Open CASCADE Technology
7.6.0

Modeling Data

October 30, 2021

Contents

1 Introduction

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

This manual explains how to use Modeling Data.

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.

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

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

You can program approximations in two ways:

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

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.

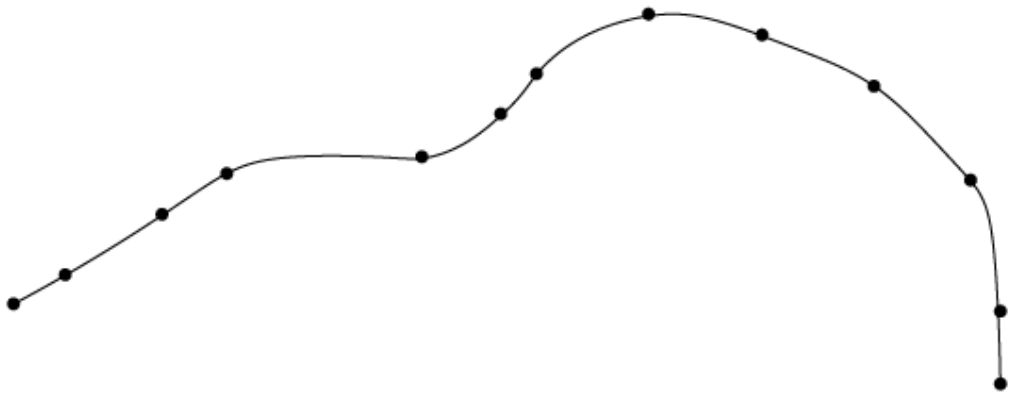

Figure 1: Approximation of a BSpline from scattered points

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 beC2 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 C1continuous.

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.

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

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

You can also find functions to compute:

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

Approximation by multiple point constraints

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

The following low level services are provided:

- Definition of an array of point constraints:

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

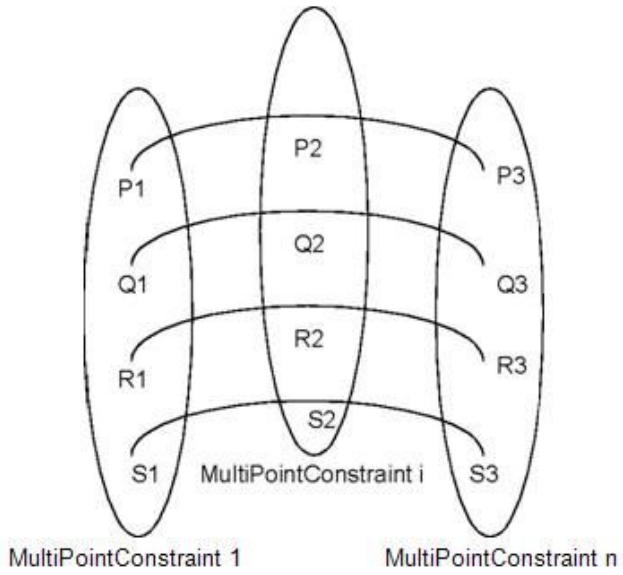

Figure 2: Definition of a MultiLine using Multiple Point Constraints

In this image:

- *Pi*, *Qi*, *Ri* ... *Si* can be 2D or 3D points.
- Defined as a group: *Pn*, *Qn*, *Rn, ... Sn* form a MultipointConstraint. They possess the same passage, tangency and curvature constraints.
- *P1*, *P2*, ... *Pn*, or the *Q*, *R*, ... or *S* series represent the lines to be approximated.

- Definition of a set of point constraints:

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

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

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

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

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

- Definition of Variational Criteria:

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

Approximation by parametric or geometric constraints

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

The algorithms used include:

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

The following low-level services are provided:

- Association of an index to an object:

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

- Definition of a set of approximations of Bezier curves:

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

- Definition of a set of approximations of BSpline curves:

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

- Definition of points making up a set of point constraints

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

Example: How to approximate a curve with respect to tangency

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

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

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.

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

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

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

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

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

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

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

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

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

2.2.1 Simple geometric entities

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

- 2D line parallel to another at a distance,
- 2D line parallel to another passing through a point,
- 2D circle passing through two points,
- 2D circle parallel to another at a distance,

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

Each class from *gp* package, such as *Circ, Circ2d, Mirror, Mirror2d*, etc., has the corresponding *MakeCirc, Make↩ Circ2d, 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 Geometric entities manipulated by handle

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

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

- trimmed cone from two coaxial circles.

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

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

- *MakeArcOfCircle*
- *MakeArcOfEllipse*
- *MakeArcOfHyperbola*
- *MakeArcOfParabola*
- *MakeSegment*

2.3 Conversion to and from BSplines

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

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

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

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

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

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

The *Geom2dConvert* package provides the following:

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

The *GeomConvert* package also provides the following:

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

2.4 Points on Curves

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

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

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

GCPnts package provides algorithms to calculate such points:

- *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.

Extrema between Point and Curve / Surface

The *GeomAPI_ProjectPointOnCurve* class allows calculation of all extrema between a point and a curve. Extrema are the lengths of the segments orthogonal to the curve. The *GeomAPI_ProjectPointOnSurface* class allows calculation of all extrema between a point and a surface. Extrema are the lengths of the segments orthogonal to the surface. These classes use the "Projection" criteria for optimization.

Extrema between Curves

The *Geom2dAPI_ExtremaCurveCurve* class allows calculation of all minimal distances between two 2D geometric curves. The *GeomAPI_ExtremaCurveCurve* class allows calculation of all minimal distances between two 3D geometric curves. These classes use Euclidean distance as the criteria for optimization.

Extrema between Curve and Surface

The *GeomAPI_ExtremaCurveSurface* class allows calculation of one extrema between a 3D curve and a surface. Extrema are the lengths of the segments orthogonal to the curve and the surface. This class uses the "Projection" criteria for optimization.

Extrema between Surfaces

The *GeomAPI_ExtremaSurfaceSurface* class allows calculation of one minimal and one maximal distance between two surfaces. This class uses Euclidean distance to compute the minimum, and "Projection" criteria to compute the maximum.

3 2D Geometry

Geom2d package defines geometric objects in 2dspace. All geometric entities are STEP processed. The objects are handled by reference.

In particular, *Geom2d* package provides classes for:

- description of points, vectors and curves,
- their positioning in the plane using coordinate systems,
- their geometric transformation, by applying translations, rotations, symmetries, scaling transformations and combinations thereof.

The following objects are available:

- point,
- Cartesian point,
- vector,
- direction,
- vector with magnitude,
- axis,
- curve,
- line,
- conic: circle, ellipse, hyperbola, parabola,
- rounded curve: trimmed curve, NURBS curve, Bezier curve,
- offset curve.

Before creating a geometric object, it is necessary to decide how the object is handled. The objects provided by *Geom2d* package are handled by reference rather than by value. Copying an instance copies the handle, not the object, so that a change to one instance is reflected in each occurrence of it. If a set of object instances is needed rather than a single object instance, *TColGeom2d* package can be used. This package provides standard and frequently used instantiations of one-dimensional arrays and sequences for curves from *Geom2d* package. All objects are available in two versions:

- handled by reference and
- handled by value.

The key characteristic of *Geom2d* curves is that they are parameterized. Each class provides functions to work with the parametric equation of the curve, and, in particular, to compute the point of parameter u on a curve and the derivative vectors of order 1, 2.., N at this point.

As a consequence of the parameterization, a *Geom2d* curve is naturally oriented.

Parameterization and orientation differentiate elementary *Geom2d*curves from their equivalent as provided by *gp* package. *Geom2d* package provides conversion functions to transform a *Geom2d* object into a *gp* object, and vice-versa, when this is possible.

Moreover, *Geom2d* package provides more complex curves, including Bezier curves, BSpline curves, trimmed curves and offset curves.

Geom2d objects are organized according to an inheritance structure over several levels.

Thus, an ellipse (specific class *Geom2d_Ellipse*) is also a conical curve and inherits from the abstract class *Geom2d_Conic*, while a Bezier curve (concrete class *Geom2d_BezierCurve*) is also a bounded curve and inherits

from the abstract class *Geom2d_BoundedCurve*; both these examples are also curves (abstract class *Geom2d_Curve*). Curves, points and vectors inherit from the abstract class *Geom2d_Geometry,* which describes the properties common to any geometric object from the *Geom2d* package.

