Laboratory for Advanced Planning and simulation project

Tutorial on Rapid Prototyping manufacturing lumen vessel by Open Cascade

Fabrizio Murgia, Piero Pili, Gabriella Pusceddu,
Gregorio Franzoni

CRS4, EIP/Geometric Modelling and Monte Carlo Simulations Area

Tutorial on Rapid Prototyping manufacturing lumen vessel by Open Cascade

Fabrizio Murgia, Gabriella Pusceddu, Gregorio Franzoni, Piero Pili
Geometric Modeling Monte Carlo Simulation Area
Energy and Chemical Processes Dept.
CRS4, Italy

12 Giugno 2003

Abstract

This document is the tutorial program for **Splinetor**. Using this tutorial a programmer with basic experience in C++ will be able to use Open Cascade to build a valid Boundary representation starting from the carotid segmented lumen, generated by the segmentation tool. Such a BRep solid can be interactively inspected by the program Sample, the visualization tool.

Splinetor is a package able to create valid solid models in Boundary Representation (BRep) starting from clouds of points extracted by the segmentation tool applied on a data set of Computer Tomography (CT) images.

Splinetor final goal is the threedimensional (3D) printing of Brep realized with a Rapid Prototyping device; therefore **Splinetor** must be able to export the final solid model in STereoLitography format (stl), till now the only input format accepted by Stratasys Rapid Prototyping devices.

Splinetor is based on two *Open Source* libraries: **OpenCascade** ©Matra Datavision for the solid modelling representation and **QT** ©Trolltech as graphics user interface and visualization tool.

Open Cascade is a set of established 3D modeling components from Matra Datavision dedicated to the development of trade-specific technical and scientific applications ranging from mechanical CAD/CAM/CAE to metrology and measuring machines, biomedical software, 3D geological mapping, and optical simulation.

Qt is a C++ toolkit for multiplatform GUI and application development. In addition to the C++ class library, Qt includes tools to make writing applications fast and straightforward.

A short description of the *Open Source* philosophy can be found in appendix A. A short description of the structure of Open Cascade can be found in appendix B. A short description of stl format can be found in appendix C.

This work as been done inside the task activities of the project LAPS (Laboratory for Advanced Planning and Simulation).

Contents

Introduction **4**

1 Splinetor **5**
1.1 Boundary Representation 5
1.1.1 BRep for lumen vessel reconstruction 7
1.2 Splinetor 8
1.2.1 Reading files 8
1.2.1.1 OCAS classes: gp_Pnt 8
Reading file: Code example (1) 9
1.2.2 Filling arrays 9
1.2.2.1 OCAS classes: Collection 9
1.2.2.2 OCAS classes: TColgp_Array1OfPnt 10
Filling arrays: Code example (2) 11
1.2.3 Scaling arrays 11
Scaling arrays: Code example (3) 11
1.2.3.1 OCAS classes: gce_MakeScale2d 12
1.2.3.2 OCAS classes: gp_Trsf2d 12
Scaling arrays: Code example (4) 13
1.2.4 Interpolating curves and surfaces 13
1.2.4.1 B-Spline curves and surfaces 14
1.2.4.2 OCAS classes: Geom_Geometry 15
1.2.4.3 OCAS classes: Geom_Curve 15
1.2.4.4 OCAS classes: Geom_BoundedCurve 16
1.2.4.5 OCAS classes: Geom_BSplineCurve 16
1.2.4.6 OCAS classes: GeomAPI_Interpolate 17
Interpolating curves and surfaces: Code example (5) . . . 18
1.2.5 Connecting contiguous B-Spline Curves 19
1.2.5.1 OCAS classes: Geom_Surface 19
1.2.5.2 OCAS classes: Geom_BoundedSurface 20
1.2.5.3 OCAS classes: Geom_BSplineSurface 20
1.2.5.4 OCAS classes: GeomFill_BSplineCurves 23
Connecting contiguous B-Spline Curves: Code Example (6) 24
1.2.6 Creating shells 24
1.2.6.1 OCAS classes: BRepBuilderAPI_MakeShape . . 25
1.2.6.2 OCAS classes: BRepBuilderAPI_MakeShell . . . 25
1.2.6.3 OCAS classes: BRepBuilderAPI_MakeSolid 26
1.2.6.4 OCAS classes: TopoDS_Shape 28
1.2.6.5 OCAS classes: TopoDS_Shell 28

1.2.6.6 OCAS classes: TopoDS_Solid 28
1.2.6.7 OCAS classes: TCollection_List 28
Creating shells: Code Example (7) 29
1.2.7 Lumen bifurcation 29
1.2.7.1 OCAS classes: Geom2dAPI_InterCurveCurve . . . 30
Lumen bifurcation: Code Example (8) 31
1.2.8 Joining shells 33
1.2.8.1 OCAS classes: BRepOffsetAPI_Sewing 33
1.2.8.2 OCAS classes: BRepTools 34
Joining shells: Code example (9) 34
1.2.9 Exporting tesselled lumen geometry: STL file 34
1.2.9.1 OCAS classes: StlAPI_Writer 35
Exporting tesselled lumen geometry: Code example (10) . 35

2 Sample: Visualization Tool **36**
2.1 Visualizer Sample 36
2.2 About Qt 36
2.2.1 Qt Object Model 37
2.2.2 Object Trees and Object Ownership 37
2.2.3 Signals and Slots 38
2.2.4 Signals 39
2.2.5 Slots 39
2.2.6 Meta Object System 40
2.2.7 Short description 41

Appendices **46**
A Open Source Philosophy 46
B Open Cascade 48
C STL format 51
D Input File Format 54

Bibliography **56**

Introduction

Rapid prototyping (RP) refers to the physical modeling of a design using a special class of machine technology. RP systems quickly produce models and prototype parts from 3D Computer-Aided Design (CAD) model data, Computer Tomography (CT) and Magnetic Resonance Imaging (MRI) scan data and data created from 3D object digitizing systems. Using an additive approach to building shapes layer-by-layer, RP systems join liquid, powder and sheet materials to form physical objects [1].

RP is impacting medicine in several important ways, with the goal of designing, developing and manufacturing medical devices and instrumentation [2].

The great advantage of RP technologies is the precise reproduction of objects from a 3D medical image data-set as a physical model that can be looked at and touched by the surgeon [3].

More details about RP and its applications in medicine can be found in [4], [5].

In our work we use the results of the segmentation tool to extract geometric data from a set of CT images of a human carotid explanted from a cadaver of a 32 years old man. Geometric data were the input for splinetor, the package written for the reconstruction of the solid geometry of the artery; the output of splinetor is an .stl file. The STL file is the input for the Rapid Prototyping device, a Stratasys FDM 2000 (©Stratasys) using Fused Deposition Modelling (FDM) technique. Further details about LAPS entire pipeline can be found in [6], details about the work progress can be found in [7] and [8].

This document is the tutorial program for **Splinetor**. Using this tutorial a programmer with basic experience in C++ will be able to use Open Cascade to build a valid Boundary representation starting from the carotid segmented lumen, generated by the segmentation tool. Such a BRep solid can be interactively inspected by the program Sample, the visualization tool.

Chapter 1

Splinetor

This chapter shows how to use Open-Source software for geometric reconstruction, OpenCascade (OCAS) library, to build a valid Boundary Representation (BRep) of a human carotid lumen. **Splinetor** module is inserted in the lumen reconstruction and manufacturing software architecture shown in Figure 1.1.

1.1 Boundary Representation

B-Rep models represent a solid indirectly by a representation of its bounding surface. In BRep an object is represented by a combination of entities of type *point*, *edge*, *face*, and volume (Figure 1.2).

Faces, *edges*, and *vertices* entities, and the related geometric information form the basic components of B-Rep models. The geometric information contains the face and edge equations (or data to compute them), and vertex coordinates. The topology description contains the information on the relationship between the components, i.e. how faces, edges and vertices of the model are connected together.

Because B-Rep includes such topological information, a solid is represented as a closed space in 3-D space.

The boundary of a solid separates points inside from points outside of the solid. An element of B-rep consists of geometric data of the element, an identification of the code of element categories, and its relationship to other elements [11].

B-rep models can represent a wide class of objects but the data structure is complex, and can require a large memory space for a complex object [12].

Normally, a face is a bounded region of a planar, quadratic, toroidal, or sculptured surface. The bounded region of the surface that forms the face is represented by a closed curve that lies on the surface. A face can have several bounding curves to represent holes in a solid. The bounding curves of faces are represented by edges. The portion of the curve that forms the edge is represented by two vertices.

