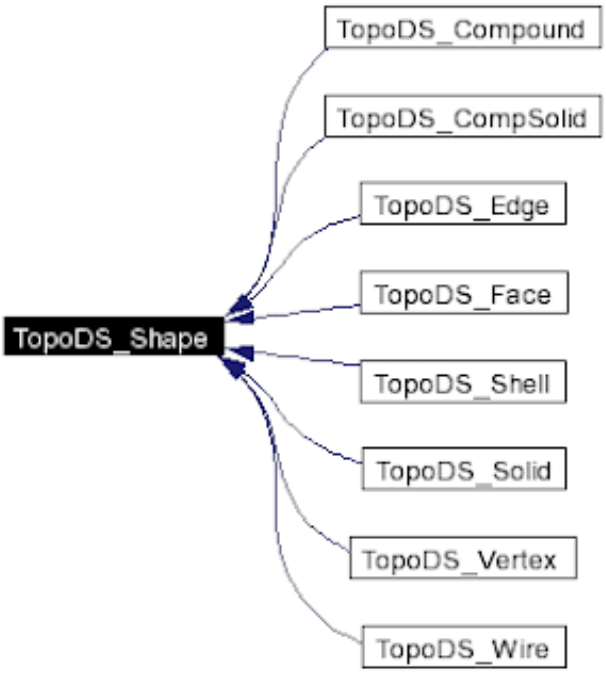
TOPOLOGY AND GEOMETRY IN OPEN CASCADE. PART 1

BY ROMAN LYGIN - 23:36

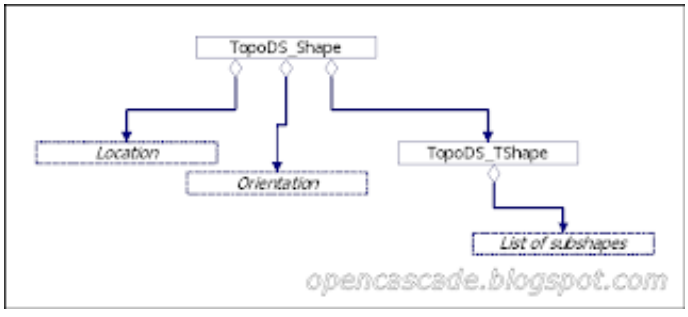
Let us consider these fundamental concepts of Open CASCADE. From time to time questions about them pop up on the forum, so I hope this post will be useful for many readers. Understanding these concepts is important if you have to work with most modeling algorithms and especially if you want to develop you own. You might also want to re-read various chapters from the Modeling Data User's Guide to refresh you memory.

Abstract topology Open CASCADE topology is designed with reference to the STEP standard ISO-10303-42. Perhaps reading some its concepts would be helpful (I myself did this once in 1997).

The structure is an oriented one-way graph, where parents refer to their children, and there are no back references. Abstract structure is implemented as C++ classes from the TopoDS package. Here is an inheritance diagram taken from the Doxygen-generated documentation.



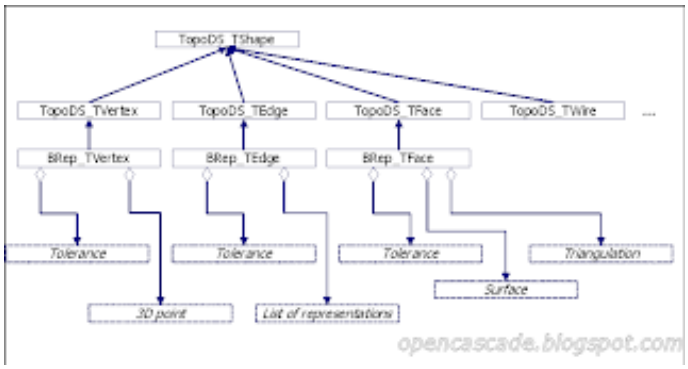
TopoDS_Shape is manipulated by value and contains 3 fields – location, orientation and a myTShape handle (of the TopoDS_TShape type) – see the diagram below (this and other contain only most important fields):



myTShape and Location are used to share data between various shapes and thus save huge amounts of memory. For example, an edge belonging to two faces has equal Locations and myTShape fields but different Orientations (Forward in context of one face and Reversed in one of the other).

Connection with Geometry Now let's consider how this abstract topology structure is connected with geometry.

This is done by inheriting abstract topology classes from the TopoDS package by those implementing a boundary representation model (from the BRep package). Only 3 types of topological objects have geometric representations – vertex, edge, and face (see the diagram below).