This inheritance structure is open and it is possible to describe new objects, which inherit from those provided in the *Geom2d* package, provided that they respect the behavior of the classes from which they are to inherit.

Finally, *Geom2d* objects can be shared within more complex data structures. This is why they are used within topological data structures, for example.

*Geom2d*package uses the services of the *gp* package to:

- implement elementary algebraic calculus and basic analytic geometry,
- describe geometric transformations which can be applied to *Geom2d* objects,
- describe the elementary data structures of *Geom2d* objects.

However, the *Geom2d* package essentially provides data structures and not algorithms. You can refer to the *GCE2d* package to find more evolved construction algorithms for *Geom2d* objects.

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.

In particular, it provides classes for:

- description of points, vectors, curves and surfaces,
- their positioning in 3D space using axis or coordinate systems, and
- their geometric transformation, by applying translations, rotations, symmetries, scaling transformations and combinations thereof.

The following objects are available:

- Point
- Cartesian point
- Vector
- Direction
- Vector with magnitude
- Axis
- Curve
- Line
- Conic: circle, ellipse, hyperbola, parabola
- Offset curve
- Elementary surface: plane, cylinder, cone, sphere, torus
- Bounded curve: trimmed curve, NURBS curve, Bezier curve
- Bounded surface: rectangular trimmed surface, NURBS surface,Bezier surface
- Swept surface: surface of linear extrusion, surface of revolution
- Offset surface.

The key characteristic of *Geom* curves and surfaces is that they are parameterized. Each class provides functions to work with the parametric equation of the curve or surface, and, in particular, to compute:

- the point of parameter u on a curve, or
- the point of parameters (u, v) on a surface. together with the derivative vectors of order 1, 2, ... N at this point.

As a consequence of this parameterization, a Geom curve or surface is naturally oriented.

Parameterization and orientation differentiate elementary Geom curves and surfaces from the classes of the same (or similar) names found in *gp* package. *Geom* package also provides conversion functions to transform a Geom object into a *gp* object, and vice-versa, when such transformation is possible.

Moreover, *Geom* package provides more complex curves and surfaces, including:

- Bezier and BSpline curves and surfaces,

- swept surfaces, for example surfaces of revolution and surfaces of linear extrusion,
- trimmed curves and surfaces, and
- offset curves and surfaces.

Geom objects are organized according to an inheritance structure over several levels. Thus, a sphere (concrete class *Geom_SphericalSurface*) is also an elementary surface and inherits from the abstract class *Geom_ElementarySurface*, while a Bezier surface (concrete class *Geom_BezierSurface*) is also a bounded surface and inherits from the abstract class *Geom_BoundedSurface*; both these examples are also surfaces (abstract class *Geom_Surface*). Curves, points and vectors inherit from the abstract class *Geom_Geometry,* which describes the properties common to any geometric object from the *Geom* package.

This inheritance structure is open and it is possible to describe new objects, which inherit from those provided in the Geom package, on the condition that they respect the behavior of the classes from which they are to inherit.

Finally, Geom objects can be shared within more complex data structures. This is why they are used within topological data structures, for example.

If a set of object instances is needed rather than a single object instance, *TColGeom* package can be used. This package provides instantiations of one- and two-dimensional arrays and sequences for curves from *Geom* package. All objects are available in two versions:

- handled by reference and
- handled by value.

The *Geom* package uses the services of the *gp* package to:

- implement elementary algebraic calculus and basic analytic geometry,
- describe geometric transformations which can be applied to Geom objects,
- describe the elementary data structures of Geom objects.

However, the Geom package essentially provides data structures, not algorithms.

You can refer to the *GC* package to find more evolved construction algorithms for Geom objects.

5 Topology

OCCT Topology allows accessing and manipulating data of objects without dealing with their 2D or 3D representations. Whereas OCCT Geometry provides a description of objects in terms of coordinates or parametric values, Topology describes data structures of objects in parametric space. These descriptions use location in and restriction of parts of this space.

Topological library allows you to build pure topological data structures. Topology defines relationships between simple geometric entities. In this way, you can model complex shapes as assemblies of simpler entities. Due to a built-in non-manifold (or mixed-dimensional) feature, you can build models mixing:

- 0D entities such as points;
- 1D entities such as curves;
- 2D entities such as surfaces;
- 3D entities such as volumes.

You can, for example, represent a single object made of several distinct bodies containing embedded curves and surfaces connected or non-connected to an outer boundary.

Abstract topological data structure describes a basic entity – a shape, which can be divided into the following component topologies:

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

The wire and the solid can be either infinite or closed.

A face with 3D underlying geometry may also refer to a collection of connected triangles that approximate the underlying surface. The surfaces can be undefined leaving the faces represented by triangles only. If so, the model is purely polyhedral.

Topology defines the relationship between simple geometric entities, which can thus be linked together to represent complex shapes.

Abstract Topology is provided by six packages. The first three packages describe the topological data structure used in Open CASCADE Technology:

- *TopAbs* package provides general resources for topology-driven applications. It contains enumerations that are used to describe basic topological notions: topological shape, orientation and state. It also provides methods to manage these enumerations.
- *TopLoc* package provides resources to handle 3D local coordinate systems: *Datum3D*and *Location*. *Datum3D* describes an elementary coordinate system, while *Location* comprises a series of elementary coordinate systems.
- *TopoDS* package describes classes to model and build data structures that are purely topological.

Three additional packages provide tools to access and manipulate this abstract topology:

- *TopTools* package provides basic tools to use on topological data structures.

- *TopExp* package provides classes to explore and manipulate the topological data structures described in the TopoDS package.
- *BRepTools* package provides classes to explore, manipulate, read and write BRep data structures. These more complex data structures combine topological descriptions with additional geometric information, and include rules for evaluating equivalence of different possible representations of the same object, for example, a point.

5.1 Shape content

The **TopAbs** package provides general enumerations describing the basic concepts of topology and methods to handle these enumerations. It contains no classes. This package has been separated from the rest of the topology because the notions it contains are sufficiently general to be used by all topological tools. This avoids redefinition of enumerations by remaining independent of modeling resources. The TopAbs package defines three notions:

- **Type** *TopAbs_ShapeEnum*;
- **Orientation** *TopAbs_Orientation* ;
- **State** *StateTopAbs_State*

5.1.1 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.