A B-Rep model to be valid, has to fulfill the following conditions: the set of faces forms a complete skin of the solid with no missing parts, and faces must be closed, orientables, bounded, connected, and do not intersect each other except at common vertices or edges [13]. Furthermore, the boundaries of faces do

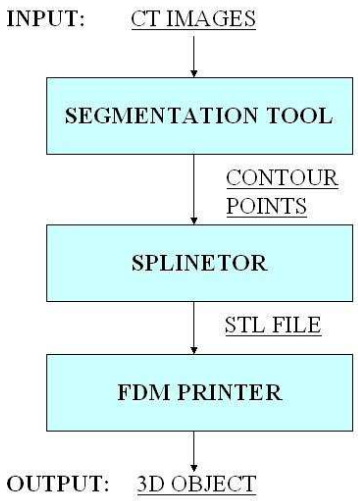

Figure 1.1: The pipeline of lumen reconstruction and manufacturing project. The segmentation tool can be either the software written during VIVA [9] project or the segmentation tool [10]. 3D printing of the object output from splinetor is realized with FDM technique [2].

not intersect themselves. These conditions disallow self-intersecting and open objects [14].

Boundary representation can be divided in three classes: facetted, elementary, and advanced B-Rep. In facetted B-Rep, a solid is bounded by planar surfaces. Only points, planes and planar polygons are necessary and are implicitly represented by their vertex points. The surfaces included in elementary B-Rep are planar, quadric, and toroidal surfaces. The bounding curves of the faces are lines or conics. In advanced B-Rep, the surfaces includes also B-Spline surfaces in addition to elementary B-Rep. The bounding curves are B-Spline curves.

In facetted B-Rep, all edges are straight line segments. Therefore faces can be represented as polygons and each polygon as a set of coordinate values x, y and z. The data structure in this case is simple and easy to implement. Facetted approximation of more sophisticated B-Rep models are normally used for generation of graphical output.

The description of faces can vary. For example, a planar face can be represented in many different ways: using an analytical equation, a parametric equation, a normal vector and a point on the surface, etc. In addition, a closed 3D boundary which lies on the surface is needed to define the face. In conventional CAD systems, the use of faces is usually restricted to the quadrics like cones, cylinders, spheres, etc. Modern CAD systems can use a variety of functions to describe sculptured (or freeform) faces which cannot be represented by simple analytical mathematical functions. We use one of the most common, the B-Spline representation, a category of surfaces employing parametric polynomials.

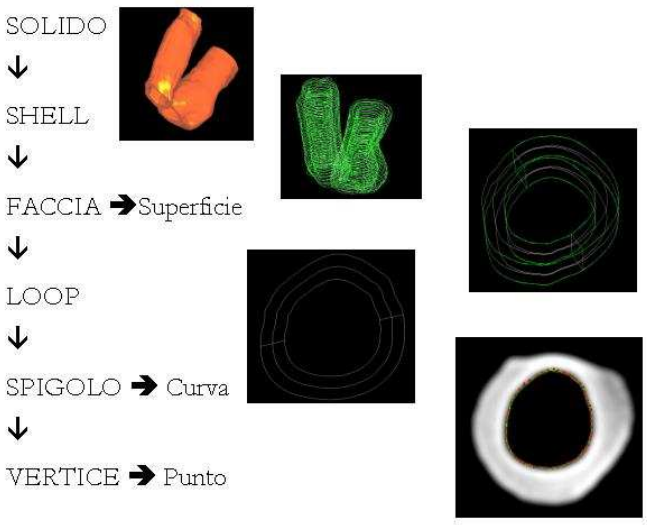

Figure 1.2: The structure of a valid Boundary Representation, starting from clouds of points defining single contours, we can use B-Spline curves to reconstruct the contour of each CT (Computer Tomography) slice, thereafter we can join contiguous slices with B-Spline surfaces, then sewing all the surfaces we obtain the solid model of the vessel.

1.1.1 BRep for lumen vessel reconstruction

In order to perform numerical analysis of an object, we must have its geometric description. The complexity of human body is such that anatomical features makes their mathematical description extremely difficult. In the case of lumen vessel reconstruction the problem is a little bit simpler because the topology of the geometric model does not present too much changes (it changes just in the bifurcation region). Also the lumen geometry is quite smooth. Even though the presence of these good features the geometric reconstruction and the automatic manufacturing is a difficult problem The segmentation tool allows the extraction of the geometry of the structures under consideration giving as output a set of points describing the object in 3-D space as a set of planar contours (Figure 1.1).

A visual scheme of the reconstruction procedure is shown in Figure 1.2. The process starts from clouds of points extracted from segmentation of CT (Computer Tomography) images, then passes through the points interpolation to obtain BSpline curves and through the curves composition to obtain BSpline surfaces, and finishes obtaining a valid Brep of the vessel.

In next section we will describe the reconstruction steps, giving some details about the classes of the software Open Cascade (OCAS) [15] used during splinetor developement. Open Cascade is a set of established 3D modeling components from Matra Datavision dedicated to the development of trade-specific technical and scientific applications ranging from mechanical CAD/CAM/CAE to metrology and measuring machines, biomedical software, 3D geological map-

ping, and optical simulation.

1.2 Splinetor

Splinetor program based on OCAS libraries, is used to reconstruct valid 3D models in Boundary Representation, starting from arrays of points. These arrays of points are obtained after a segmentation process on CT images of an autoptic human carotid. The entire process ends with the 3D printing of reconstructed solid models.

Splinetor is composed by the steps illustrated in Figure 1.3:

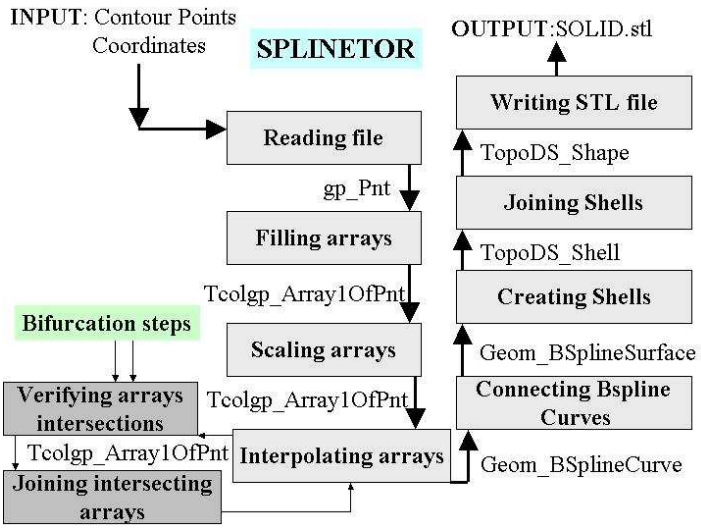

Figure 1.3: The steps composing splinetor program.

Each module is descripted in a subsection. Each OC class and method used is shown. An example of thecode implementation is included.

1.2.1 Reading files

This module reads a *file.dat*, the file containing cartesian coordinates of contour points. This file is obtained as output of segmentation step. Inside the input file besides the cartesian coordinates we find also the number of branches composing the vessel structure, the number of slices composing each branch and the number of points composing each slice. The data format of the file to be read as input is specified on Appendix D. The cartesian coordinates read become arguments of the class gp_Pnt, the class representing points in OCAS. A detailed description of gp_Pnt class can be found on section 1.2.1.1, with an example of our code.

1.2.1.1 OCAS classes: gp_Pnt

Purpose: Describes a 3D Cartesian point.

constructors:

gp_Pnt() Constructs an undefined point

gp_Pnt (const gp_XYZ& Coord) Constructs a point from the coordinate triple Coord

gp_Pnt (const Standard_Real Xp, const Standard_Real Yp, const Standard_Real Zp) Constructs a point with three Cartesian coordinates (Xp, Yp, Zp)

Standard_Real Distance (const gp_Pnt& Other) const Computes the distance between this point and the point Other

Standard_Real SquareDistance (const gp_Pnt& Other) const Computes the square distance, between this point and the point Other

void SetX (const Standard_Real X) Assigns the given value to the X coordinate of this point

void SetY (const Standard_Real Y) Assigns the given value to the Y coordinate of this point

void SetZ (const Standard_Real Z) Assigns the given value to the Z coordinate of this point

Reading file: Code example (1)

```
LogFile= input53mm286dx
char file[30]= "input33mm286dx";
logFile = fopen( file, "r" );
fscanf(logFile,"%f %d %s %s %s %s %s", &z0,&npnt, a, a, a, a, a);
for(n=1;n<=npnt;n++) //For each slide
     fscanf(logFile,"%f %f ", &x0, &y0);
gp_Pnt point(x0,y0,z0);
```

1.2.2 Filling arrays

After the acquisition of data from *file.dat* and the creation of gp_Pnt, we put them in ordered arrays with the class **TColgp_Array1OfPnt**. Each array contains a number of points equal to npnt, with the same z coordinate. Basic features of such a class will be illustrated in the section 1.2.2.2 with an example of our code, and characteristics of abstract mother class **collection** will be in section 1.2.2.1;

1.2.2.1 OCAS classes: Collection

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

Collections classes are generic, that is, they can hold a variety of objects which do not necessarily inherit from a unique root class. When you need to use a collection of a given type of object you must instantiate it for this specific

type of element. Once this declaration is compiled all the functions available on the generic collection are available on your instantiated class.