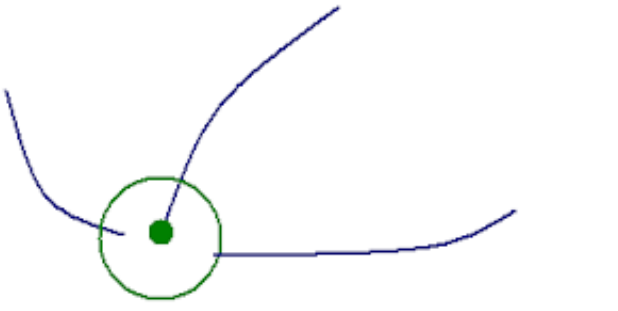
Let's consider them more closely.

TOPOLOGY AND GEOMETRY IN OPEN CASCADE. PART 2

BY ROMAN LYGIN - 11:44

Vertex Vertex has a primary geometric representation which is a point in 3D space encoded as gp_Pnt. It may have other representations but they are virtually never used in practice.

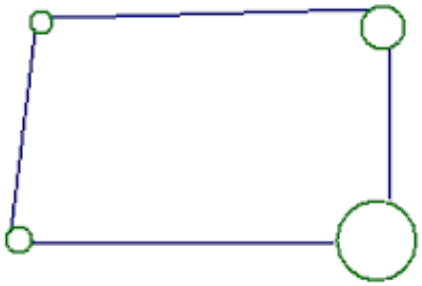
Another important attribute of a vertex is its tolerance which indicates possible inaccuracy of its placement. Geometrical meaning of the vertex tolerance T is a sphere with a radius T centered in vertex's point. This sphere must encompass curves ends of all edges connected at that point.



Vertex tolerance must encopmass edge curves ends

Tolerances Let me elaborate a little bit on tolerances.

Unlike some other geometric libraries which have some global precision (e.g. as 1/1000th of the model bounding box dimension), Open CASCADE treats tolerances as local properties. As indicated in the diagram in Part1, tolerance is a field of any vertex, edge, or face instance. This approach helps to describe the model with higher accuracy in general. Look at the picture below:

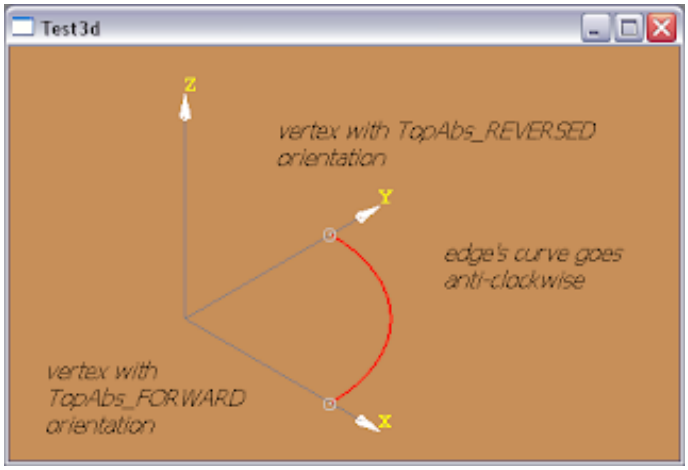


Individual tolerances allow to specify local inaccuracies while leaving the rest of the model well defined. If you have to build your shape bottom up, the best approach is to specify the smallest valid tolerance. By default, it is Precision::Confusion() which is 1e-07.

In general some modeling algorithms are quite sensitive to tolerances, especially if they have to work with a single value to be specified by the user. For instance, importing a model from another 3D geometric kernel (via IGES or STEP formats, or directly from native format of other kernels or CAD systems), involves a global precision specified in a file header. The importer tries to be robust regardless of that value, no matter how coarse it is (e.g. it could that big so that some tiny faces are fully encompassed by its boundaries what would even violate the standard).

Another example is Sewing (stitching topologically disconnected faces into a shell). Gaps between faces can be quite different across the model. Specifying too small tolerance would leave too many disconnected faces, specifying too big upfront would connect too distant faces (maybe even implied too be disconnected by user intent, like tiny holes).