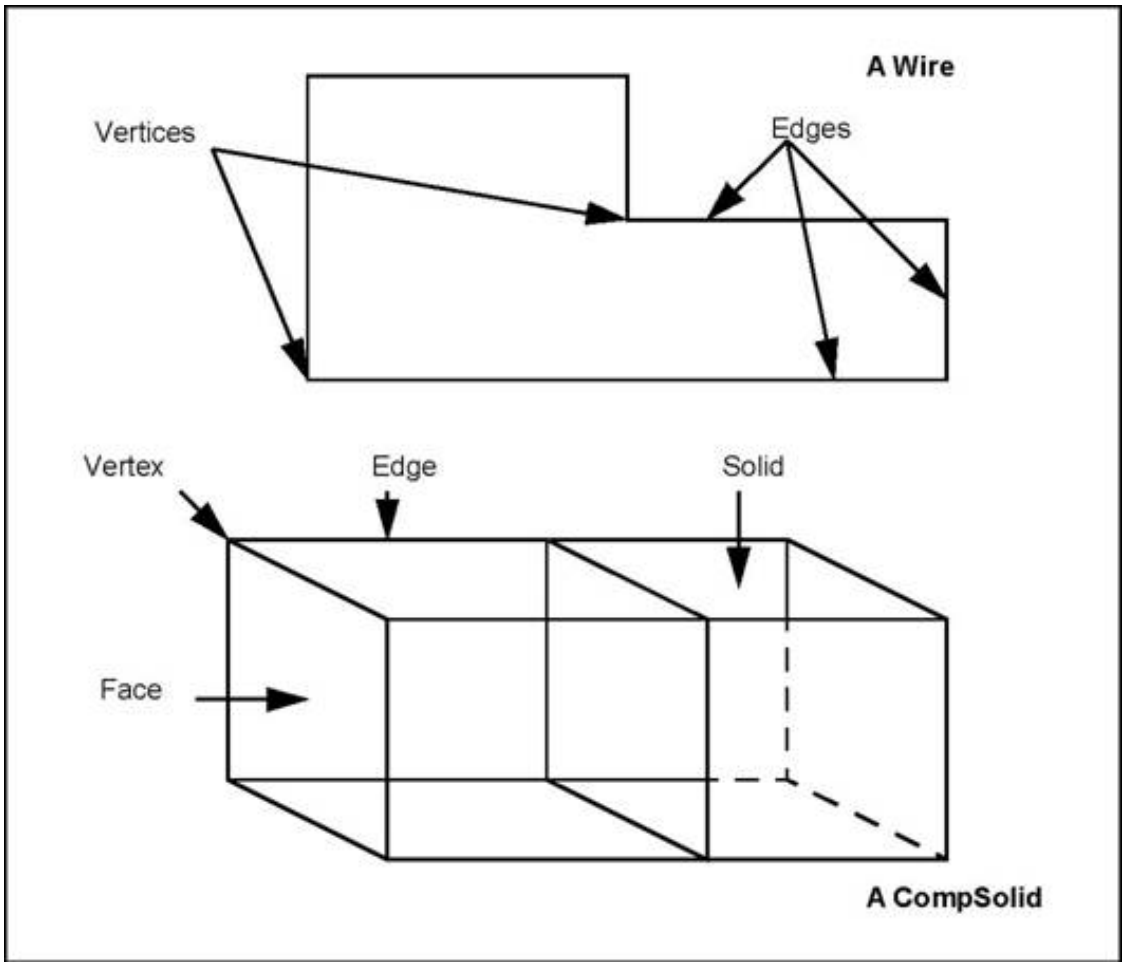

Figure 3: ShapeEnum

5.1.2 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.

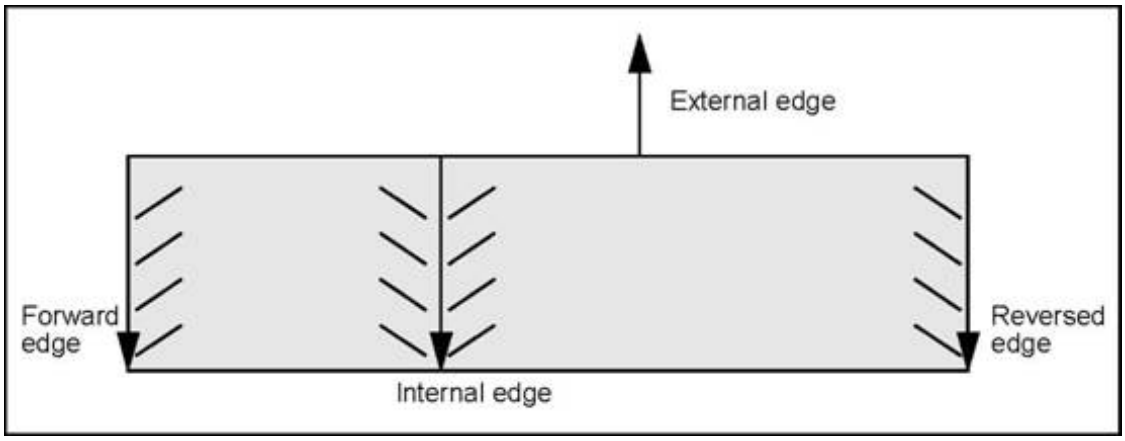

Figure 4: Four Orientations

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 and the intersection vertex would be 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

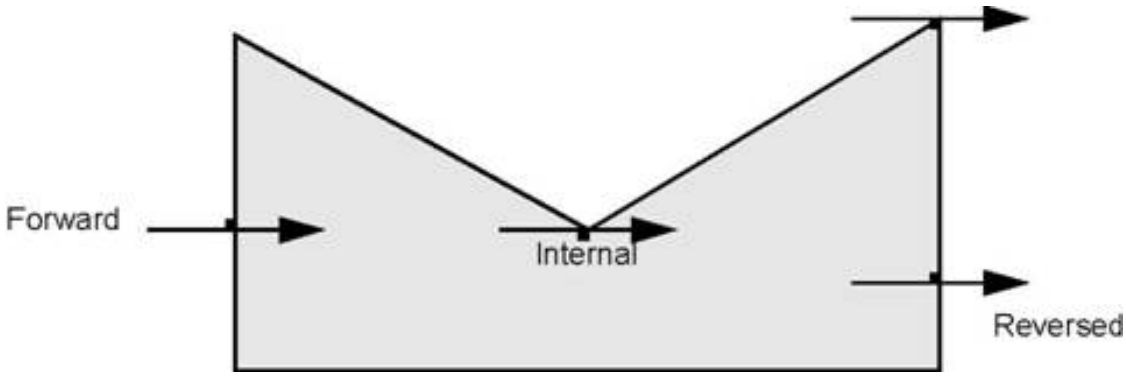

Figure 5: Four orientations of intersection vertices

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

5.1.3 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.

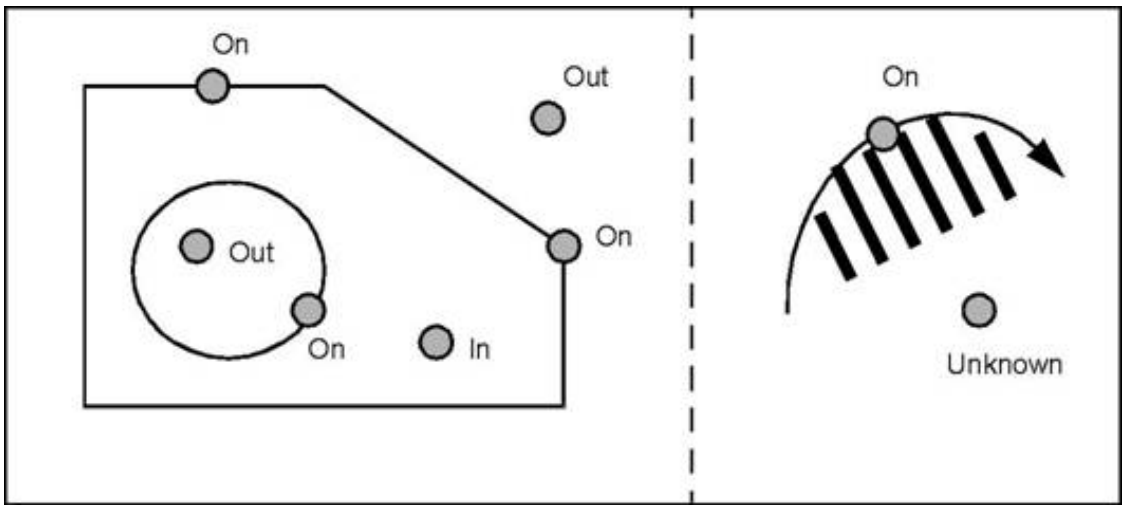

Figure 6: The four states

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.

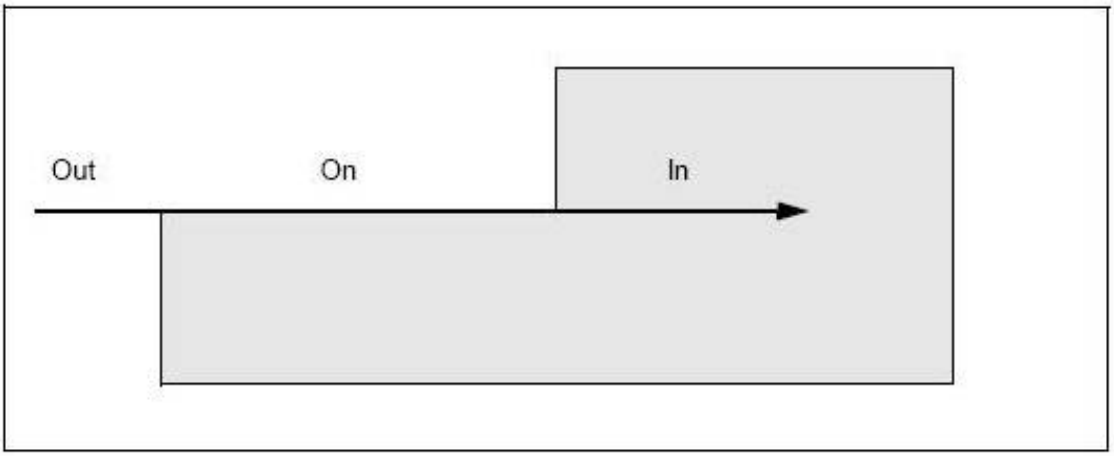

Figure 7: State specifies the parts of an edge intersecting a face

5.1.4 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.

