



Open CASCADE Technology  
6.8.0

Modeling Data

November 7, 2014

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                    | <b>1</b>  |
| <b>2</b> | <b>Geometry Utilities</b>                              | <b>2</b>  |
| 2.1      | Interpolations and Approximations                      | 2         |
| 2.1.1    | Analysis of a set of points                            | 2         |
| 2.1.2    | Basic Interpolation and Approximation                  | 2         |
| 2.1.3    | Advanced Approximation                                 | 3         |
| 2.2      | Direct Construction                                    | 5         |
| 2.2.1    | Non-persistent entities                                | 5         |
| 2.2.2    | Persistent entities                                    | 7         |
| 2.3      | Conversion to and from BSplines                        | 8         |
| 2.4      | Points on Curves                                       | 8         |
| 2.5      | Extrema  | 9         |
| 2.5.1    | Extrema between Curves                                 | 9         |
| 2.5.2    | Extrema between Curve and Surface                      | 9         |
| 2.5.3    | Extrema between Surfaces                               | 10        |
| <b>3</b> | <b>2D Geometry</b>                                     | <b>11</b> |
| <b>4</b> | <b>3D Geometry</b>                                     | <b>12</b> |
| 4.1      | Local Properties of Curves and Surfaces                | 12        |
| <b>5</b> | <b>Topology</b>  | <b>14</b> |
| 5.1      | Shape Location   | 14        |
| 5.2      | Naming shapes, sub-shapes, their orientation and state | 15        |
| 5.2.1    | Topological types                                      | 16        |
| 5.2.2    | Orientation  | 17        |
| 5.2.3    | State  | 19        |
| 5.3      | Manipulating shapes and sub-shapes                     | 20        |
| 5.4      | Exploration of Topological Data Structures             | 24        |
| 5.5      | Lists and Maps of Shapes                               | 26        |
| 5.5.1    | Wire Explorer  | 28        |

## 1 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 offerings on our web site at [www.opencascade.org/support/training/](http://www.opencascade.org/support/training/)

## 2 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.

### 2.1 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. This component provides both high and low level services to approximate or interpolate points into curves and surfaces. The lower level services allow performing parallel approximation of groups of points into groups of Bezier or B-spline curves.

#### 2.1.1 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.

#### 2.1.2 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.

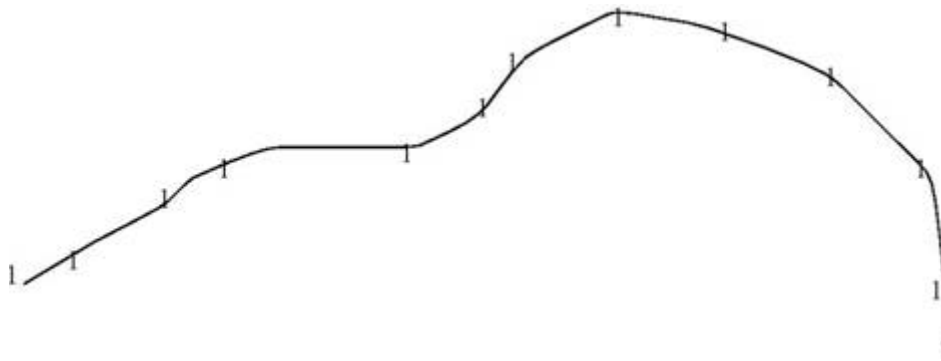


Figure 1: Approximation of a BSpline from scattered points

This class may be instantiated as follows:

