADE Technology applications.

6. Appendix

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

```
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);

myBody = BRepOffsetAPI_MakeThickSolid(myBody, facesToRemove, -myThickness / 50,
1.e-3);

// 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, 0,
M_PI);

Handle(Geom2d_TrimmedCurve) anArc2 = new Geom2d_TrimmedCurve(anEllipse2, 0,
```

```
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;
}
```