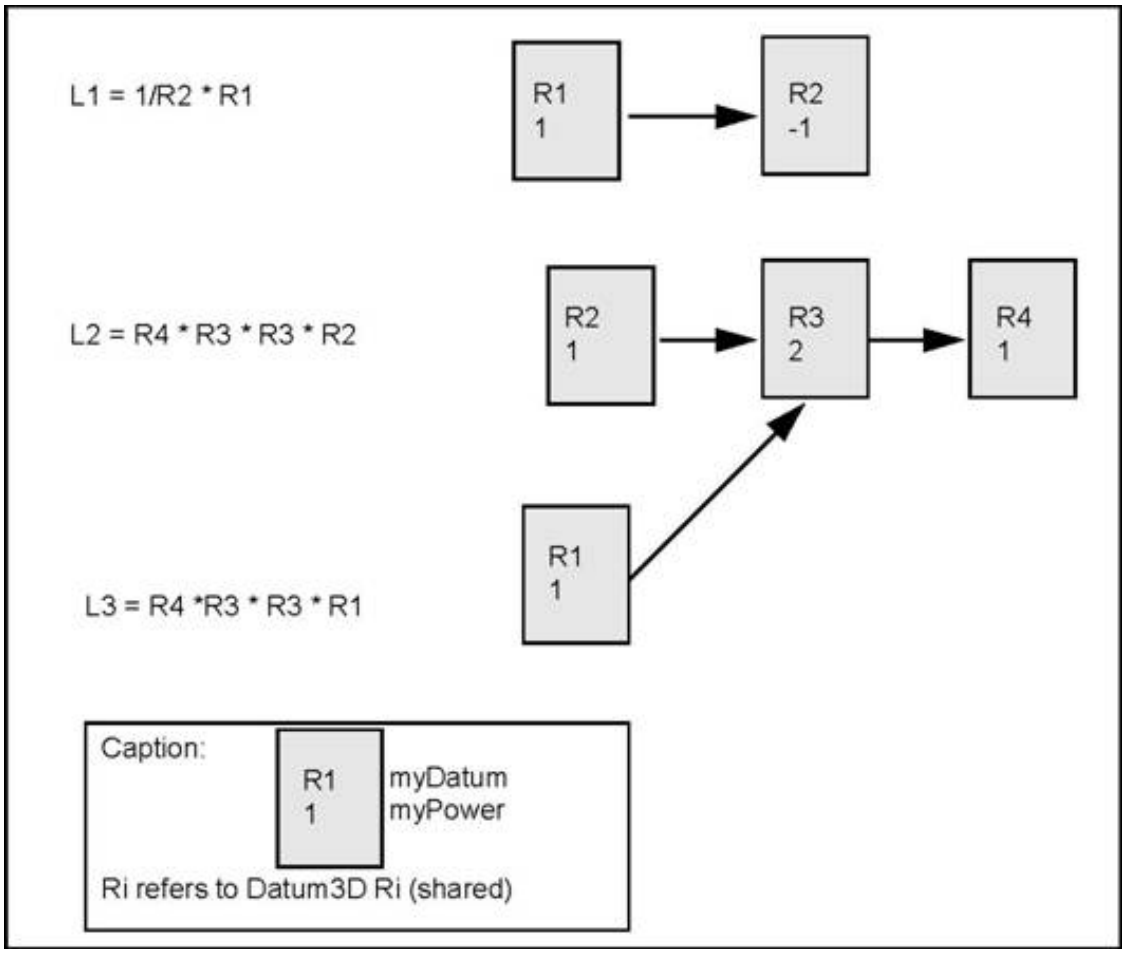

Figure 8: Structure of TopLoc_Location

Two reference coordinates are equal if they are made up of the same elementary coordinates in the same order. There is no numerical comparison. Two coordinates can thus correspond to the same transformation without being equal if they were not built from the same elementary coordinates.

For example, consider three elementary coordinates: R1, R2, R3 The composite coordinates are: C1 = R1 $*$ R2, C2 = R2 $*$ R3 C3 = C1 $*$ R3 C4 = R1 $*$ C2

NOTE C3 and C4 are equal because they are both R1 $*$ R2 $*$ R3.

The *TopLoc* package is chiefly targeted at the topological data structure, but it can be used for other purposes.

Change of coordinates

TopLoc_Datum3D class represents a change of elementary coordinates. Such changes must be shared so this class inherits from *Standard_Transient*. The coordinate is represented by a transformation *gp_Trsfpackage*. This transformation has no scaling factor.

5.2 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.

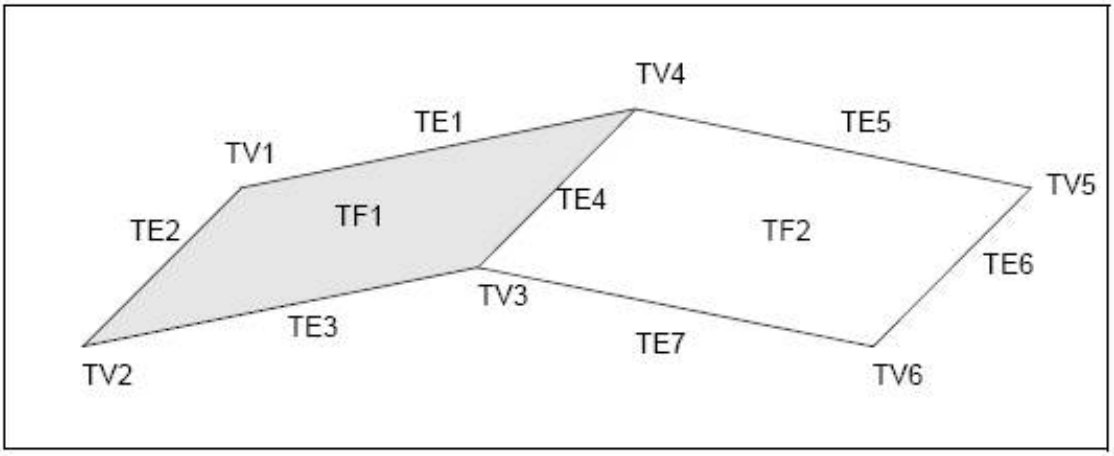

Figure 9: Structure of a shell formed from two faces

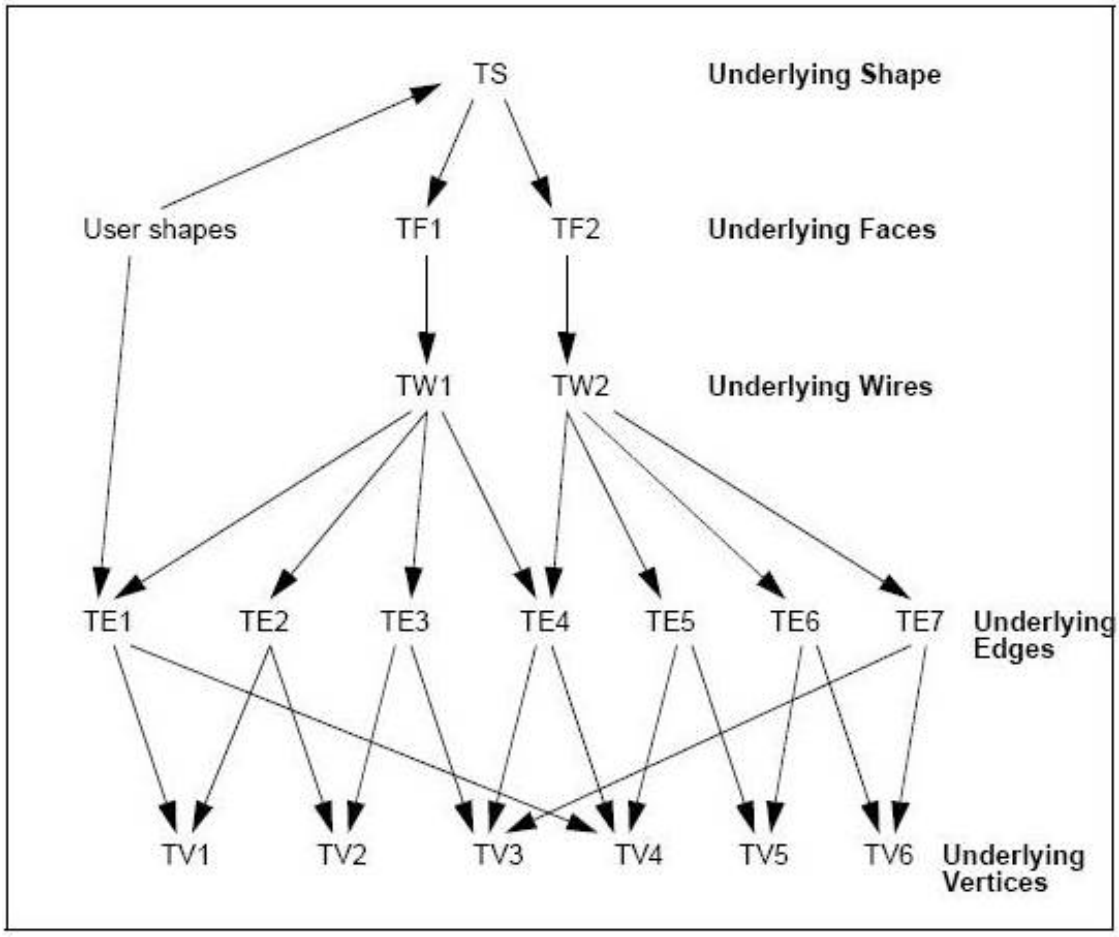

Figure 10: Data structure of the above shell

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(T5,TV6),TE7(TV3,TV6).