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

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

```
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 BSplinecurve, 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.

#### 2.1.3 Advanced Approximation

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

### 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 multipoint constraints in order to build the multi-line, multiple lines passing through ordered multiple point constraints.

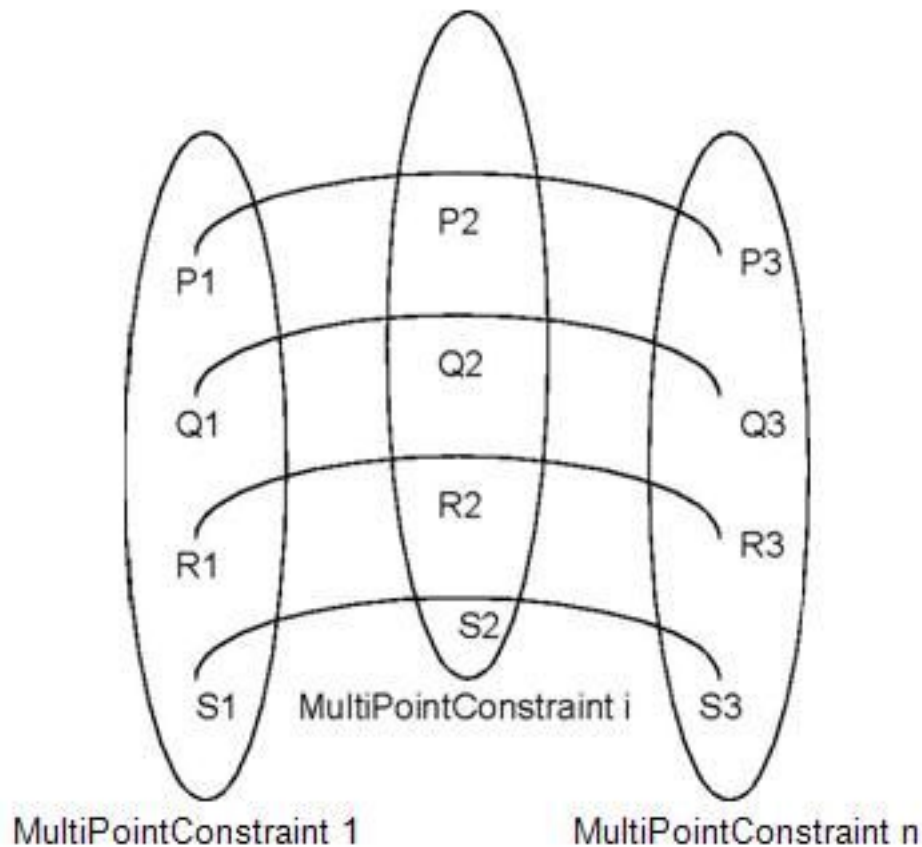


Figure 2: Definition of a MultiLine using Multiple Point Constraints

In this image:

- $P_i, Q_i, R_i \dots S_i$  can be 2D or 3D points.
- Defined as a group:  $P_n, Q_n, R_n, \dots S_n$  form a MultipointConstraint. They possess the same passage, tangency and curvature constraints.
- $P_1, P_2, \dots P_n$ , or the  $Q, R, \dots$  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.

## 2.2 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.

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.

### 2.2.1 Non-persistent entities

The following algorithms used to build entities from non-persistent *gp* entities 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.

```
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:

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

### 2.2.2 Persistent entities

*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 but create **persistent** entities, 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*

## 2.3 Conversion to and from BSplines

The following algorithms to convert geometric curves or surfaces into their BSpline or Bezier equivalents are provided by *GeomConvert*, *Geom2dConvert* and *Convert* packages:

- Conversion of a conic into a rational BSpline.
- Conversion of an elementary surface into a rational BSpline.
- Conversion of a BSpline or Bezier curve into two or more Bezier curves or surfaces.
- Conversion of a BSpline curve or surface into two or more BSplinecurves or surfaces with constraints on continuity.
- Conversion of a set of joining Bezier curves into a BSplinecurve.
- Conversion of a polynomial representation into a BSpline curve.

## 2.4 Points on Curves

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:

- *AbscissaPoint* calculates a point on a curve at a given distance from another point on the curve.
- *UniformAbscissa* calculates a set of points at a given abscissa on a curve.
- *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 :**

```
Handle(Geom2d_Curve) mycurve = ... ;
Geom2dAdaptor_Curve C (mycurve) ;
```