Note however:

- Each collection directly used as an argument in a **Open CASCADE** public syntax is instantiated in a **Open CASCADE** component.
- The **TColStd** package (Collections of **Standard Objects** component) provides numerous instantiations of these generic collections with objects from the package Standard or from the Strings component.

The Collections component provides a wide range of generic collections:

- *Arrays* are generally used for a quick access to the item, however an array is a fixed sized aggregate.
- *Sequences* are variable sized structures, they avoid the use of large and quasi-empty arrays. But a sequence item is longer to access than an array item: only an exploration in sequence is performant (but sequences are not adapted for numerous explorations).
- *Arrays and sequences* are commonly used as data structures for more complex objects.
- On the other hand *maps* are dynamic structures where the size is constantly adapted to the number of inserted items and the access time for an item is performant. Maps structures are commonly used in cases of numerous explorations: they are typically internal data structures for complex algorithms. *Sets* generate the same results as maps but computation time is considerable.
- *Lists, queues and stacks* are minor structures similar to sequences but with other exploration algorithms.

Most collections follow value semantics: their instances are the actual collections, not handles to a collection. Only arrays, sequences and sets may also be manipulated by handle, and therefore shared.

1.2.2.2 OCAS classes: TColgp_Array1OfPnt

Is a class that inherits from the generic class TCollection_Array1, instantiating it with gp_Pnt.

Purpose: to instantiate unidimensional arrays similar to C arrays, that is, of fixed size, but dynamically dimensioned at construction time. As with a C array, the access time for an Array1 indexed item is constant and is independent of the array size. Arrays are commonly used as elementary data structures for more complex objects. Array1 is a generic class which depends on Item, the type of element in the array.

TCollection_Array1 (const Standard_Integer lower, const Standard_Integer upper) Constructs an array of the lower and upper bounds. The size of the array is upper - lower + 1.

TCollection_HArray1 (const Standard_Integer lower, const Standard-_Integer upper) *HArray1* objects are handles to arrays.

void Init(const Item & Value) This function assigns an initial value to all the items of the array

void SetValue (const Standard_Integer Index, const Item & Value) This function assigns a value to the Indexth item of this array.

Standard_Integer Length() const This function returns the number of items on this array.

Standard_Integer Lower() const Returns the index of the lower bound of this array.

Standard_Integer Upper() const Returns the index of the upper bound of this array.

Item& operator() (const Standard_Integer Index) Returns the value of the item of index Index in this array, or returns a modifiable reference to the item of index Index in this array, in order to assign a new value to this item.

Filling arrays: Code example (2)

```
for(n=1;n<=npnt;n++){
   TColgp_Array1OfPnt mieipnt(1,npnt+1);
if(n==1)
   mieipnt.Init(point);
mieipnt.SetValue(n,point);}
```

1.2.3 Scaling arrays

First of all, we have to calculate the center point of the array, it will be the reference point for the scale transformation as shown in the code example (3) below:

Scaling arrays: Code example (3)

```
for(n=1;n<=npnt;n++){
   xa[n]=x0;
   ya[n]=y0;
   xam=xam+xa[n];
   yam=yam+ya[n];}
xm=xam/npnt;
ym=yam/npnt;
zm=z0;
gp_Pnt Med(xm,ym,zm);
```

The operation of scaling arrays is useful to obtain the arrays of points necessary to build a solid parallel to the main solid. With this step is possible to give a thickness to the solid to be printed.

We choose to work with plane slices, so the scale transformation will be 2D on plane x-y for each point on each slice, i.e. the transformation affects just x and y coordinates. We give a scale factor of 1.5 as argument of the class **gce_MakeScale2d**; the scale operation will be applied to each point by the method *Transforms(...)* of the class **gp_Trsf2d**. Basic characteristics of these classes with an example of our code will be illustrated in the sections 1.2.3.1 and 1.2.3.2.

1.2.3.1 OCAS classes: gce_MakeScale2d

Purpose:

Implements an elementary construction algorithm for a scaling transformation in 2D space. The result is a **gp_Trsf2d** transformation.

A MakeScale2d object provides the framework for:

- defining the construction of the transformation,
- implementing the construction algorithm, and
- consulting the result.

constructor:

gce_MakeScale2d (const gp_Pnt2d& Point, const Standard_Real Scale)

Constructs a scaling transformation with:

- Point as the center of the transformation, and
- Scale as the scale factor.

const gp_Trsf2d& Value() const Returns the constructed transformation.

1.2.3.2 OCAS classes: gp_Trsf2d

Purpose:

Describes a group of geometric transformations performed in the plane (2D space). These transformations include:

- translations,
- rotations,
- scaling transformations,
- reflections with respect to a point or line, and
- transformations obtained from combinations of the above.

These transformations respect the nature of geometric objects. For example, a circle is transformed into a circle by a gp_Trsf2d.

The transformation of point P with coordinates (x,y,z) is the point P' with coordinates (x',y',z') where:

$x' = a11.x + a12.y + a13$

$y' = a21.x + a22.y + a23$

We could also write the above as:

$P' = V * P + T$

where:

- V, known as the vectorial part of the transformation, is the 2 * 2 matrix:

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$$

- T, known as the translation part of the transformation, is the vector:

$$\begin{pmatrix} a13 \\ a23 \end{pmatrix}$$

gp_Trsf2d() Constructs an Identity transformation.

void Transforms (Standard_Real& X, Standard_Real& Y) const

Applies this transformation to the coordinates:

- X and Y.

Modifies Coord or X and Y.

Scaling arrays: Code example (4)

```
for(i=1;1<=npnt;i++){
   gp_Pnt2d med2d(Med.X(),Med.Y());
   gce_MakeScale2d sca2d (med2d,1.5);
   gp_Trsf2d tra2d = sca2d.Value();
   point = mieipnt(i);
   tra2d.Transforms(point.X(),point.Y());
   gp_Pnt pundopo (point.X(), point.Y(), point.Z());
}
```

1.2.4 Interpolating curves and surfaces

The procedure of interpolation of points of singles arrays (real alone in Figure. 1.4 and real with scaled in Figure. 1.5) is realized with one B-Spline curve for each slice.

We could interpolate the points with other curves, for instance Bezier Curves, but starting from the point of view that Bezier Curves are special cases of B-spline curves we chose B-Spline. B-spline curves satisfy all important properties that Bezier curves have; B-spline curves provide more control flexibility, we can use lower degree curves and still maintain a large number of control points. We can change the position of a control point without globally changing the shape of the whole curve (local modification property). Since B-spline curves satisfy the strong convex hull property, they have a finer shape control. More details about B-Spline theory can be found in section 1.2.4.1. In the subsequent section, i.e. section 1.2.4.2, section 1.2.4.3, section 1.2.4.4, section 1.2.4.5, section 1.2.4.6, we will describe the OCAS classes for the interpolation of clouds of points that we used in our code with an example of the code.

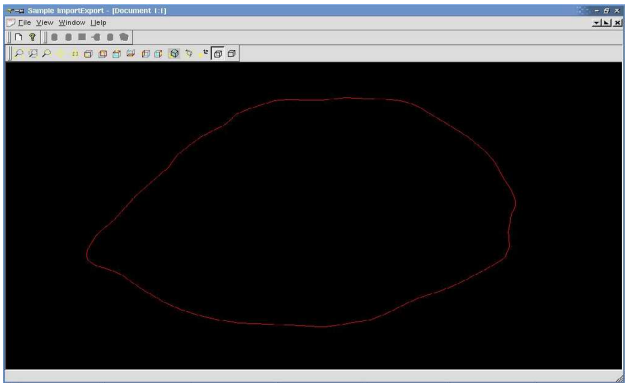

Figure 1.4: Points interpolation of a single array.

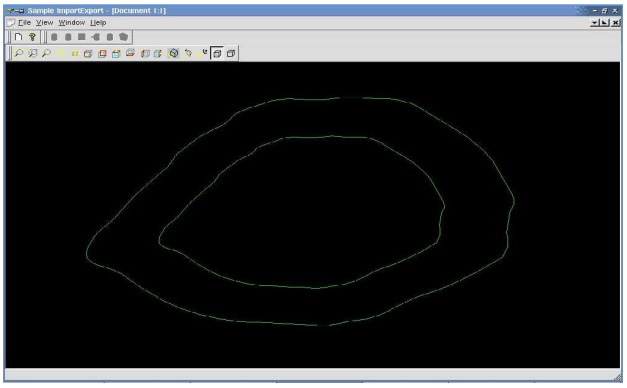

Figure 1.5: Points interpolation of a real array and a scaled array.