Note that this data structure does not contain any *back references*. All references go from more complex underlying shapes to less complex ones. The techniques used to access the information are described later. The data structure is as compact as possible. Sub-objects can be shared among different objects.

Two very similar objects, perhaps two versions of the same object, might share identical sub-objects. The usage of local coordinates in the data structure allows the description of a repetitive sub-structure to be shared.

The compact data structure avoids the loss of information associated with copy operations which are usually used in creating a new version of an object or when applying a coordinate change.

The following figure shows a data structure containing two versions of a solid. The second version presents a series of identical holes bored at different positions. The data structure is compact and yet keeps all information on the sub-elements.

The three references from *TSh2* to the underlying face *TFcyl* have associated local coordinate systems, which correspond to the successive positions of the hole.

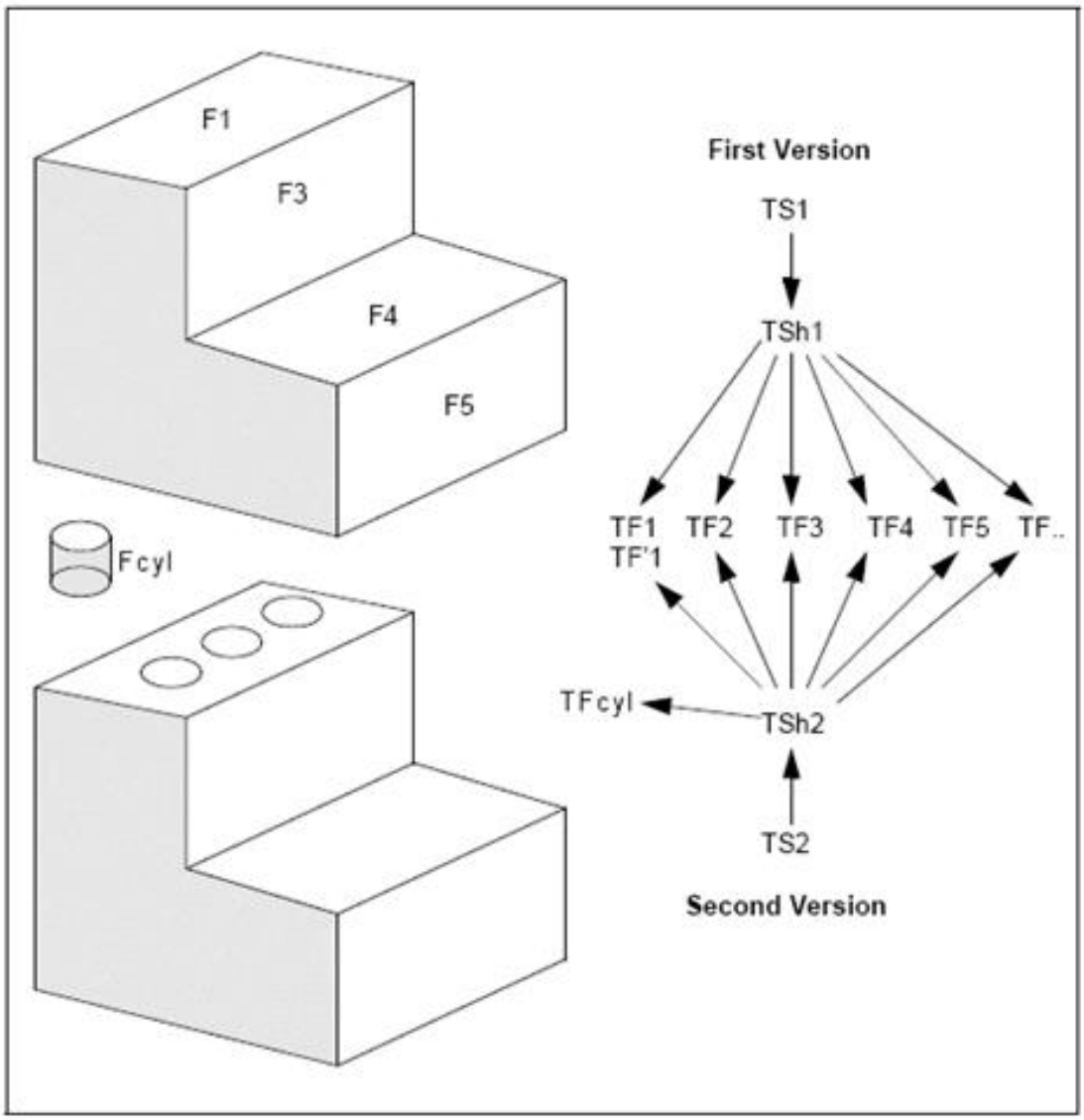

Figure 11: Data structure containing two versions of a solid

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 inTopoDS. 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
  }
}
```

5.3 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 ina 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 theMap
  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;
    }
  }
}
```

5.3.1 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 *NCollection_Array1* with *TopoDS_Shape*.
- *TopTools_SequenceOfShape* – instantiation of the *NCollection_Sequence* with *TopoDS_Shape*.
- *TopTools_MapOfShape* - instantiation of the *NCollection_Map*. Allows the construction of sets of shapes.
- *TopTools_IndexedMapOfShape* - instantiation of the *NCollection_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 is1 + 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 Mapand return the size.
  TopTools_MapOfShape M;
  MapShapes(aShape,M);
  return M.Extent();
}
```

Note For more details about Maps, 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 wholestructure of aShape using aBuilder.
  // Stores all thesub-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_Array1OfShapetheCopies(1,theMap.Extent());

// Use a recursivefunction 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 thecorrect 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);
    }
  }
}
```

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} :

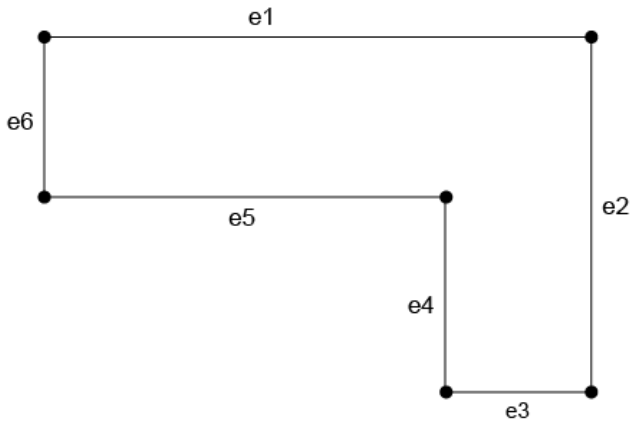

Figure 12: A wire composed of 6 edges.