**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) ;
        ...
    }
}
```

**2.5 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.

**2.5.1 Extrema between Curves**

The *Geom2dAPI\_ExtremaCurveCurve* class allows calculation of all extrema between two 2D geometric curves. Extrema are the lengths of the segments orthogonal to two curves.

The *GeomAPI\_ExtremaCurveCurve* class allows calculation of all extrema between two 3D geometric curves. Extrema are the lengths of the segments orthogonal to two curves.

**2.5.2 Extrema between Curve and Surface**

The *GeomAPI\_ExtremaCurveSurface* class allows calculation of all extrema between a 3D curve and a surface. Extrema are the lengths of the segments orthogonal to the curve and the surface.

### 2.5.3 Extrema between Surfaces

The *GeomAPI\_ExtremaSurfaceSurface* class allows calculation of all extrema between two surfaces. Extrema are the lengths of the segments orthogonal to two surfaces.

### 3 2D Geometry

*Geom2d* package defines geometric objects in 2dspace. All geometric entities are STEP processed. The objects are non-persistent and are handled by reference. 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.

## 4 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. The following non-persistent and reference-handled 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.

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.

### 4.1 Local Properties of Curves and Surfaces

Packages **GeomLProp**, **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:

- *CLProps* - calculates the local properties of a curve (tangency, curvature,normal);
- *CurAndInf2d* - calculates the maximum and minimum curvatures and the inflection points of 2d curves;
- *SLProps* - calculates the local properties of a surface (tangency, the normal and curvature).
- *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.

## 5 Topology

Open CASCADE Technology Topology allows accessing and manipulating objects data 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.

To provide its descriptions, OCCT abstract topology offers the following services:

- Keeping track of Location of shapes