Coming back to vertices, let me mention about their orientation field. It does not have a direct geometric meaning, but by convention vertex with TopAbs_FORWARD orientation must match an edge's end corresponding to the smaller parameter of its curve. Respectively a vertex with TopAbs_REVERSED – to the curve's end with greater parameter. For instance, if you have an edge lying on a circular arc of radius 1 on plane Z=0 starting at point (1, 0, 0) and moving anti-clockwise (if to look in the direction opposite to Z axis) then its forward vertex will have a point (1, 0, 0) and reversed one – (0, 1, 0):



Building a vertex bottom-up BRep_Builder is the tool that actually performs low-level operations.

```
gp_Pnt aPoint (100., 200., 300.);
BRep_Builder aBuilder;
TopoDS_Vertex aVertex;
aBuilder.MakeVertex (aVertex, aPoint, Precision::Confusion());
aVertex.Orientation (TopAbs_REVERSED);
```

There is a convenience class BRepBuilderAPI_MakeVertex which eventually uses BRep_Builder inside. So, if you have to construct topology bottom up make sure you are familiar with BRep_Builder.

Accessing vertex info BRep_Tool is the tool that provides access to geometry of the topological entities. Most its methods are static.

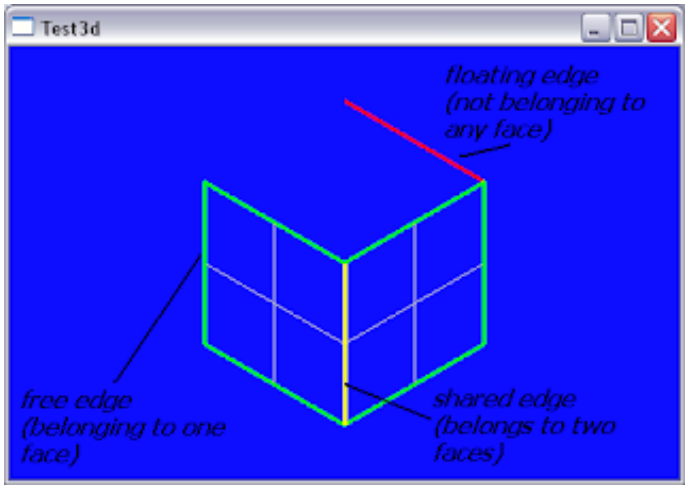
```
Standard_Real aTolerance = BRep_Tool::Tolerance (aVertex);
gp_Pnt aPoint = BRep_Tool::Pnt (aVertex);
```

TOPOLOGY AND GEOMETRY IN OPEN CASCADE. PART 3

BY ROMAN LYGIN - 23:08

OK, let's continue to eat our elephant bit by bit. The next bit is edge. Hope you won't get difficulties with it.

Edge Edge is a topological entity that corresponds to 1D object – a curve. It may designate a face boundary (e.g. one of the twelve edges of a box) or just a 'floating' edge not belonging to a face (imagine an initial contour before constructing a prism or a sweep). Face edges can be shared by two (or more) faces (e.g. in a stamp model they represent connection lines between faces) or can only belong to one face (in a stamp model these are boundary edges). I'm sure you saw all of these types – in default viewer, in wireframe mode, they are displayed in red, yellow and green respectively.



Edge contains several geometric representations (refer to the diagram in Part1): - Curve C(t) in 3D space, encoded as Geom_Curve. This is considered as a primary representation; - Curve(s) P(t) in parametric 2D space of a surface underlying each face the edge belongs to. These are often called pcurves and are encoded as Geom2d_Curve; - Polygonal representation as an array of points in 3D, encoded as Poly_Polygon3D; - Polygonal representation as an array of indexes in array of points of face triangulation, encoded as Poly_PlygonOnTriangulation.

The latter two are tessellation analogues of exact representations with the help of former two.

These representations can be retrieved using already mentioned BRep_Tool, for instance:

`Standard_Real aFirst, aLast, aPFirst, aPLast;` `Handle(Geom_Curve) aCurve3d = BRep_Tool::Curve (anEdge, aFirst, aLast);` `Handle(Geom2d_Curve) aPCurve = BRep_Tool::CurveOnSurface (anEdge, aFace, aPFirst, aPLast);`

The edge must have pcurves on all surfaces, the only exception is planes where pcurves can be computed on the fly.