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());
}
```

6 Properties of Shapes

6.1 Local Properties of Shapes

BRepLProp package provides the Local Properties of Shapes component, which contains algorithms computing various local properties on edges and faces in a BRep model.

The local properties which may be queried are:

- for a point of parameter u on a curve which supports an edge :
  - the point,
  - the derivative vectors, up to the third degree,
  - the tangent vector,
  - the normal,
  - the curvature, and the center of curvature;
- for a point of parameter (u, v) on a surface which supports a face :
  - the point,
  - the derivative vectors, up to the second degree,
  - the tangent vectors to the u and v isoparametric curves,
  - the normal vector,
  - the minimum or maximum curvature, and the corresponding directions of curvature;
- the degree of continuity of a curve which supports an edge, built by the concatenation of two other edges, at their junction point.

Analyzed edges and faces are described as *BRepAdaptor* curves and surfaces, which provide shapes with an interface for the description of their geometric support. The base point for local properties is defined by its u parameter value on a curve, or its (u, v) parameter values on a surface.

6.2 Local Properties of Curves and Surfaces

The "Local Properties of Curves and Surfaces" component provides algorithms for computing various local properties on a Geom curve (in 2D or 3D space) or a surface. It is composed of:

- *Geom2dLProp* package, which allows computing Derivative and Tangent vectors (normal and curvature) of a parametric point on a 2D curve;
- *GeomLProp* package, which provides local properties on 3D curves and surfaces
- *LProp* package, which provides an enumeration used to characterize a particular point on a 2D curve.

Curves are either *Geom_Curve* curves (in 3D space) or *Geom2d_Curve* curves (in the plane). Surfaces are *Geom_Surface* surfaces. The point on which local properties are calculated is defined by its u parameter value on a curve, and its (u,v) parameter values on a surface.

It is possible to query the same local properties for points as mentioned above, and additionally for 2D curves:

- the points corresponding to a minimum or a maximum of curvature;
- the inflection points.

Example: How to check the surface concavity

To check the concavity of a surface, proceed as follows:

1. Sample the surface and compute at each point the Gaussian curvature.
2. If the value of the curvature changes of sign, the surface is concave or convex depending on the point of view.
3. To compute a Gaussian curvature, use the class *SLprops* from *GeomLProp*, which instantiates the generic class *SLProps* from *LProp* and use the method *GaussianCurvature*.

6.3 Continuity of Curves and Surfaces

Types of supported continuities for curves and surfaces are described in *GeomAbs_Shape* enumeration.

In respect of curves, the following types of continuity are supported (see the figure below):

- C0 (*GeomAbs_C0*) - parametric continuity. It is the same as G0 (geometric continuity), so the last one is not represented by separate variable.
- G1 (*GeomAbs_G1*) - tangent vectors on left and on right are parallel.
- C1 (*GeomAbs_C1*) - indicates the continuity of the first derivative.
- G2 (*GeomAbs_G2*) - in addition to G1 continuity, the centers of curvature on left and on right are the same.
- C2 (*GeomAbs_C2*) - continuity of all derivatives till the second order.
- C3 (*GeomAbs_C3*) - continuity of all derivatives till the third order.
- CN (*GeomAbs_CN*) - continuity of all derivatives till the N-th order (infinite order of continuity).

Note: Geometric continuity (G1, G2) means that the curve can be reparametrized to have parametric (C1, C2) continuity.

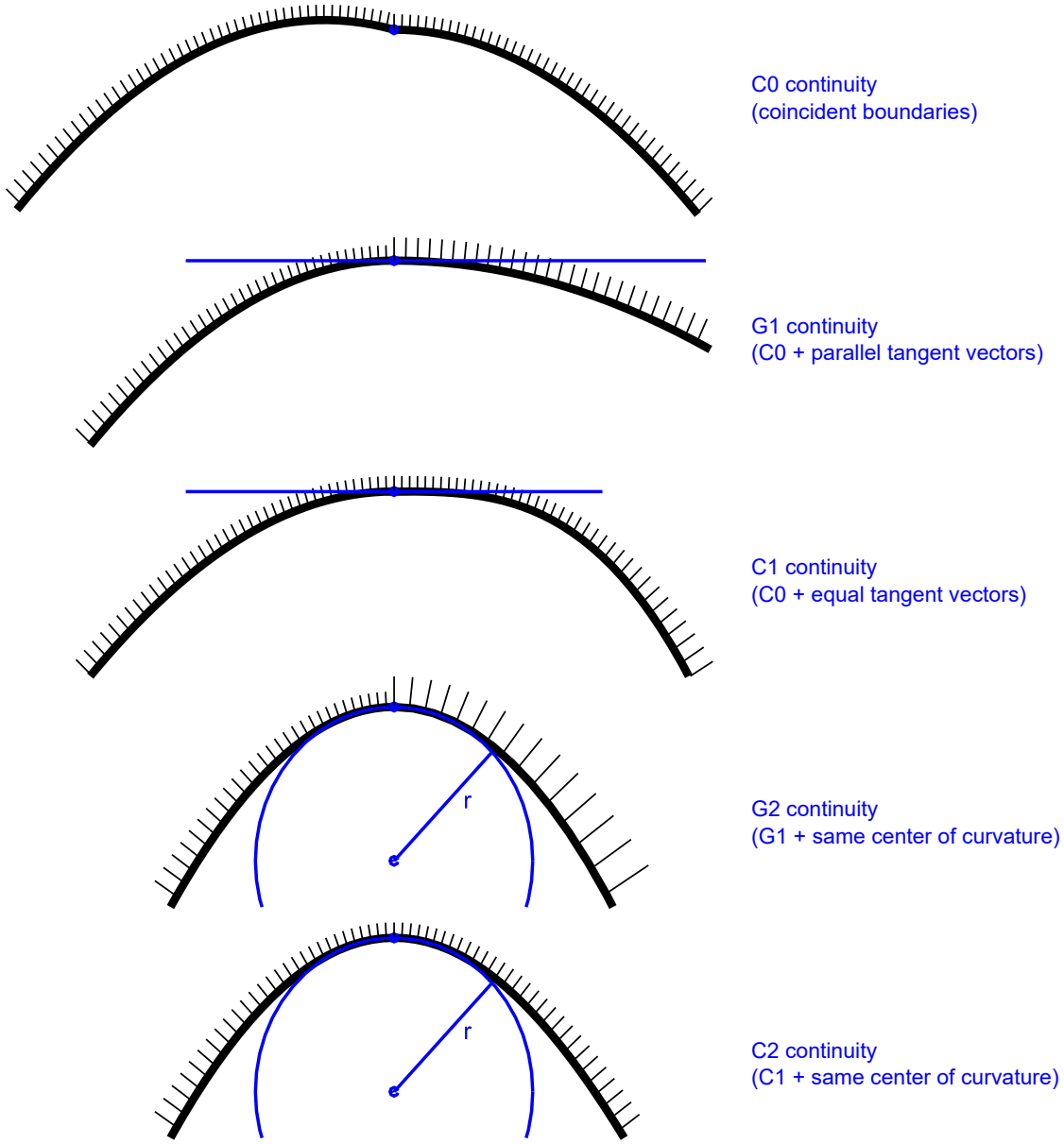

Figure 13: Continuity of Curves

The following types of surface continuity are supported:

- C0 (*GeomAbs_C0*) - parametric continuity (the surface has no points or curves of discontinuity).
- G1 (*GeomAbs_G1*) - surface has single tangent plane in each point.
- C1 (*GeomAbs_C1*) - indicates the continuity of the first derivatives.
- G2 (*GeomAbs_G2*) - in addition to G1 continuity, principal curvatures and directions are continuous.
- C2 (*GeomAbs_C2*) - continuity of all derivatives till the second order.
- C3 (*GeomAbs_C3*) - continuity of all derivatives till the third order.
- CN (*GeomAbs_CN*) - continuity of all derivatives till the N-th order (infinite order of continuity).

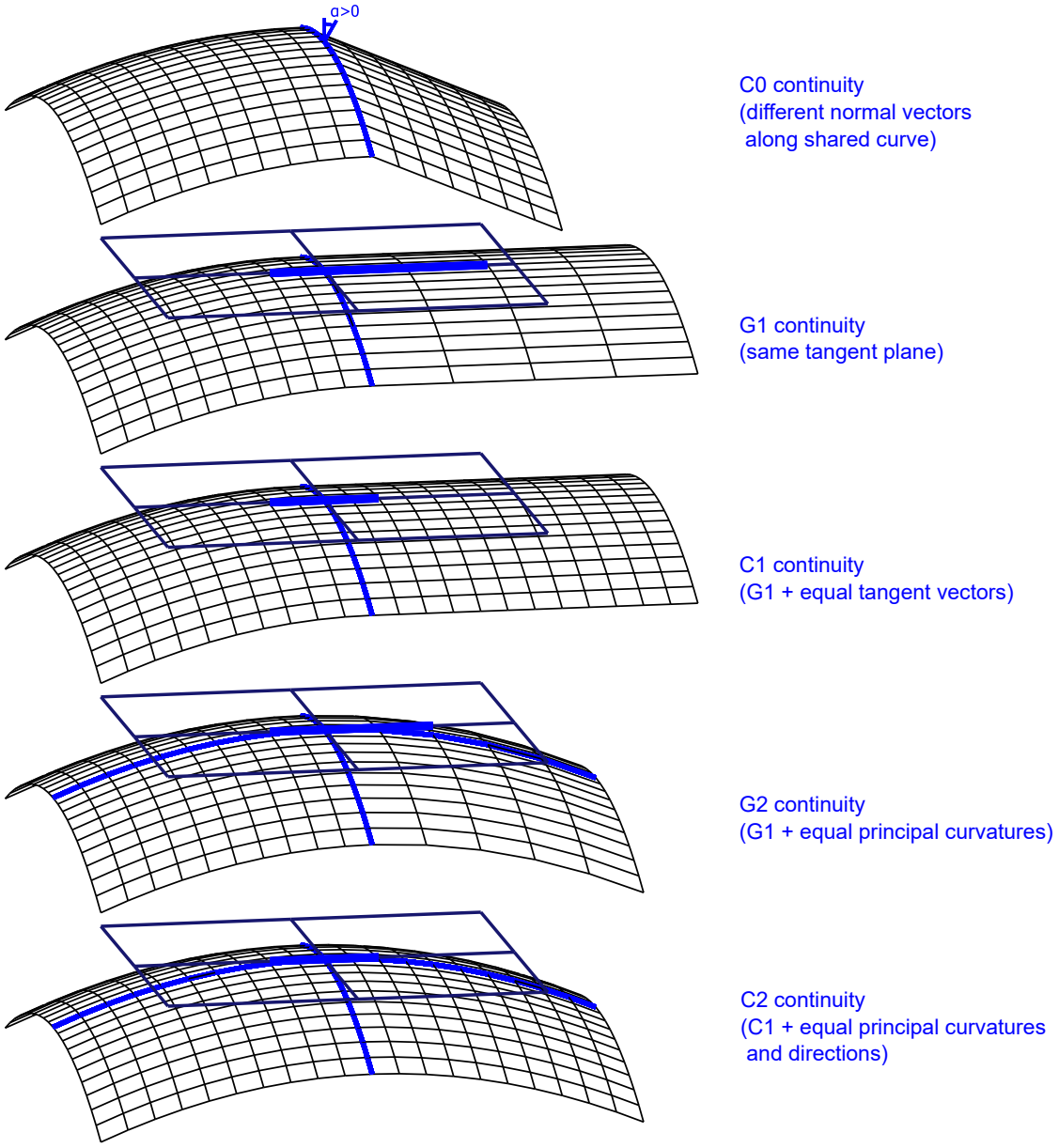

Figure 14: Continuity of Surfaces

Against single surface, the connection of two surfaces (see the figure above) defines its continuity in each intersection point only. Smoothness of connection is a minimal value of continuities on the intersection curve.

6.4 Regularity of Shared Edges

Regularity of an edge is a smoothness of connection of two faces sharing this edge. In other words, regularity is a minimal continuity between connected faces in each point on edge.

Edge's regularity can be set by *BRep_Builder::Continuity* method. To get the regularity use *BRep_Tool::Continuity* method.

Some algorithms like Fillet set regularity of produced edges by their own algorithms. On the other hand, some other algorithms (like Boolean Operations, Shape Healing, etc.) do not set regularity. If the regularity is needed to be set correctly on a shape, the method *BRepLib::EncodeRegularity* can be used. It calculates and sets correct values for all edges of the shape.

The regularity flag is extensively used by the following high level algorithms: Chamfer, Draft Angle, Hidden Line

Removal, Gluer.

6.5 Global Properties of Shapes

The Global Properties of Shapes component provides algorithms for computing the global properties of a composite geometric system in 3D space, and frameworks to query the computed results.

The global properties computed for a system are :

- mass,
- mass center,
- matrix of inertia,
- moment about an axis,
- radius of gyration about an axis,