- Naming shapes, sub-shapes, their orientations and states
- Manipulating shapes and sub-shapes
- Exploring topological data structures
- Using lists and maps of shapes

### 5.1 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.

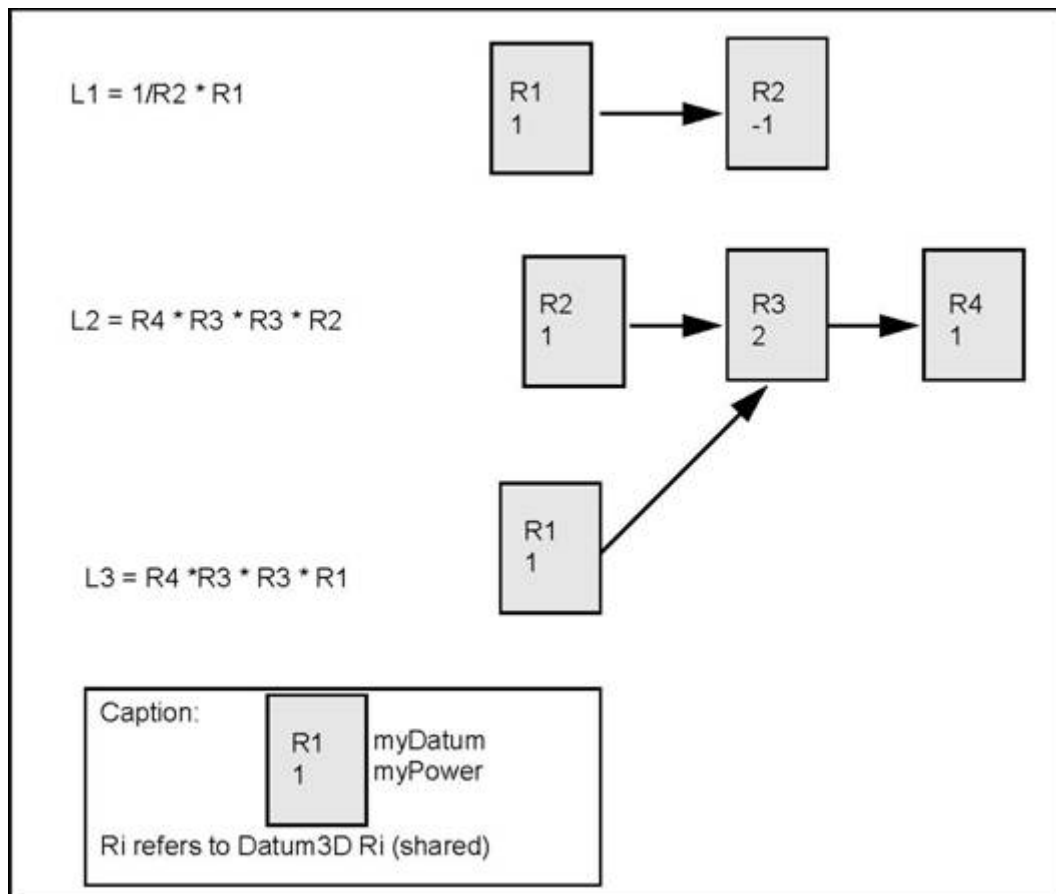


Figure 3: Structure of TopLoc\_Location

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 `MMgt_TShared`. The coordinate is represented by a transformation `gp_Trsfpackage`. This transformation has no scaling factor.

## 5.2 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:

- Topological type (`TopAbs_ShapeEnum`)

- Orientation (TopAbs\_Orientation)
- StateTopAbs\_State)

### 5.2.1 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 TopAbspackage 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.

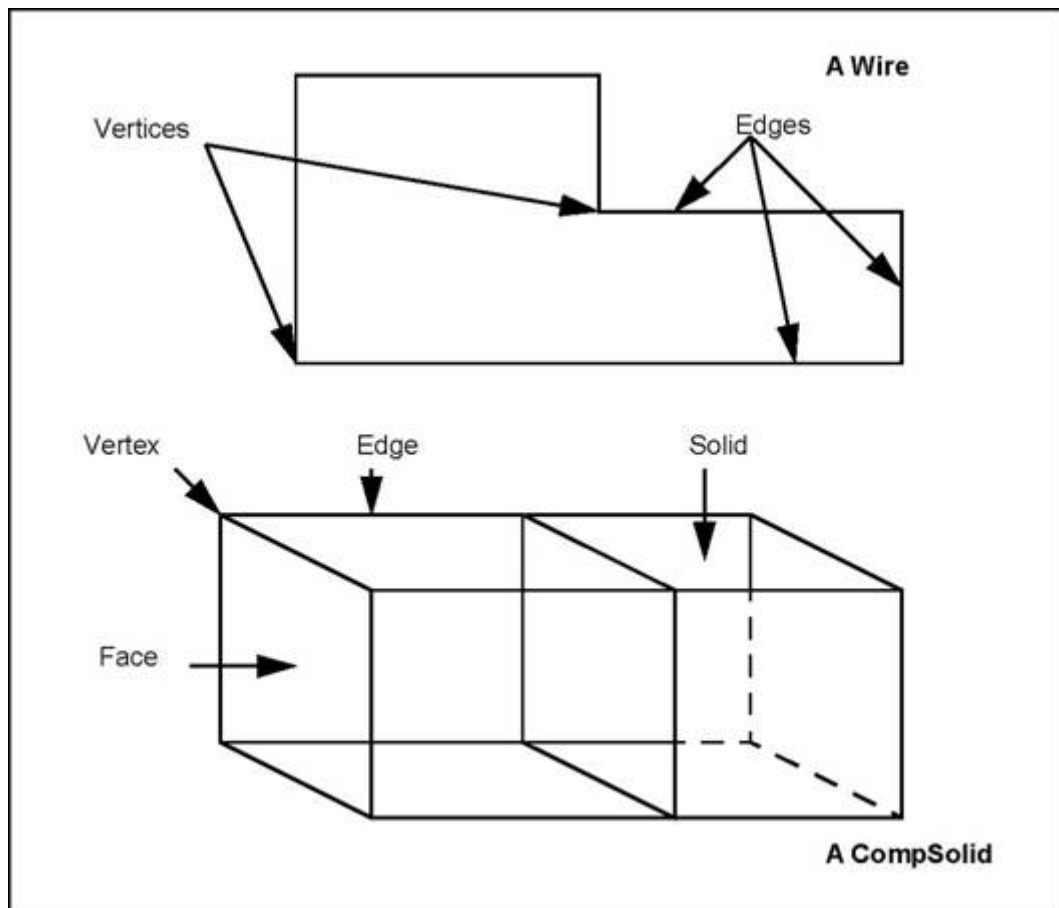


Figure 4: ShapeEnum

### 5.2.2 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.

| Orientation | Description  |
|-------------|--|
| FORWARD     | The interior is the default region.  |
| REVERSED    | The interior is the region complementary to the default.   |