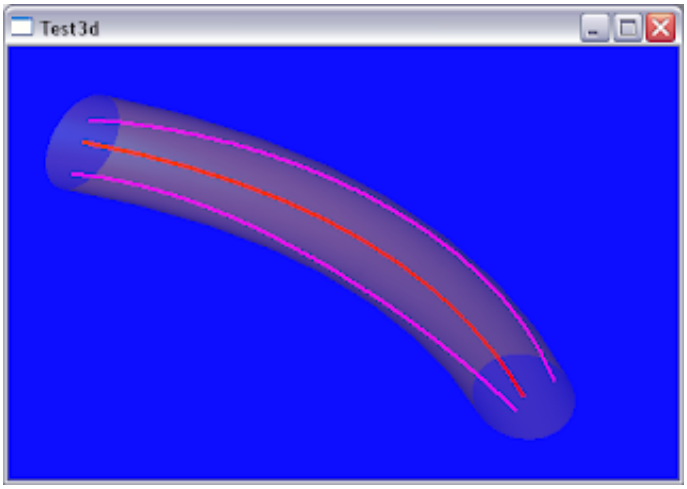
The edge curves must be coherent, i.e. go in one direction. Thus, a point on the edge can be computed using any representation - as C(t), t from [first, last]; S1 (P1x (u), P1y(u)), u from [first1, last1], where Pi – pcurve in parametric space of surface Si

Edge flags

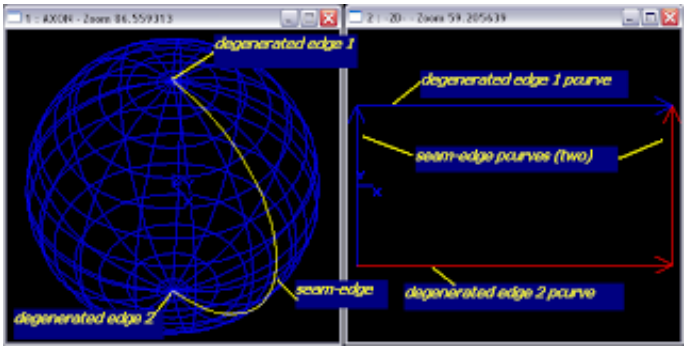
Edge has two special flags: - "same range" (BRep_Tool::SameRange()), which is true when first = first_i and last = last_i, i.e. all geometric representations are within the same range; - "same parameter" (BRep_Tool::SameParameter()), which is true when C(t) = S1(P1x(t), P1y(t)), i.e. any point along the edge corresponds to the same parameter on any of its curves.

Many algorithms assume that they are both set, therefore it is recommended that you ensure that these conditions are respected and flags are set.

Tolerance Edge's tolerance is a maximum deviation between its 3D curve and any other representation. Thus, its geometric meaning is a radius of a pipe that goes along its 3D curve and encompass curves restored from all representations.



Special edge types There are two kinds of edges that are distinct from others. These are: - seam edge – one which is shared by the same face twice (i.e. has 2 pcurves on the same surface) - degenerated edge – one which lies on a surface singularity that corresponds to a single point in 3D space. The sphere contains both of these types. Seam-edge lies on pcurves corresponding to surface U iso-lines with parameters 0 and 2*PI. Degenerated edges lies on North and South poles and correspond to V iso-lines with parameters –PI/2 and PI/2.



Other examples - torus, cylinder, and cone. Torus has two seam-edges – corresponding to its parametric space boundaries; cylinder has a seam-edge. Degenerated edge represents on a cone apex.

To check if the edge is either seam or degenerated, use BRep_Tool::IsClosed(), and BRep_Tool::Degenerated().

Edge orientation Forward edge orientation means that its logical direction matches direction of its curve(s). Reversed orientation means that logical direction is opposite to curve's direction. Therefore, seam-edge always has 2 orientations within a face – one reversed and one forward.

P.S. As usual, many thanks to those who voted and sent comments. Is this series helpful ?

TOPOLOGY AND GEOMETRY IN OPEN CASCADE. PART 4

BY ROMAN LYGIN - 23:14

Though the next topology type after the edge considered in the previous post is a wire, let's jump to the face which is the last topology entity that binds with geometry. We'll consider a wire as well as the rest of topology types in the future posts. Frankly, I was not originally going to speak of them but several folks on the blog asked to.

Face A face is a topology entity that describes a boundary unit of the 3D body. A face is described with its underlying surface and one or more wires. For instance a solid cylinder consists of 3 faces – bottom and top, and lateral. Each of respective underlying surfaces is infinite (Geom_Plane and Geom_CylindricalSurface), while each face bounds its surface with a wire – two of them consists of a single edge lying on Geom_Circle and the lateral face consists of 4 edges – 2 are shared with top and bottom faces, and remaining two represent a seam edge (see previous post on edges), i.e. the face contains it twice in its wire (with different orientations).