1.2.4.1 B-Spline curves and surfaces

The B-Spline surface is a collection of B-Spline curves, i.e. the tensor product of two curves defined by two parameters. The surface is defined as the set of points obtained by evaluating equation 1 for all parameter values of u and v between some u_{min} and u_{max}, and v_{min} and v_{max}:

$$S(u,v) = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \sum_{i=1}^{n} \sum_{j=1}^{m} N_i^k(u) N_j^l(v) P_{i,j}$$

where the **k,l** are the orders of the B-Spline surface in both directions and $P_{i,j}$ is the array of n x m control points $\mathbf{x}_{i,j}$,$\mathbf{y}_{i,j}$,$\mathbf{z}_{i,j}$. The term $N_i^k(u)$ represents the polynomial B-spline basis functions of degree **k-1** in **u** parameter direction, and the basis functions of degree **l-1** in **v** direction [13].

A user creates a B-Spline surface by defining the order and drawing only the control points. The shape of the surface is modified by moving one or several control points.

One property of a B-Spline surface is local control which means that when one control point is moved only a part of the surface is affected. The control

points are the actual shape descriptors of the curve. A control point can be either interpolating, i.e. the surface intersects the control point, or approximating where the surface does not intersect the points. The orders of the surface also describes the minimum array of control points required to define it. If more points are used, each additional point row stands for a new patch. A very important feature of B-Splines is the convex hull property. This means that the surface is always included in the convex hull of its control points. The convex hull is like a skin covered around the control points. The convex hull property is used to compute B-Spline bounding boxes and preliminary intersection computations

Since the complexity of the construction of the B-Rep models, it is not trivial for a designer to build correct B-Rep models directly. The designer needs a sufficient collection of more convenient and efficient solid description methods. For a B-Rep modeller, it is not easy to implement a textual user interface.

1.2.4.2 OCAS classes: Geom_Geometry

Purpose: The abstract class **Geometry** for 3D space is the root class of all geometric objects from the **Geom** package. It describes the common behavior of these objects when:

* applying geometric transformations to objects

* constructing objects by geometric transformation (including copying).

Only transformations which do not modify the nature of the geometry can be applied to **Geom** objects: this is the case with translations, rotations, symmetries and scales; this is also the case with **gp_Trsf** composite transformations which are used to define the geometric transformations applied using the **Transform** or **Transformed** functions. **Geometry** defines the "prototype" of the abstract method Transform which is defined for each concrete type of derived object. All other transformations are implemented using the Transform method.

1.2.4.3 OCAS classes: Geom_Curve

Purpose: The abstract class **Curve** describes the common behavior of curves in 3D space. The **Geom** package provides numerous concrete classes of derived curves, including lines, circles, conics, Bezier or BSpline curves, etc.

The main characteristic of these curves is that they are parameterized. The **Geom_Curve** class shows:

- how to work with the parametric equation of a curve in order to calculate the point of parameter u, together with the vector tangent and the derivative vectors of order 2, 3,..., N at this point;

- how to obtain general information about the curve (for example, level of continuity, closed characteristics, periodicity, bounds of the parameter field);

- how the parameter changes when a geometric transformation is applied to the curve or when the orientation of the curve is inverted.

All curves must have a geometric continuity: a curve is at least "C0". Generally, this property is checked at the time of construction or when

the curve is edited. Where this is not the case, the documentation states so explicitly.

The **Geom** package does not prevent the construction of curves with null length or curves which self-intersect.

1.2.4.4 OCAS classes: Geom_BoundedCurve

Purpose: The abstract class **BoundedCurve** describes the common behavior of bounded curves in 3D space. A bounded curve is limited by two finite values of the parameter, termed respectively "first parameter" and "last parameter". The "first parameter" gives the "start point" of the bounded curve, and the "last parameter" gives the "end point" of the bounded curve.

The length of a bounded curve is finite.

The **Geom** package provides three concrete classes of bounded curves:

- two frequently used mathematical formulations of complex curves:
  - **Geom_BezierCurve**
  - **Geom_BSplineCurve**
- **Geom_TrimmedCurve** to trim a curve, i.e. to only take part of the curve limited by two values of the parameter of the basis curve.

1.2.4.5 OCAS classes: Geom_BSplineCurve

Purpose: Describes a BSpline curve.

A BSpline curve can be:

- uniform or non-uniform,
- rational or non-rational,
- periodic or non-periodic.

A BSpline curve is defined by :

- its degree; the degree for a **Geom_BSplineCurve** is limited to a value (25) which is defined and controlled by the system. This value is returned by the function MaxDegree;
- its periodic or non-periodic nature;
- a table of poles (also called control points), with their associated weights if the BSpline curve is rational. The poles of the curve are "control points" used to deform the curve. If the curve is non-periodic, the first pole is the start point of the curve, and the last pole is the end point of the curve. The segment which joins the first pole to the second pole is the tangent to the curve at its start point, and the segment which joins the last pole to the second-from-last pole is the tangent to the curve at its end point. If the curve is periodic, these geometric properties are not verified. It is more difficult to give a geometric signification to the weights but are useful for providing exact representations of the arcs of a circle or

ellipse. Moreover, if the weights of all the poles are equal, the curve has a polynomial equation; it is therefore a non-rational curve.

- a table of knots with their multiplicities. For a **Geom_BSplineCurve**, the table of knots is an increasing sequence of reals without repetition; the multiplicities define the repetition of the knots. A BSpline curve is a piecewise polynomial or rational curve. The knots are the parameters of junction points between two pieces. The multiplicity Mult(i) of the knot Knot(i) of the BSpline curve is related to the degree of continuity of the curve at the knot Knot(i), which is equal to Degree - Mult(i) where Degree is the degree of the BSpline curve.

If the knots are regularly spaced (i.e. the difference between two consecutive knots is a constant), three specific and frequently used cases of knot distribution can be identified:

- "uniform" if all multiplicities are equal to 1,

- "quasi-uniform" if all multiplicities are equal to 1, except the first and the last knot which have a multiplicity of Degree + 1, where Degree is the degree of the BSpline curve,

- "Piecewise Bezier" if all multiplicities are equal to Degree except the first and last knot which have a multiplicity of Degree + 1, where Degree is the degree of the BSpline curve. A curve of this type is a concatenation of arcs of Bezier curves.

If the BSpline curve is not periodic:

- the bounds of the Poles and Weights tables are 1 and NbPoles, where NbPoles is the number of poles of the BSpline curve,

- the bounds of the Knots and Multiplicities tables are 1 and NbKnots, where NbKnots is the number of knots of the BSpline curve.

If the BSpline curve is periodic, and if there are k periodic knots and p periodic poles, the period is:

-period = Knot(k + 1) - Knot(1)

and the poles and knots tables can be considered as infinite tables, verifying:

- Knot(i+k) = Knot(i) + period

- Pole(i+p) = Pole(i)

Data structures of a periodic BSpline curve are more complex than those of a non-periodic one.

In this class, weight value is considered to be zero if the weight is less than or equal to gp::Resolution().

1.2.4.6 OCAS classes: GeomAPI_Interpolate

Purpose:

Describes functions for building a constrained 3D BSpline curve.

The curve is defined by a table of points through which it passes, and if required:

- by a parallel table of reals which gives the value of the parameter of each point through which the resulting BSpline curve passes, and

- by vectors tangential to these points.

An **Interpolate** object provides the framework for:

- defining the constraints of the BSpline curve,

- implementing the interpolation algorithm, and

- consulting the results.

GeomAPI_Interpolate (const Handle(TColgp_HArray1OfPnt)& Points, const Standard_Boolean PeriodicFlag, const Standard_Real Tolerance)

Initializes an algorithm for constructing a constrained BSpline curve:

- passing through the points of the table Points, or

If PeriodicFlag is true, the constrained BSpline curve will be periodic and closed. In this case, the junction point is the first point of the table Points.

The tolerance value Tolerance is used to check that:

- points are not too close to each other, or

- tangential vectors (defined using the function Load) are not too small.

The resulting BSpline curve will be *"C2"* continuous, except where a tangency constraint is defined on a point through which the curve passes (by using the **Load** function). In this case, it will be only *"C1"* continuous.

Once all the constraints are defined, use the function **Perform** to compute the curve.

- There must be at least 2 points in the table Points.

- If PeriodicFlag is false, there must be as many parameters in the array Parameters as there are points in the array Points.

- If PeriodicFlag is true, there must be one more parameter in the table Parameters : this is used to give the parameter on the resulting BSpline curve of the junction point of the curve (which is also the first point of the table Points).

void Perform() Computes the constrained BSpline curve.

Standard_Boolean IsDone() const Returns true if the constrained BSpline curve is successfully constructed.

const Handle(Geom_BSplineCurve)& Curve() const Returns the computed BSpline curve.