| INTERNAL    | The interior includes both regions. The boundary lies inside the material. For example a surface inside a solid.         |
| EXTERNAL    | The interior includes neither region. The boundary lies outside the material. For example an edge in a wire-frame model. |

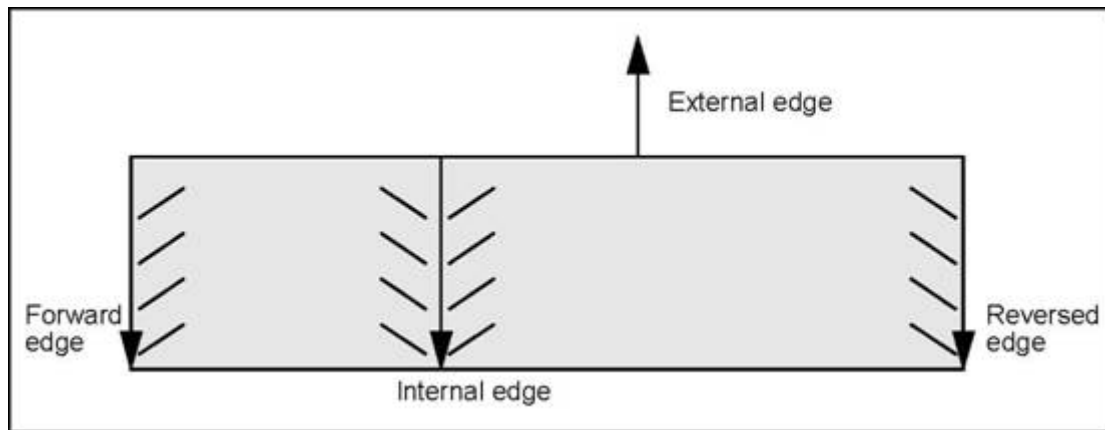


Figure 5: Four Orientations

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 - with the intersection vertex as the boundary. Thus an orientation can be associated with an intersection vertex as in the following figure:

| Orientation | Association           |
|-------------|-----------------------|
| FORWARD     | Entering              |
| REVERSED    | Exiting               |
| INTERNAL    | Touching from inside  |
| EXTERNAL    | Touching from outside |

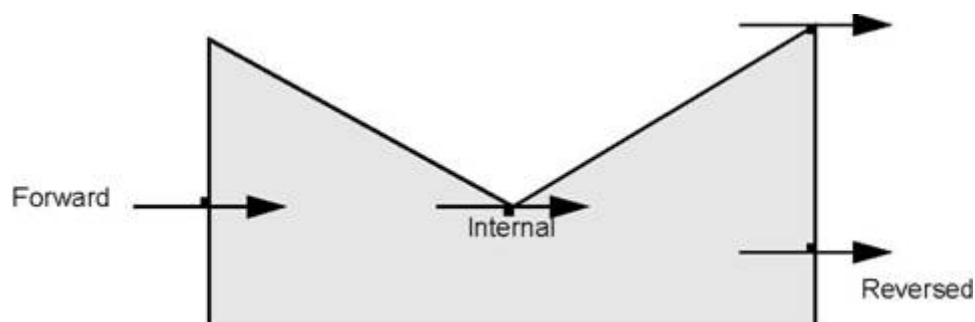


Figure 6: Four orientations of intersection vertices

Along with the Orientation enumeration the *TopAbs* package defines four methods:

### 5.2.3 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:

| Position | Description                                     |