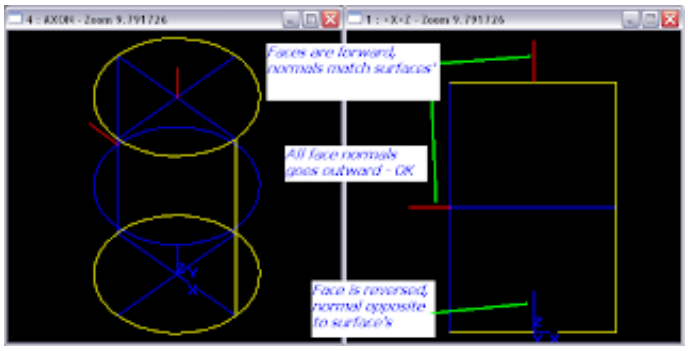
Surface Let's briefly recall what a surface is. If you had a course of mathematical analysis in your higher school you likely remember this by heart, if not you might want to read some articles to educate yourself. First part of this one on wikipedia give simple examples of parametric surfaces.

A surface maps its 2D parametric space {U,V} into a 3D space object (though still two-dimensional). Compare it with molding when a planar steel sheet is transformed into something curved. Parametric space can be bounded or unbounded, or (un)bounded in one direction only. For instance, a plane's parametric space is unbounded, while NURBS is bounded, and a cylindrical surface is bounded in U (from 0 to 2*PI) and is unbounded in V (-infinity, + infinity). Geom_Surface::Value() returns a 3D point (X, Y, Z) from a point in a parametric space (U, V). For instance any location on Earth is described by latitude (V) and longitude (U), but can be viewed as 3D point in the Universe (if Earth center had some origin defined).

Let's recall that an edge must have both a 3D curve and a pcurve in its face surface space. Open CASCADE requires that face boundaries (wires) represent closed contours in 3D and 2D space. Therefore a face cylinder's lateral face is described as we considered above.

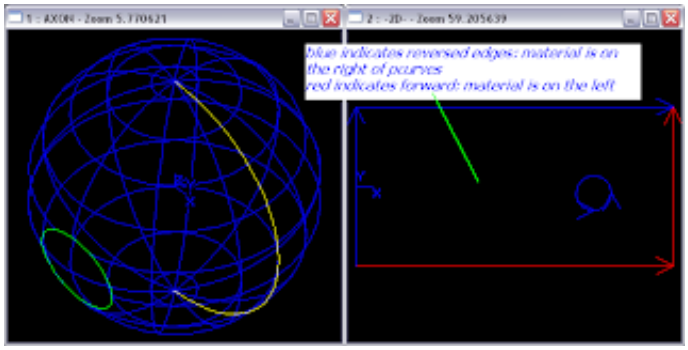
Not all modeling kernels impose such restrictions. I remember a cylinder coming from Unigraphics (via STEP) which lateral face was described with 2 wires, each consisting of a circular edge. We had a few discussions with my colleagues from the Data Exchange team on how to recognize such cases and how to convert them into Open CASCADE valid wire. As a result, Shape Healing's ShapeFix_Face has been extended with FixMissingSeam().

Orientation Face orientation shows how face normal is aligned with its surface normal. If orientation is TopAbs_FORWARD then normals match, if TopAbs_REVERSED then they are opposite to each other. Face normal shows where body material is – it lies 'behind' the face. In a correct solid body all face normals go 'outward' (see below):

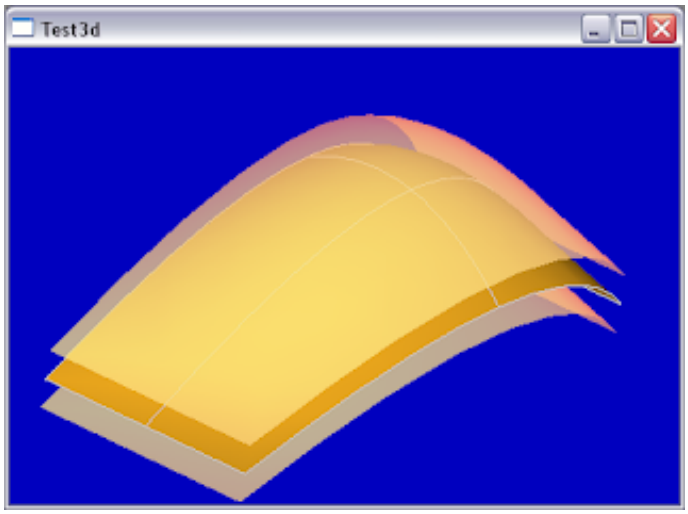


Material on the face is defined by orientation of its edges. The side is defined by a cross product of a surface (not face!) normal and edge derivative. Edge derivative equals its 3D curve derivative if edge is forward and is opposite, if reversed. Perhaps easier understanding can be got if to consider edge pcurves: if an edge is forward then material is on the left, if reversed, material is on the right. This may sound complicated but once

you get it, it will be simpler ;-). We'll speak more of orientation in a dedicated chapter.