Interpolating curves and surfaces: Code example (5)

```
GeomAPI_Interpolate interpolscaled =
        GeomAPI_Interpolate(mieipntscaled, Standard_False,1.0e-3);
interpolscaled.Perform();
```

```
GeomAPI_Interpolate interpol =
        GeomAPI_Interpolate(mieipnt,Standard_False,1.0e-3);
interpol.Perform();

Handle(Geom_BSplineCurve) baseOn, baseOff;
if(interpolscala.IsDone()){// if interpolation curve exists
     if(interpol.IsDone()){
          baseOn=interpol.Curve();
          baseOff=interpolscaled.Curve();}
```

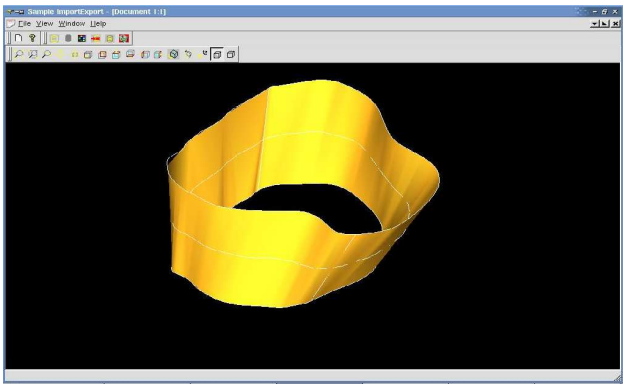

Figure 1.6: Junction shell of 2 contiguous B-Spline curves.

1.2.5 Connecting contiguous B-Spline Curves

For the connection of B-Spline curves obtained during the precedent step we use B-Spline surfaces, surface properties are similar to curves properties and are described in section 1. In sections 1.2.5.1, 1.2.5.2, 1.2.5.3, 1.2.5.4, we give an OCAS classes description, illustrating constructors, methods and possible arguments choices; we also give an example of the code.

A visualization of the results obtained connecting two contiguous B-Spline curves using B-Spline surfaces in Figure 1.6 and Figure 1.7.

1.2.5.1 OCAS classes: Geom_Surface

Purpose:

Describes the common behavior of surfaces in 3D space. The Geom package provides many implementations of concrete derived surfaces, such as planes, cylinders, cones, spheres and tori, surfaces of linear extrusion, surfaces of revolution, Bezier and BSpline surfaces, and so on.

The key characteristic of these surfaces is that they are parameterized.

Geom_Surface demonstrates:

- how to work with the parametric equation of a surface to compute the point of parameters (u, v), and, at this point, the 1st, 2nd ... Nth derivative,

- how to find global information about a surface in each parametric direction (for example, level of continuity, whether the surface is closed, its periodicity, the bounds of the parameters and so on),

- how the parameters change when geometric transformations are applied to the surface, or the orientation is modified.

Note that all surfaces must have a geometric continuity, and any surface is at least *"C0"*. Generally, continuity is checked at construction time or when the curve is edited. Where this is not the case, the documentation makes this explicit.

The Geom package does not prevent the construction of surfaces with null areas, or surfaces which self-intersect.

1.2.5.2 OCAS classes: Geom_BoundedSurface

Purpose: The root class for bounded surfaces in 3D space. A bounded surface is defined by a rectangle in its 2D parametric space, i.e.

- its u parameter, which ranges between two finite values *u0* and *u1*, referred to as "First u parameter" and "Last u parameter" respectively, and

- its v parameter, which ranges between two finite values *v0* and *v1*, referred to as "First v parameter" and the "Last v parameter" respectively.

The surface is limited by four curves which are the boundaries of the surface:

- its u0 and u1 isoparametric curves in the u parametric direction

- its v0 and v1 isoparametric curves in the v parametric direction.

A bounded surface is finite.

The common behavior of all bounded surfaces is described by the **Geom_Surface** class.

The Geom package provides three concrete implementations of bounded surfaces:

- **Geom_BezierSurface**

- **Geom_BSplineSurface**

- **Geom_RectangularTrimmedSurface**

The first two of these implement well known mathematical definitions of complex surfaces, the third trims a surface using four isoparametric curves, i.e. it limits the variation of its parameters to a rectangle in 2D parametric space.

1.2.5.3 OCAS classes: Geom_BSplineSurface

Purpose: Describes a BSpline surface.

In each parametric direction, a BSpline surface can be:

- uniform or non-uniform,

- rational or non-rational,

- periodic or non-periodic.

A BSpline surface is defined by:

- its degrees, in the u and v parametric directions,

- its periodic characteristic, in the u and v parametric directions,

- a table of poles, also called control points (together with the associated weights if the surface is rational),

- a table of knots, together with the associated multiplicities.

The degree of a **Geom_BSplineSurface** is limited to a value (25) which is defined and controlled by the system. This value is returned by the function **MaxDegree**.

Poles and Weights

Poles and Weights are manipulated using two associative double arrays:

- the poles table, which is a double array of **gp_Pnt** points,

- the weights table, which is a double array of reals.

The bounds of the poles and weights arrays are:

- 1 and NbUPoles for the row bounds (provided that the BSpline surface is not periodic in the u parametric direction), where NbUPoles is the number of poles of the surface in the u parametric direction, and

- 1 and NbVPoles for the column bounds (provided that the BSpline surface is not periodic in the v parametric direction), where NbVPoles is the number of poles of the surface in the v parametric direction.

The poles of the surface are the points used to shape and reshape the surface. They comprise a rectangular network.

If the surface is not periodic:

- The points (1, 1), (NbUPoles, 1), (1, NbVPoles), and (NbUPoles, NbVPoles) are the four parametric "corners" of the surface.

- The first column of poles and the last column of poles define two BSpline curves which delimit the surface in the v parametric direction. These are the v isoparametric curves corresponding to the two bounds of the v parameter.

- The first row of poles and the last row of poles define two BSpline curves which delimit the surface in the u parametric direction. These are the u isoparametric curves corresponding to the two bounds of the u parameter.

If the surface is periodic, these geometric properties are not verified.

It is more difficult to define a geometrical significance for the weights. However they are useful for representing a quadric surface precisely. Moreover, if the weights of all the poles are equal, the surface has a polynomial equation, and hence is a "non-rational surface".

The non-rational surface is a special, but frequently used, case, where all poles have identical weights. The weights are defined and used only in

the case of a rational surface. The rational characteristic is defined in each parametric direction. A surface can be rational in the u parametric direction, and non-rational in the v parametric direction.

Knots and Multiplicities

For a **Geom_BSplineSurface** the table of knots is made up of two increasing sequences of reals, without repetition, one for each parametric direction. The multiplicities define the repetition of the knots.

A BSpline surface comprises multiple contiguous patches, which are themselves polynomial or rational surfaces. The knots are the parameters of the isoparametric curves which limit these contiguous patches. The multiplicity of a knot on a BSpline surface (in a given parametric direction) is related to the degree of continuity of the surface at that knot in that parametric direction:

Degree of continuity at

$knot(i) = Degree - Multi(i)$

where:

- Degree is the degree of the BSpline surface in the given parametric direction, and

- Multi(i) is the multiplicity of knot number i in the given parametric direction.

There are some special cases, where the knots are regularly spaced in one parametric direction (i.e. the difference between two consecutive knots is a constant).

- *"Uniform"*: all the multiplicities are equal to 1.

- *"Quasi-uniform"*: all the multiplicities are equal to 1, except for the first and last knots in this parametric direction, and these are equal to Degree + 1.

- *"Piecewise Bezier"*: all the multiplicities are equal to Degree except for the first and last knots, which are equal to Degree + 1. This surface is a concatenation of Bezier patches in the given parametric direction.

If the BSpline surface is not periodic in a given parametric direction, the bounds of the knots and multiplicities tables are 1 and NbKnots, where NbKnots is the number of knots of the BSpline surface in that parametric direction.

If the BSpline surface is periodic in a given parametric direction, and there are k periodic knots and p periodic poles in that parametric direction:

- the period is such that:

$period = Knot(k+1) - Knot(1)$

- the poles and knots tables in that parametric direction can be considered as infinite tables, such that:

$Knot(i+k) = Knot(i) + period$

$Pole(i+p) = Pole(i)$

The data structure tables for a periodic BSpline surface are more complex than those of a non-periodic one.

1.2.5.4 OCAS classes: GeomFill_BSplineCurves

Purpose: An algorithm for constructing a BSpline surface filled from contiguous BSpline curves which form its boundaries.

The algorithm accepts two, three or four BSpline curves as the target surface's boundaries.

A range of filling styles - more or less rounded, more or less flat - is available.

A BSplineCurves object provides the framework for:

- defining the boundaries, and the filling style of the surface
- implementing the construction algorithm
- consulting the result.

Some problems may show up with rational curves.

constructors:

GeomFill_BSplineCurves()
Constructs a default BSpline surface framework.

GeomFill_BSplineCurves (const Handle(Geom_BSplineCurve)& C1,
const Handle(Geom_BSplineCurve)& C2,
const Handle(Geom_BSplineCurve)& C3,
const Handle(Geom_BSplineCurve)& C4,
const GeomFill_FillingStyle Type)
Constructs a framework for building a BSpline surface from the four contiguous BSpline curves, C1, C2, C3 and C4

GeomFill_BSplineCurves (const Handle(Geom_BSplineCurve)& C1,
const Handle(Geom_BSplineCurve)& C2,
const Handle(Geom_BSplineCurve)& C3,
const GeomFill_FillingStyle Type)
Constructs a framework for building a BSpline surface from the three contiguous BSpline curves, C1, C2 and C3

GeomFill_BSplineCurves (const Handle(Geom_BSplineCurve)& C1,
const Handle(Geom_BSplineCurve)& C2,
const GeomFill_FillingStyle Type)
Constructs a framework for building a BSpline surface from the two contiguous BSpline curves, C1 and C2

const Handle_Geom_BSplineSurface& Surface() const

Returns the BSpline surface **Surface** resulting from the computation performed by this algorithm.