|----------|---|
| IN       | The point is interior.                          |
| OUT      | The point is exterior.                          |
| ON       | The point is on the boundary(within tolerance). |
| UNKNOWN  | The state of the point is indeterminate.        |

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.

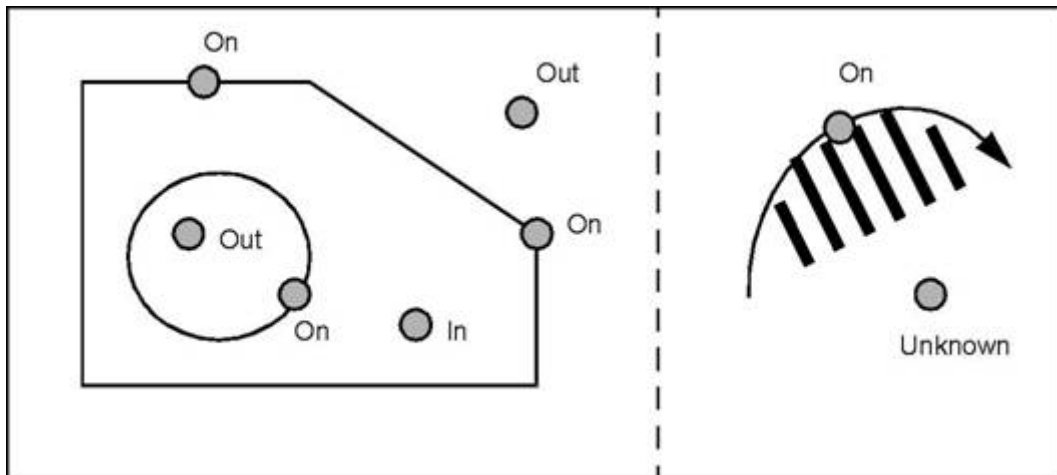


Figure 7: The four states

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.

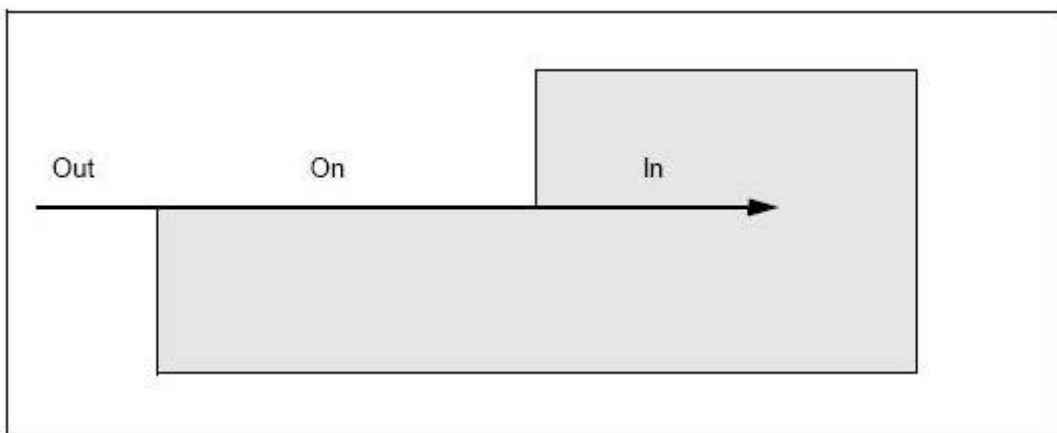


Figure 8: State specifies the parts of an edge intersecting a face

### 5.3 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.

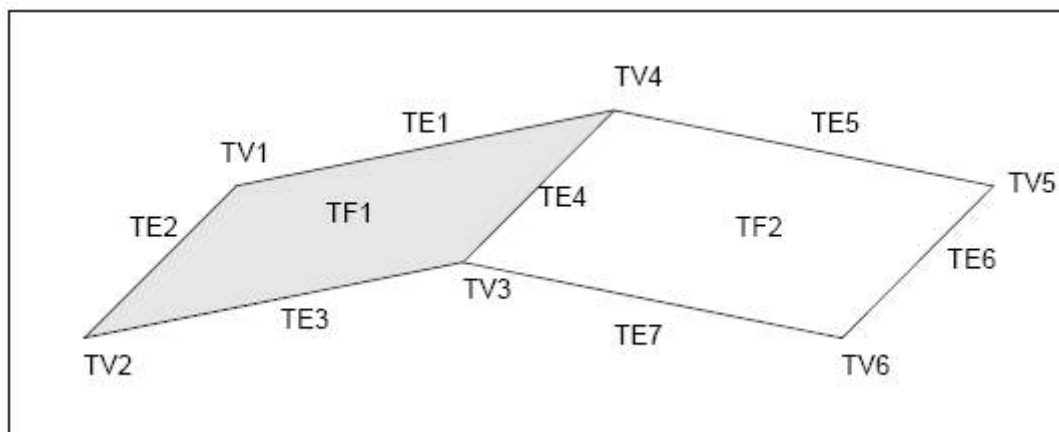


Figure 9: Structure of a shell formed from two faces

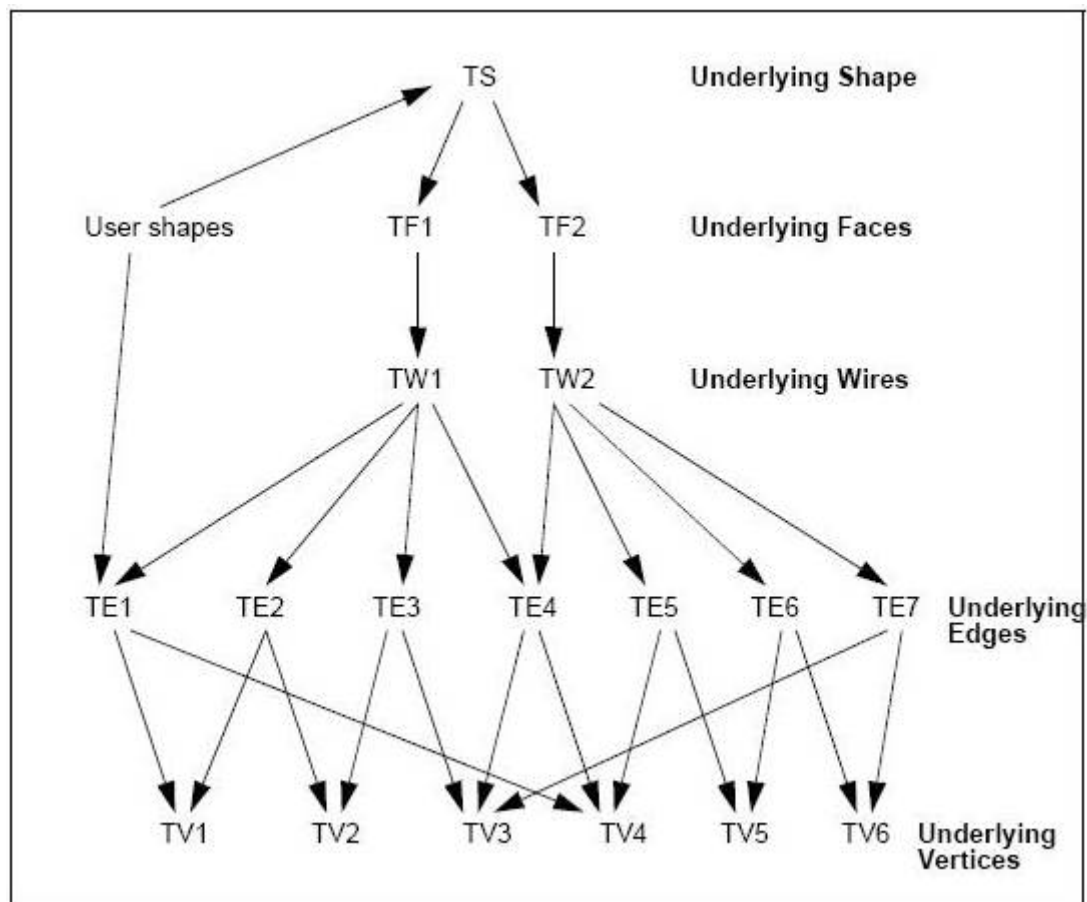


Figure 10: Data structure of the above shell

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(TV5,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.

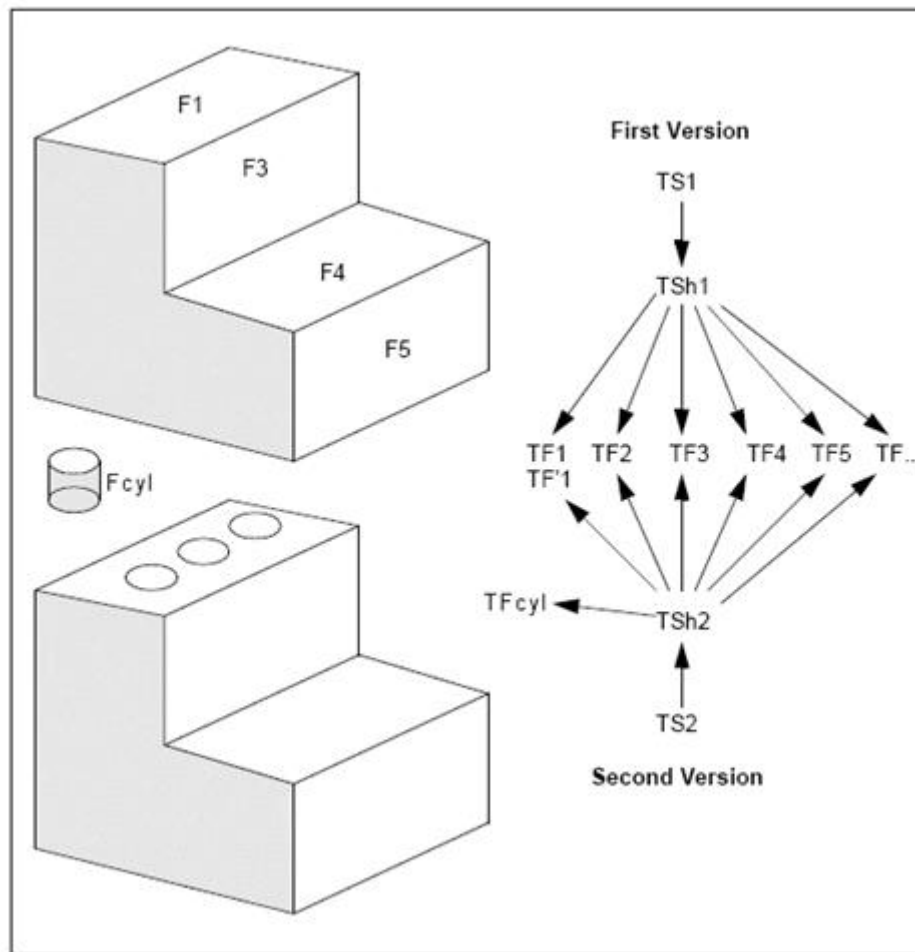


Figure 11: Data structure containing two versions of a solid

### 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.