Tolerance Geometric meaning of a face tolerance is a thickness of a pie surrounding a surface.



A face tolerance is used by modeling algorithms significantly more rarely than edge's or vertex' and is often retained at default level (Precision::Confusion()). By convention, Open CASCADE requires that the following condition is respected: Face tolerance <= Edge tolerance <= Vertex tolerance, where an Edge lies on a Face, and a Vertex – on an Edge.

Triangulation In addition to underlying surface, a face may contains a tessellated representation (composed of triangles). It is computed, for example, when a face is displayed in shading mode. Visualization algorithms internally call BRepMesh::Mesh() which calculates and adds triangulation for every face.

Additional location Unlike an edge or a vertex, a face has an additional location (TopLoc_Location) which is a member field of BRep_TFace. So, do not forget to take it into account when using underlying surface or triangulation. Stephane has recently pointed this out on a forum.

Creating a face bottom-up and accessing its data Like in the case of a vertex and an edge, BRep_Builder and BRep_Tool is what you need. `BRep_Builder aBuilder;` `TopoDS_Face aFace;` `aBuilder.MakeFace (aFace, aSurface, Precision::Confusion());` `...`

`TopLoc_Location aLocation;` `Handle(Geom_Surface) aSurf = BRep_Tool::Surface (aFace, aLocation);` `gp_Pnt aPnt = aSurf->Value (aU, aV).Transformed (aLocation.Transformation());` `//or to transform a surface at once` `//Handle(Geom_Surface) aSurf = BRep_Tool::Surface (aFace);` `//gp_Pnt aPnt = aSurf->Value (aU, aV);`

`Handle(Poly_Triangulation) aTri = BRep_Tool::Triangulation (aFace, aLocation);` `aPnt = aTri->Nodes.Value (i).Transformed (aLocation.Transformation());`

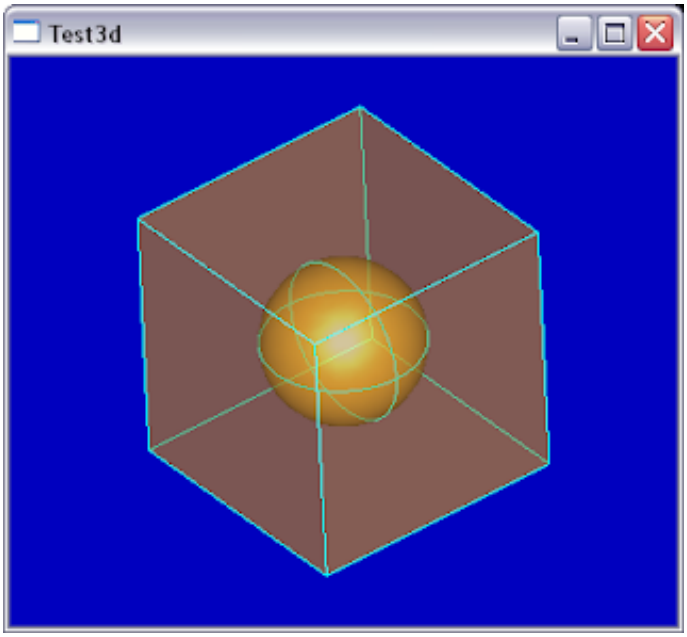
Hope this was not too boring ;-). Just bookmark it and get back when you need it!

TOPOLOGY AND GEOMETRY IN OPEN CASCADE. PART 5

BY ROMAN LYGIN - 13:18

Other topology types So far we considered vertex, edge, and face – those which have connection with geometry. The rest – wire, shell, solid, compsolid and compound – do not connect with geometry directly and are just containers for other topological entities: - wire consists of edge(s); - shell – of face(s); - solid – of shell(s); - compsolid – of solid(s) sharing common face(s); - compound – of any arbitrary type (including compound).

A minor note on solids. A solid is expected to contain a single shell that describes its external boundary. If there are two or more shells, it's considered to be a solid with voids but Open CASCADE can be not too robust to work with such bodies. So beware.