GeomFill_FillingStyle

enum GeomFill_FillingStyle (GeomFill_Coons, GeomFill_Curved, GeomFill_Stretch)

Defines the three filling styles used in this package:

- **GeomFill_Stretch** - the style with the flattest patches

- **GeomFill_Coons** - a rounded style of patch with less depth than those of Curved

- **GeomFill_Curved** - the style with the most rounded patches.

Connecting contiguous B-Spline Curves: Code Example (6)

```
GeomFill_BSplineCurves surfgenOff;
GeomFill_FillingStyle Type=GeomFill_CoonsStyle;
surfgenOff = GeomFill_BSplineCurves (baseOn,baseOff,Type);
Handle(Geom_BSplineSurface) supfOff=surfgenOff.Surface();
```

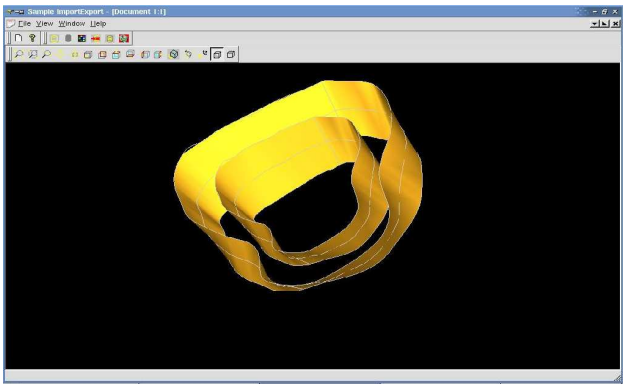

Figure 1.7: Shell for the inner and for the outer curves.

1.2.6 Creating shells

Geometry definition of surfaces ends with the section 5, and now starts the topological construction of the object.

Geometry is representation of simple shapes which have a mathematical description (Cylinder, Planes, Bezier and BSpline surfaces...)

Topology Defines relations between simple geometric entities and involves structuring geometry in order to:

Delimit a geometric region limited by boundaries

Group regions together to define complex shapes

In sections 1.2.6.1, 1.2.6.2, 1.2.6.3, 1.2.6.4, 1.2.6.5, 1.2.6.6 we describe classes and methods utilized during the development of the code and we give an example of the developed code. After the creation of **TopoDS_Shell** from union B-Spline surfaces we put them in a list (see section 7) useful in the future to create **TopoDS_Shape** valid objects; after the last shell is realized, we scan the list and compose the surfaces to realize the entire object, as can be seen in (Figure 1.8).

1.2.6.1 OCAS classes: BRepBuilderAPI_MakeShape

Root class of all shape construction algorithms.

A MakeShape object provides the framework for consulting and storing the result of a shape construction.

virtual Standard_Boolean IsDone() const Returns true if the object was correctly built by the shape construction algorithm.

If at the construction time of the shape, the algorithm cannot be completed, or the original data is corrupted, **IsDone** returns false and therefore protects the use of functions to access the result of the construction (typically the **Shape** function).

const TopoDS_Shape& Shape() const Returns the shape built by the shape construction algorithm

1.2.6.2 OCAS classes: BRepBuilderAPI_MakeShell

Purpose: Describes functions to build a shape corresponding to the skin of a surface.

Note that the term shell in the class name has the same definition as a shell in STEP, in other words the skin of a shape, and not a solid model defined by a surface and a thickness. If you want to build the second sort of shell, you must use **BRepAPI_MakeOffsetShape**.

A shell is made of a series of faces connected by their common edges.

If the surface is *C2* continuous, the shell will contain only 1 face. If the surface is not *C2* continuous, **MakeShell** breaks down the surface into several faces which are all *C2* continuous and which are connected along the non-regular curves on the surface. The resulting shell contains all these faces.

Construction of a Shell from a non-C2 continuous Surface

A MakeShell object provides the framework for:

- defining the construction of a shell,
- implementing the construction algorithm,
- consulting the result.

The connected C2 faces in the shell resulting from a decompositon of the surface are not sewn.

Constructors:

BRepBuilderAPI_MakeShell(const Handle(Geom_Surface)& S, const Standard_Boolean Segment)

Constructs a shell from the surface S.

1.2.6.3 OCAS classes: BRepBuilderAPI_MakeSolid

Purpose:

Describes functions to build a solid from shells.

A solid is made of one shell, or a series of shells which do not intersect each other. One of these shells constitutes the outside skin of the solid. It may be closed (a finite solid) or open (an infinite solid). Other shells make holes in the previous ones. Each must bound a closed volume.

A MakeSolid object provides the framework for:

- defining and implementing the construction of a solid, and

- consulting the result.

Constructors:

BRepBuilderAPI_MakeSolid() Initializes the construction of a solid. An empty solid is considered to cover the full space. The **Add** function is used to define shells to bound it.

BRepBuilderAPI_MakeSolid(const TopoDS_Shell& S) Constructs a solid from the shell S

BRepBuilderAPI_MakeSolid(const TopoDS_Shell& S1, const TopoDS_Shell& S2) Constructs a solid from two shells S1 and S2

BRepBuilderAPI_MakeSolid(const TopoDS_Shell& S1, const TopoDS_Shell& S2, const TopoDS_Shell& S3) Constructs a solid from three shells S1, S2 and S3

No check is done to verify conditions of coherence of the resulting solid. In particular, S1, S2 (and S3) must not intersect each other.

Also, after all shells have been added using the Add function, one of these shells should constitute the outside skin of the solid, it may be closed (a finite solid) or open (an infinite solid). Other shells make holes in the previous ones. Each must bound a closed volume.

BRepBuilderAPI_MakeSolid(const TopoDS_Solid& So) Constructs a solid from the solid So, to which shells can be added

BRepBuilderAPI_MakeSolid(const TopoDS_Solid& So, const TopoDS_Shell& S) Constructs a solid by adding the shell S to the solid So

No check is done to verify conditions of coherence of the resulting solid. In particular S must not intersect the solid S0.

Also, after all shells have been added using the Add function, one of these shells should constitute the outside skin of the solid. It may be closed (a finite solid) or open (an infinite solid). Other shells make holes in the previous ones. Each must bound a closed volume.

void Add(const TopoDS_Shell& S) Adds the shell S to the solid under construction

OCAS classes: TopAbs_ShapeEnum Enumeration

enum TopAbs_ShapeEnum
(TopAbs_COMPOUND,
TopAbs_COMPSOLID,
TopAbs_SOLID,
TopAbs_SHELL,
TopAbs_FACE,
TopAbs_WIRE,
TopAbs_EDGE,
TopAbs_VERTEX,
TopAbs_SHAPE)

Purpose:

Identifies the various topological shapes. This enumeration allows you to use dynamic typing of shapes.

The values are listed in order of complexity, from the most complex to the most simple i.e.

COMPOUND > COMPSOLID > SOLID > SHELL > FACE > WIRE > EDGE > VERTEX > SHAPE.

Any shape can contain simpler shapes in its definition.

Abstract topological data structure describes a basic entity, the shape (present in this enumeration as the SHAPE value), which can be divided into the following component topologies:

COMPOUND: A group of any of the shapes below.

COMPSOLID: A set of solids connected by their faces. This expands the notions of WIRE and SHELL to solids.

SOLID: A part of 3D space bounded by shells.

SHELL: A set of faces connected by some of the edges of their wire boundaries. A shell can be open or closed.

FACE: Part of a plane (in 2D geometry) or a surface (in 3D geometry) bounded by a closed wire. Its geometry is constrained (trimmed) by contours.

WIRE: A sequence of edges connected by their vertices. It can be open or closed depending on whether the edges are linked or not.

EDGE: A single dimensional shape corresponding to a curve, and bound by a vertex at each extremity.

VERTEX: A zero-dimensional shape corresponding to a point in geometry.

1.2.6.4 OCAS classes: TopoDS_Shape

Purpose: Describes a shape which:

- references an underlying shape with the potential to be given a location and an orientation

- has a location for the underlying shape, giving its placement in the local coordinate system

- has an orientation for the underlying shape, in terms of its geometry (as opposed to orientation in relation to other shapes).

TopoDS_Shape() Constructs an empty shape.

1.2.6.5 OCAS classes: TopoDS_Shell

Purpose: Describes a shell which:

- references an underlying shell with the potential to be given a location and an orientation

- has a location for the underlying shell, giving its placement in the local coordinate system

- has an orientation for the underlying shell, in terms of its geometry (as opposed to orientation in relation to other shapes).