- principal properties of inertia such as principal axis, principal moments, and principal radius of gyration.

Geometric systems are generally defined as shapes. Depending on the way they are analyzed, these shapes will give properties of:

- lines induced from the edges of the shape,
- surfaces induced from the faces of the shape, or
- volumes induced from the solid bounded by the shape.

The global properties of several systems may be brought together to give the global properties of the system composed of the sum of all individual systems.

The Global Properties of Shapes component is composed of:

- seven functions for computing global properties of a shape: one function for lines, two functions for surfaces and four functions for volumes. The choice of functions depends on input parameters and algorithms used for computation (*BRepGProp* global functions),
- a framework for computing global properties for a set of points (*GProp_PGProps*),
- a general framework to bring together the global properties retained by several more elementary frameworks, and provide a general programming interface to consult computed global properties.

Packages *GeomLProp* and *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.

6.6 Adaptors for Curves and Surfaces

Some Open CASCADE Technology general algorithms may work theoretically on numerous types of curves or surfaces.

To do this, they simply get the services required of the analyzed curve or surface through an interface so as to a single API, whatever the type of curve or surface. These interfaces are called adaptors.

For example, *Adaptor3d_Curve* is the abstract class which provides the required services by an algorithm which uses any 3d curve.

GeomAdaptor package provides interfaces:

- On a Geom curve;
- On a curve lying on a Geom surface;
- On a Geom surface;

Geom2dAdaptor package provides interfaces :

- On a *Geom2d* curve.

BRepAdaptor package provides interfaces:

- On a Face
- On an Edge

When you write an algorithm which operates on geometric objects, use *Adaptor3d* (or *Adaptor2d*) objects.

As a result, you can use the algorithm with any kind of object, if you provide for this object an interface derived from *Adaptor3d* or *Adaptor2d*. These interfaces are easy to use: simply create an adapted curve or surface from a *Geom2d* curve, and then use this adapted curve as an argument for the algorithm? which requires it.

7 Bounding boxes

Bounding boxes are used in many OCCT algorithms. The most common use is as a filter avoiding check of excess interferences between pairs of shapes (check of interferences between bounding boxes is much simpler then between shapes and if they do not interfere then there is no point in searching interferences between the corresponding shapes). Generally, bounding boxes can be divided into two main types:

- axis-aligned bounding box (AABB) is the box whose edges are parallel to the axes of the World Coordinate System (WCS);
- oriented BndBox (OBB) is defined in its own coordinate system that can be rotated with respect to the WCS. Indeed, an AABB is a specific case of OBB.

The image below illustrates the example, when using OBB is better than AABB.

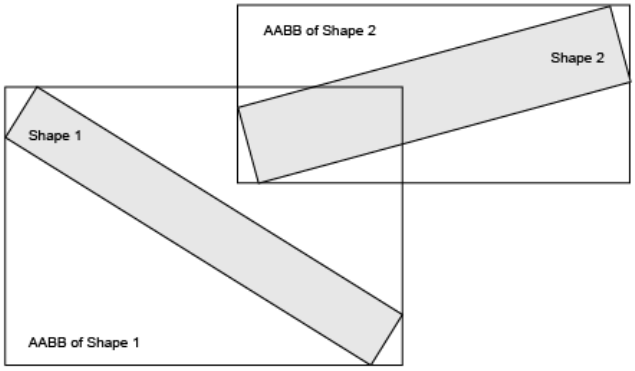

Figure 15: Illustrating the problem with AABB.

AABBs in this picture are interfered. Therefore, many OCCT algorithms will spend much time to interfere the shapes. However, if we check OBBs, which are not interfered, then searching of interferences between the shapes will not be necessary. At that, creation and analysis of OBBs takes significantly more time than the analogical operations with AABB.

Later in this section, the bounding boxes having the smallest surface area will be called *optimal*.

In OCCT, bounding boxes are defined in *Bnd* package. *Bnd_Box* class defines AABB, *Bnd_OBB* class defines OBB. These classes contain the following common methods (this list is not complete; see the documentation about the corresponding class for detailed information):

- *IsVoid* method indicates whether the bounding box is empty (uninitialized).
- *SetVoid* method clears the existing bounding box.
- *Enlarge(...)* extends the current bounding box.
- *Add(...)* extends the bounding box as necessary to include the object (a point, a shape, etc.) passed as the argument.
- *IsOut(...)* checks whether the argument is inside/outside of the current BndBox.

BRepBndLib class contains methods for creation of bounding boxes (both AABB and OBB) from the shapes.

7.1 Brief description of some algorithms working with OBB

7.1.1 Creation of OBB from set of points

The algorithm is described in "Fast Computation of Tight Fitting Oriented Bounding Boxes" by Thomas Larsson and Linus Källberg (FastOBBs.pdf). It includes the following steps:

1. Choose $N_a(N_a \geq 3)$ initial axes.
2. Project every given point to the every chosen (in item 1) axis. At that, "minimal" and "maximal" points of every axis (i.e. point having minimal and maximal parameter (correspondingly) of the projection to this axis) are chosen. I.e. $2 * N_a$ points will be held and this set can contain equal points. Later (unless otherwise specified) in this algorithm we will work with these $2 * N_a$ points only.
3. Choose one pair of points among all pairs of "minimal" and "maximal" points of every axis (from item 1), with two furthest points. Let p_0 and p_1 be the "minimal" and "maximal" point of this pair.
4. Create an axis $\mathbf{e_0}\left\{\overrightarrow{p_0p_1}\right\}$ (i.e. having direction $\overrightarrow{p_0p_1}$).
5. Choose the point p_2 (from the set defined in item 2) which is in the maximal distance from the infinite line directed along $\mathbf{e_0}$ axis.

Further, let us consider the triangle $T_0\left\langle p_0, p_1, p_2\right\rangle$ (i.e. having vertices p_0, p_1 and p_2). Namely:

6. Create new axes: $\mathbf{e_1}\left\{\overrightarrow{p_1p_2}\right\}$, $\mathbf{e_2}\left\{\overrightarrow{p_2p_0}\right\}$, $\mathbf{n}\left\{\overrightarrow{\mathbf{e_0}} \times \overrightarrow{\mathbf{e_1}}\right\}$, $\mathbf{m_0}\left\{\overrightarrow{\mathbf{e_0}} \times \overrightarrow{\mathbf{n}}\right\}$, $\mathbf{m_1}\left\{\overrightarrow{\mathbf{e_1}} \times \overrightarrow{\mathbf{n}}\right\}$, $\mathbf{m_2}\left\{\overrightarrow{\mathbf{e_2}} \times \overrightarrow{\mathbf{n}}\right\}$.