Iteration over children The are two ways to iterate over child subshapes. \1. Direct children can be retrieved using TopoDS_Iterator. The following function will traverse through entire shape structure: `void TraverseShape (const TopoDS_Shape& theShape)` `{` `TopoDS_Iterator anIt (theShape);` `for (; anIt.More(); anIt.Next()) {` `const TopoDS_Shape& aChild = anIt.Value();` `TraverseShape (aChild);` `}` `}` TopoDS_Iterator has two flags specifying whether to take into account location and orientation of a parent shape when extracting a child. If the location flag is on then any child is returned as if it were a standalone shape and placed exactly at its location in 3D space (i.e. the user would see an extracted edge right where it is displayed in the context of its parent wire). If the orientation flag is on, then returned child orientation will be a product of a parent and own child orientation (e.g. two reversed or forward will give forward, reversed and forward in any combination will give reversed). If flags are off, then a child shape is returned with its own location and orientation as stored inside (recall diagram 2 in Part1). By default both flags are on.

\2. Children of a particular subtype If you want to retrieve all edges of your shape you can do the following: `TopExp_Explorer anExp (theShape, TopAbs_EDGE);` `for (; anExp.More(); anExp.Next()) {` `const TopoDS_Edge& anEdge = TopoDS::Edge (anExp.Current());` `//do something with anEdge` `}`

TopExp_Explorer has an additional parameter that specifies which parent type you want to skip. For example, if you want to retrieve only floating edges (i.e. not belonging to any face – refer to Part3), you can do the following: `TopExp_Explorer aFloatingEdgeExp (theShape; TopAbs_EDGE, TopAbs_FACE);`

More on location and orientation As already described in the previous chapters, an individual location of a geometrically bounded entity (vertex, edge, face) defines a displacement relative to its underlying geometry. Location of any topology entity (regardless if it has geometric binding or not) also defines a displacement relative to its children. For instance, if a wire has a location consisting of a translation part along the {0, 0, 10} vector then it just means that all its edges are actually translated along the Z axis by 10 units.

The same works for orientation. Parent orientation affects its children orientation when extracting them from inside the shape. There is one important exception however – which is about edge orientation within a face. If you recall Part4 we discussed there that face material lies on the left of forward edge pcurves and on the right of the reversed edge pcurves. This reversed or forward orientation of an edge must be calculated *excluding* own face orientation. That is, if you need to use edge pcurves you should rather use:

`TopExp_Explorer aFaceExp (myFace.Oriented (TopAbs_FORWARD), TopAbs_EDGE);` `for (; aFaceExp.More(); aFaceExp.Next()) {` `const TopoDS_Edge& anEdge = TopoDS::Edge (aFaceExp.Current());` `}` This exception becomes understandable if you try to understand the details. Let's construct a face bottom-up:

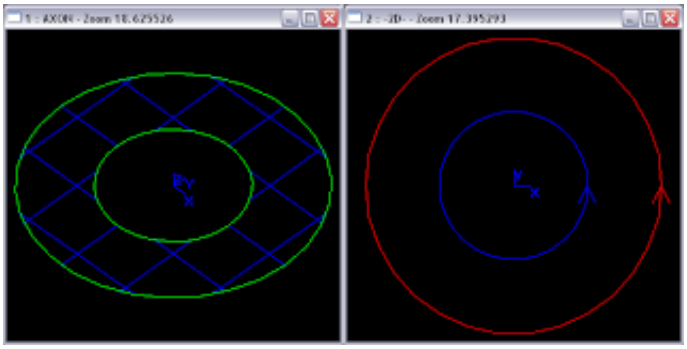
`Handle(Geom_Surface) aSurf = new Geom_Plane (gp::XOY());`

`//anti-clockwise circles if too look from surface normal` `Handle(Geom_Curve) anExtC = new Geom_Circle (gp::XOY(), 10.);` `Handle(Geom_Curve) anIntC = new Geom_Circle (gp::XOY(), 5.);` `TopoDS_Edge anExtE = BRepBuilderAPI_MakeEdge (anExtC);` `TopoDS_Edge anIntE = BRepBuilderAPI_MakeEdge (anExtC);` `TopoDS_Wire anExtW = BRepBuilderAPI_MakeWire (anExtE);` `TopoDS_Wire anIntW = BRepBuilderAPI_MakeWire (anIntE);` `BRep_Builder aB;` `TopoDS_Face aFace;` `aB.MakeFace (aFace, aSurf, Precision::Confusion());` `aB.Update (aFace, aSurf);` `aB.Add (aFace, anExtW);` `aB.Add (aFace, anIntW.Reversed()); //material should lie on the right of the inner wire`

aFace has a forward orientation (default). Let's explore its edges and pcurves. Though we did not explicitly add them, recall (see Part 3) that for planes they can be computed on the fly:

`void TraversePCurves (const TopoDS_Face& theFace)` `{` `TopExp_Explorer anExp (theFace, TopAbs_EDGE);` `for (; anExp.More(); anExp.Next()) {` `const TopoDS_Edge& anEdge = TopoDS::Edge (anExp.Current());` `Standard_Real aF, aL;` `Handle(Geom2d_Curve) aPCurve = BRep_Tool::CurveOnSurface (anEdge, theFace, aF, aL);` `}` `}`

Returned pcurves will be as shown below (material will be on the left of red and on the right of blue).



Everything is correct. Now imagine we reverse the face and explore again:

```
TopoDS_Face aRFace = TopoDS::Face (aFace.Reversed()); TraversePCurves (aRFace);
```

What will we see ? All the edges will be extracted with opposite orientations and respectively their pcurves will mean that material is beyond an external wire and within an internal wire. This is clearly wrong, though original aRFace is perfectly correct. Recall Part4 that a face orientation just shows face logical orientation regarding its underlying surface. In our case aRFace will be just aFace with a normal {0, 0, -1}.

Thus, the only way out is to do the following: `TopExp_Explorer anExp (theFace.Oriented (TopAbs_FORWARD), TopAbs_EDGE);`

This will ensure that edges will show orientation regarding surface (not face!) normal. I made exact same comment in Part 4. Open CASCADE algorithms take care of this particular case, make sure so do you.

Hope orientation is now also a well swallowed bit of an elephant we have been eating in this series. I think we are almost done with it. There are a few tiny bones to pick, and unless you are still hungry we will finish it in the last post to follow. Bon appetite ;-)

TOPOLOGY AND GEOMETRY IN OPEN CASCADE. PART 6

BY ROMAN LYGIN - 15:06

Back references As you likely noticed using OpenCASCADE or analyzing the diagram in Part1, shapes refer to their sub-shapes and not the other way round. This is understandable as the same (sub-)shape can belong to multiple parent shapes. For instance, any shared edge will belong to at least two faces. However it is sometimes needed to trace parent shape back from a child. To do this use TopExp::MapShapesAndAncestors().

```
TopTools_IndexedDataMapOfShapeListOfShape anEFsMap; TopExp::MapShapesAndAncestors (myShape, TopAbs_EDGE, TopAbs_FACE, anEFsMap);
```

The code above fills out a map of parent faces for each edge in myShape. If myShape is a solid box, each edge will map to 2 faces. If you explore the same box into faces and try to fill out edge's ancestors in context of each face, then obviously the map will contain a single face for each edge – that very face you are currently in.

Adaptors Some Open CASCADE algorithms can work on objects representing a curve. However they provide an API that does not accept Geom_Curve but rather Adaptor3d_Curve. For instance, Extrema does so what enables its use in intersection, projection and other algorithms both on geometrical curves (Geom_Curve) and topological edges (TopoDS_Edge). Other examples – calculation of lengths, or surface areas. This approach is known as Adapter pattern. GeomAdaptor3d_Curve subclasses Adaptor3d_Curve to 'adapt' Geom_Curve,

BRepAdaptor_Curve to TopoDS_Edge, BRepAdaptor_CompCurve to TopoDS_Wire. There are similar classes for 2D curves and surfaces.

So you could write the following to measure lengths of a curve and an edge:

```
Handle(Geom_Curve) aCurve = ...; Standard_Real aCurveLen = GCPoints_AbscissaPoints::Length (GeomAdaptor_Curve (aCurve));
```

```
TopoDS_Edge anEdge = ...; Standard_Real anEdgeLen = GCPoints_AbscissaPoints::Length (BRepAdaptor_Curve (anEdge));
```

Conclusion So, this has been a long story about fundamental concepts of Open CASCADE. Hope you are now more familiar with them and understand what geometry and topology are. As a first step, make yourself correctly use the terms – curve or edge, surface or face, point or vertex. This will help you and people reading your questions clearly distinguish if you mean geometry or topology. Once you have started using correct definitions and keeping in mind their distinctions, part of your problems may simply go away.

That's it! Full up with this elephant ;-) ?