TopoDS_Shell() Constructs an empty shell.

1.2.6.6 OCAS classes: TopoDS_Solid

Purpose: Describes a solid shape which:

- references an underlying solid shape with the potential to be given a location and an orientation

- has a location for the underlying shape, giving its placement in the local coordinate system

- has an orientation for the underlying shape, in terms of its geometry (as opposed to orientation in relation to other shapes).

TopoDS_Solid() Constructs an undefined solid.

1.2.6.7 OCAS classes: TCollection_List

Purpose:

Ordered lists of non-unique objects which can be accessed sequentially using an iterator.

Item insertion in a list is very fast at any position. But searching for items by value may be slow if the list is long, because it requires a sequential search.

List is a generic class which depends on Item, the type of element in the structure.

Use a ListIterator iterator to explore a List structure.

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

- A sequence is a better structure when searching for items by value is an important goal for a data structure.

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

void Append (const Item& thing) Inserts the item thing:

- at the end of this list

void Prepend (const Item& thing) Inserts the item thing:

- at the beginning of this list

void Remove (TCollection_ListIterator& pos) Removes from this list the current item of the list iterator pos.

This operation moves the list iterator pos to the next item of this list, if it exists.

Item& First() const Returns the first item in this list

Item& Last() const Returns the last item in this list

Standard_Boolean IsEmpty() const Returns true if this list is empty

OCAS classes: TopTools_ListOfShape

Instantiates **TCollection_List** with **TopoDS_Shape**

Creating shells: Code Example (7)

```
BRepBuilderAPI_MakeShell shellgenoff(supfOff, Standard_False);
TopTools_ListOfShape lista;
lista.Prepend(shellgenoff);
```

1.2.7 Lumen bifurcation

During the reconstruction of bifurcation we have to add some steps to the procedure of reconstruction illustrated above. First of all we have to verify the intersections of B-Spline curves with same z (see Figure 1.9) using the class **Geom2dAPI_InterCurveCurve**; for further details on this class and for a code example see section 1.2.7.1; between class methods there is one that gives us the number of intersections and another one that gives us intersection cartesian coordinates. If there is an intersection we fuse two intersecting arrays to compose a single array as shown in Figure 1.10; then we divide the single array in two open arrays. We need open arrays as supports to overcome the difficulties given by the topological change passing from slices with just one contour

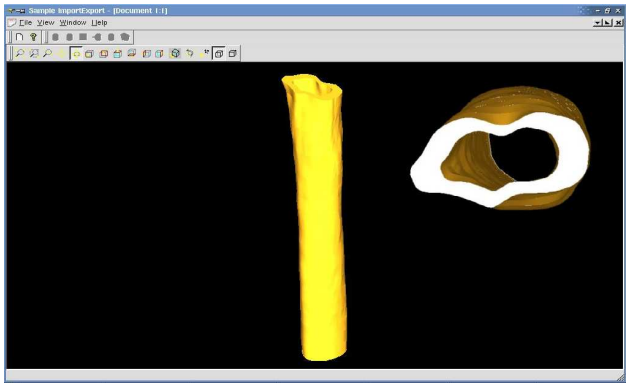

Figure 1.8: Main branch of carotid in Boundary Representation (BRep).

to slices with two different contours. With open arrays we can create different open **B-Spline** Curves, joining them with B-Spline surfaces we obtain three patches (Figure 1.11) and joining them is possible to create a single surface of bifurcation (Figure 1.12).

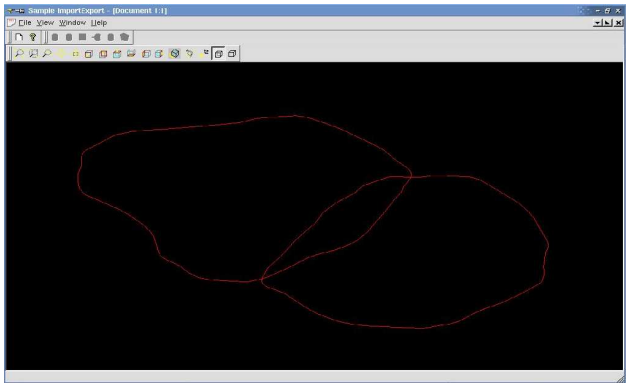

Figure 1.9: Intersection of contours.

1.2.7.1 OCAS classes: Geom2dAPI_InterCurveCurve

Purpose:

Describes functions for computing the intersections between two 2D curves, or the self-intersections of a 2D curve.

An **InterCurveCurve** algorithm computes both:

- intersection points in the case of cross intersections, and
- intersection segments in the case of tangential intersections.

An **InterCurveCurve** object provides the framework for:

- defining the construction of the intersections,

- implementing the intersection algorithm, and

- consulting the results.

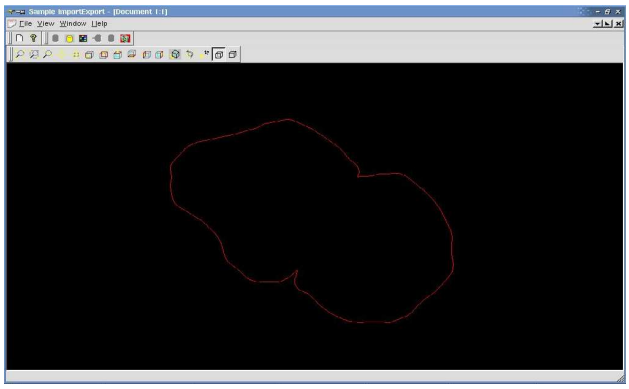

Figure 1.10: Single contour generation from two intersecting contours.

Geom2dAPI_InterCurveCurve (const Handle(Geom2d_Curve)& C1, const Handle(Geom2d_Curve)& C2, const Standard_Real Tol = 1.0e-6)

Computes the intersections between the curves C1 and C2

The tolerance value Tol, defaulted to $1.0\,\exp^{-6}$, defines the precision with which an intersection point is computed. In the case of a tangential intersection, Tol also enables the size of an intersection segment to be limited to a portion of the curve where the distance between the points and the other curve is less than Tol.

Standard_Integer NbPoints() const Returns the number of intersection points computed by this algorithm.

Standard_Integer NbSegments() const Returns the number of tangential intersections computed by this algorithm.

If this algorithm fails, NbPoints and NbSegments return 0.

gp_Pnt2d Point (const Standard_Integer Index) const Returns the intersection point of index Index computed by this algorithm.

An intersection point is computed in the case of a free intersection (i.e. non-tangential intersection). The solution is defined with a precision equal to the tolerance value assigned to this algorithm at the time of construction (this value is defaulted to $1.0\,\exp^{-6}$).

Lumen bifurcation: Code Example (8)

```
Geom2dAPI_InterCurveCurve inters(baseOn,baseOff,1.0e-5);
Standard_Real f1x,f1y;
f1x=inters.Point(1).X();
f1y=inters.Point(1).Y();
...
```

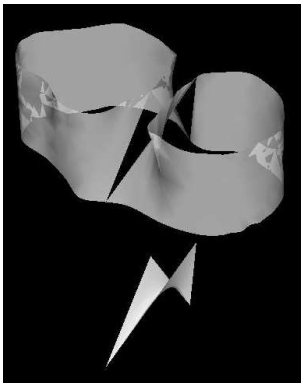

Figure 1.11: Generation of three patch-shells for bifurcation.

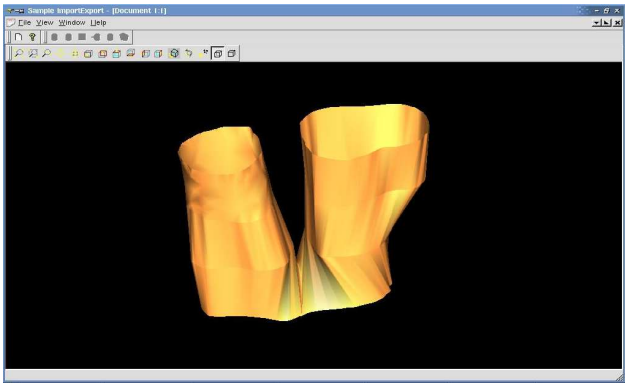

Figure 1.12: Union of patches to obtain the bifurcation shell.

```
GeomAPI_Interpolate in_ar1= GeomAPI_Interpolate(ar1,Seg,AngTol);
GeomAPI_Interpolate in_ar2= GeomAPI_Interpolate(ar2,Seg,AngTol);
cur1=in_ar1.Curve();
cur2=in_ar2.Curve();
GeomFill_BSplineCurves surfgreg;
surfgreg = GeomFill_BSplineCurves (cur1, cur2, Type);
Handle(Geom_BSplineSurface) supfgreg=surfgreg.Surface();
BRepBuilderAPI_MakeShell shellgreg(supfgreg, Seg);
...
BRepOffsetAPI_Sewing union(RealTol,Sewer);
union.Add(shell1);
union.Add(shell2);
...
union.Perform();
TopoDS_Shape bifurcation=union.SewedShape();
```

1.2.8 Joining shells

We have just to join the shells of each branch to obtain a single solid (Figure 1.13). Each **TopoDS_Shape** object can be exported in a file .brep format using the tools described on section 1.2.8.2 with a code sample.