```
#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
    }
}
```

## 5.4 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*:

```
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

```
for (Ex.Init (S, TopAbs_VERTEX, TopAbs_EDGE); ...)
```

Find all the faces in a SHELL, then all the faces not in a SHELL:

```
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

```
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 *FaceIsoColor* 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 *DrawFaceIso* 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.

```
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;
        }
    }
}

```

## 5.5 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*.

```

#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:

```

#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.

```

#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.

```

//Create an array to store the copies.
TopTools_Array1OfShape theCopies(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.

```

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);
        }
    }
}

```

### 5.5.1 Wire Explorer

BRepTools\_WireExplorer class can access edges of a wire in their order of connection.

For example, in the wire in the image we want to recuperate the edges in the order {e1, e2, e3,e4, e5} :

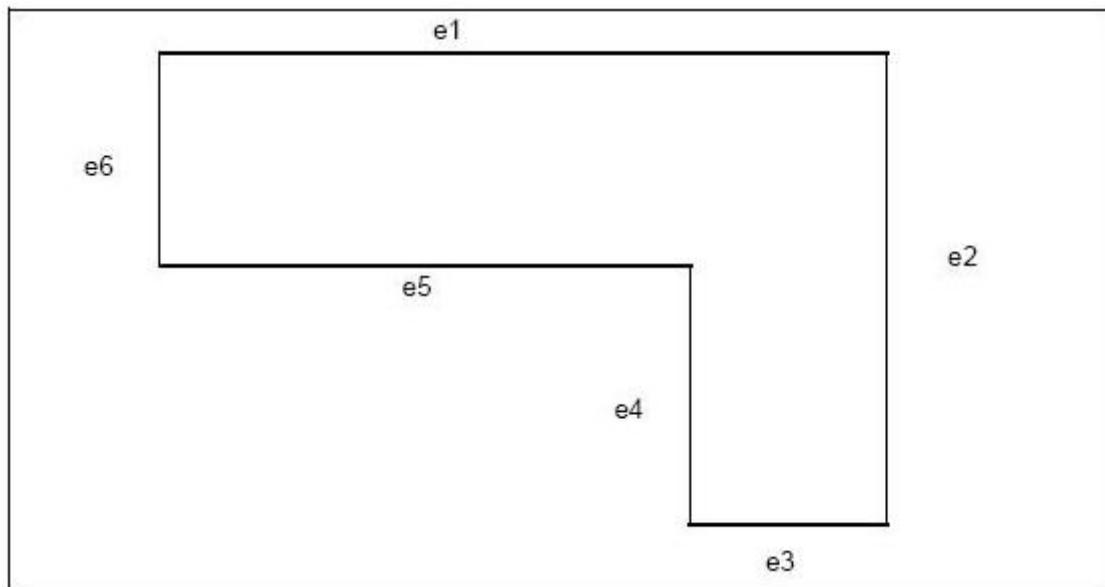


Figure 12: A wire composed of 6 edges.

*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());
    ProcessTheVertexConnectingTheCurrentEdgeToThePrevious
    One (Ex.CurrentVertex());
}
```