7. Create OBBs based on the following axis: $\left\{\mathbf{e_0}\vdots\mathbf{m_0}\vdots\mathbf{n}\right\}$, $\left\{\mathbf{e_1}\vdots\mathbf{m_1}\vdots\mathbf{n}\right\}$ and $\left\{\mathbf{e_2}\vdots\mathbf{m_2}\vdots\mathbf{n}\right\}$. Choose optimal OBB.
8. Choose the points q_0 and q_1 (from the set defined in item 2), which are in maximal distance from the plane of the triangle T_0 (from both sides of this plane). At that, q_0 has minimal coordinate along the axis $\mathbf{n}$, q_1 has a maximal coordinate.
9. Repeat the step 6...7 for the triangles $T_1\left\langle p_0, p_1, q_0\right\rangle$, $T_2\left\langle p_1, p_2, q_0\right\rangle$, $T_3\left\langle p_0, p_2, q_0\right\rangle$, $T_4\left\langle p_0, p_1, q_1\right\rangle$, $T_5\left\langle p_1, p_2, q_1\right\rangle$, $T_6\left\langle p_0, p_2, q_1\right\rangle$.
10. Compute the center of OBB and its half dimensions.
11. Create OBB using the center, axes and half dimensions.

7.1.2 Creation of Optimal OBB from set of points

For creation of the optimal OBB from set of points the same algorithm as described above is used but with some simplifications in logic and increased computation time. For the optimal OBB it is necessary to check all possible axes which can be created by the extremal points. And since the extremal points are only valid for the initial axes it is necessary to project the whole set of points on each axis. This approach usually provides much tighter OBB but the performance is lower. The complexity of the algorithm is still linear and with use of BVH for the set of points it is O(N + C∗log(N)).

Here is the example of optimal and not optimal OBB for the model using the set of 125K nodes:

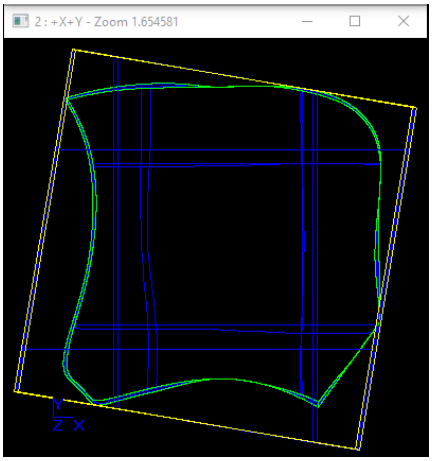

Figure 16: Not optimal OBB by DiTo-14

Computation of the not optimal OBB in this case took 0.007 sec, optimal - 0.1 sec, which is about 14 times slower. Such performance is comparable to creation of the OBB for this shape by PCA approach (see below) which takes about 0.17 sec.

The computation of optimal OBB is controlled by the same *theIsOptimal* flag in the BRepBndLib::AddOBB method as for PCA algorithm.

These algorithms are implemented in the *Bnd_OBB::ReBuild(...)* method.

7.1.3 Creation of OBB based on Axes of inertia

The algorithm contains the following steps:

1. Calculate three inertia axes, which will be the axes of the OBB.
2. Transform the source object ∗(TopoDS_Shape)∗ into the local coordinate system based on the axes from item 1.
3. Create an AABB for the shape obtained in the item 2.
4. Compute the center of AABB and its half dimensions.
5. Transform the center into the WCS.
6. Create OBB using the center, axes and half dimensions.

7.1.4 Method IsOut for a point

1. Project the point to each axis.
2. Check, whether the absolute value of the projection parameter greater than the correspond half-dimension. In this case, *IsOut* method will return TRUE.

7.1.5 Method IsOut for another OBB

According to the "Separating Axis Theorem for Oriented Bounding Boxes", it is necessary to check the 15 separating axes: 6 axes of the boxes and 9 are their cross products.
The algorithm of analyzing axis $\mathbf{l}$ is following:

1. Compute the "length" according to the formula: $L_j = \sum_{i=0}^{2} H_i \cdot \left| \overrightarrow{\mathbf{a_i}} \cdot \overrightarrow{\mathbf{l}} \right|$. Here, $\mathbf{a_i}$ is an i-th axis (X-axis, Y-axis, Z-axis) of j-th BndBox (j=1...2). H_i is a half-dimension along i-th axis.
2. If $\left| \overrightarrow{C_1C_2} \cdot \overrightarrow{\mathbf{l}} \right| > L_1 + L_2$ (where C_j is the center of j-th OBB) then the considered OBBs are not interfered in terms of the axis $\mathbf{l}$.

If OBBs are not interfered in terms of at least one axis (of 15) then they are not interfered at all.

7.1.6 Method Add for point or another bounding box

Create a new OBB (see the section Creation of OBB from set of points) based on the source point and all vertices of the given bounding boxes.

7.2 Add a shape

Method *BRepBndLib::AddOBB(...)* allows creating the bounding box from a complex object ∗(TopoDS_Shape)∗. This method uses both algorithms described in the sections Creation of OBB from set of points and sections Creation of OBB based on Axes of inertia.

The first algorithm is used if the outer shell of the shape can be represented by a set of points contained in it. Namely, only the following elements are the source of set of points:

- Nodes of triangulation;
- Nodes of *Poly_Polygon3D*;
- Vertices of edges with a linear 3D-curve lying in the planar face;
- Vertices of edges with a linear 3D-curve if the source shape does not contain a more complex topological structure (e.g. the source shape is a compound of edges);
- Vertices if the source shape does not contain a more complex topological structure (e.g. the source shape is a compound of vertices).

If the required set of points cannot be extracted then the algorithm from section Creation of OBB based on Axes of inertia is used for OBB creation.

The package *BRepBndLib* contains methods *BRepBndLib::Add(...), BRepBndLib::AddClose(...)* and *BRepBndLib::AddOptimal(...)* for creation of AABB of a shape. See the reference manual for the detailed information.

7.3 Limitations of algorithm for OBB creation.

1. The algorithm described in the section Creation of OBB from set of points works significantly better (finds resulting OBB with less surface area) and faster than the algorithm from the section Creation of OBB based on Axes of inertia. Nevertheless, (in general) the result returned by both algorithms is not always optimal (i.e. sometimes another OBB exists with a smaller surface area). Moreover, the first method does not allow computing OBBs of shapes with a complex geometry.
2. Currently, the algorithm of OBB creation is implemented for objects in 3D space only.