1.2.8.1 OCAS classes: BRepOffsetAPI_Sewing

Purpose:

Describes functions to assemble shapes by sewing them along common edges or sections of edges.

The edges of the initial shapes are broken down by the algorithm into contiguous and non-contiguous sections, and copies of the initial shapes are modified accordingly.

A Sewing object provides the framework for:

- initializing a sewing algorithm,
- defining the shapes to be sewn,
- implementing the sewing algorithm, and
- consulting the results.

BRepOffsetAPI_Sewing (const Standard_Real tolerance = $1.0\exp^{-06}$, const Standard_Boolean option = Standard_True)

Initializes a sewing algorithm with tolerance as the tolerance of contiguity. The default value is $1.0\exp^{-6}$.

This tolerance value is used to determine whether two edges or two portions of edges are coincident.

If an analysis of degenerate shapes is required, you should keep the default value of option: true; if not, you should set this argument to false.

void Add(const TopoDS_Shape& shape)

Adds the shape *shape* to the list of shapes to be sewn by this algorithm.

Once all the shapes to be sewn have been added, use the function **Perform** to build the sewn shape and the function **SewedShape** to return the sewn shape.

void Perform()

Finds the coincident parts of edges on two or more shapes added to this algorithm and breaks these edges down into contiguous and non-contiguous sections on copies of the initial shapes.

This function also checks for multiple edges, i.e. edges common to three or more shapes. If there are no multiple edges, the sewn shape is then built.

The function SewedShape returns the resulting sewn shape. The function MultipleEdge can be used to return the multiple edges.

The function Modified can be used to return modified copies of the initial shapes where one or more edges has been broken down into contiguous and non-contiguous sections.

This function is to be used once all the shapes to be sewn have been added. You cannot then add any more shapes and repeat the call to Perform.

const TopoDS_Shape& SewedShape() const

Returns the sewn shape built by this algorithm.

A null shape is returned if the sewn shape is not constructed.

1.2.8.2 OCAS classes: BRepTools

Group of global functions for manipulation of BRep data structures.

static Standard_Boolean Write(const TopoDS_Shape& Sh, const Standard_CString File)

Saves the shape Sh in ASCII format in the file File.

File is in the Open CASCADE .brep format. This format is designed for saving shapes.

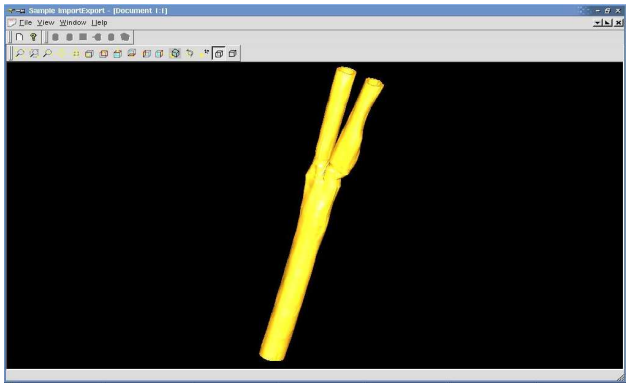

Figure 1.13: The final result of the reconstruction process, as it appears on the visualization tool

Joining shells: Code example (9)

```
BRepBuilderAPI_MakeSolid SOLID;
SOLID.Add(shellgen.Shell());
...
TopoDS_Solid final=SOLID.Solid();
const Standard_CString filebrep ="final.brep";
BRepTools::Write(final, filebrep);
```

1.2.9 Exporting tesselled lumen geometry: STL file

Splinetor ends its work exporting the obtained shape in a file written in stl format; for details about stl see appendix C. STL is the input format accepted by FDM 2000 device, our 3D printer. A global description of OCAS class can be found on section 1.2.9.1 with a code example. Changing values of deflection and coefficient we can affect the number of triangles of the resulting mesh.

1.2.9.1 OCAS classes: StlAPI_Writer

Purpose: Writing shapes to stereolithography format.

void SetDeflection(const Standard_Real aDeflection) Sets the deflection for meshing algorithm. Deflection is used, only if relative mode is false. Default value = 0.01.

void SetCoefficient(const Standard_Real aCoefficient) Sets the coefficient for computation of deflection through relative size of shape. Default value = 0.001.

Standard_Boolean RelativeMode() Returns (modifiable) the flag which defines relative mode, if true (default),then the value of the deflection is to be evaluated with the Shape bounding box method.

Standard_Boolean ASCIIMode() Returns (modifiable) the flag which defines mode for writing file, if true (default),then the file is ascii, else file is binary.

void Write(const TopoDS_Shape& aShape,const Standard_CString aFileNameme) Convert given shape to STL format and write it to file with given filename.

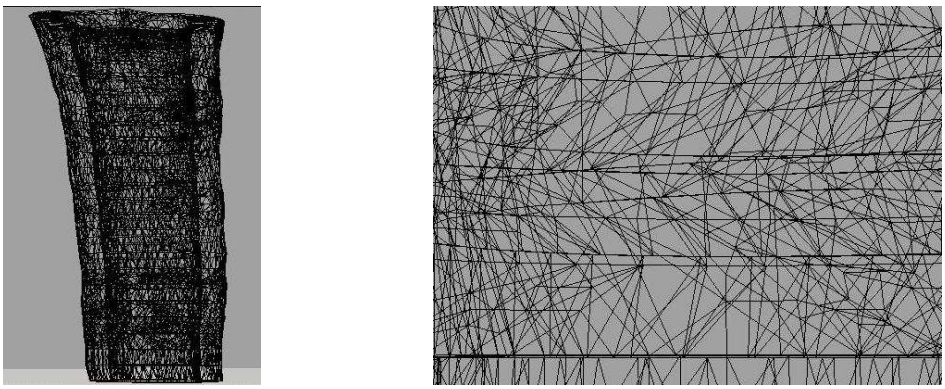

Figure 1.14: a) Visualization of file.stl obtained. b) Detail of the mesh.

Exporting tesselled lumen geometry: Code example (10)

```
StlAPI_Writer stlwriter;
stlwriter = StlAPI_Writer();
const Standard_CString filestl ="final.stl";
stlwriter.Write(final,filestl);
```

Chapter 2

Sample: Visualization Tool

2.1 Visualizer Sample

We developed with **Qt** libraries an *"ad hoc"* tool for visualization of vessel geometry realized with **OCAS** library. This tool is able to visualize 3D carotid lumen reconstruction (see Figure 2.1) . This section describes how to use Qt to develop an interactive 3D visualizer tool able to handle OpenCascade BRep models.

2.2 About Qt

Qt [16] is a multiplatform **C++ GUI** application framework. It provides application developers with all the functionality needed to build applications with **state-of-the-art** graphical user interfaces. **Qt** is fully object-oriented, easily extensible, and allows true component programming.

Qt is also the basis of the popular KDE Linux desktop environment, a standard component of all major Linux distributions.

Qt is supported on the following platforms:

- MS/Windows – 95, 98, NT 4.0, ME, 2000, and XP
- Unix/X11 – Linux, Sun Solaris, HP-UX, Compaq Tru64 UNIX, IBM AIX, SGI IRIX and a wide range of others

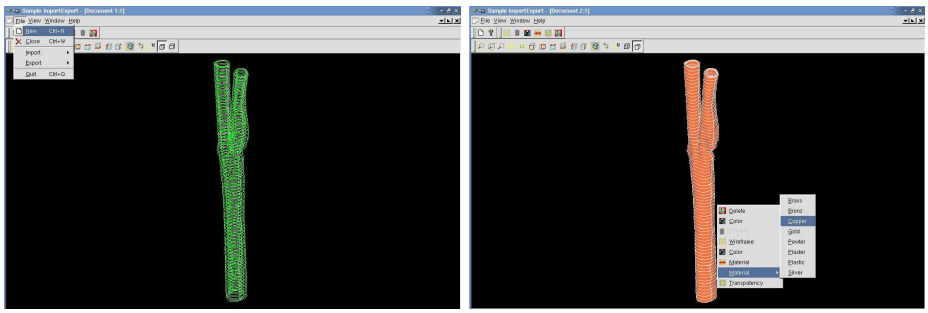

Figure 2.1: Reconstructed carotid visualization with developed QT-visualizer

- Macintosh – Mac OS X
- Embedded – Linux platforms with framebuffer support.

2.2.1 Qt Object Model

The standard C++ Object Model provides very efficient runtime support for the object paradigm. But the C++ Object Model's static nature is inflexibile in certain problem domains. Graphical User Interface programming is a domain that requires both runtime efficiency and a high level of flexibility. **Qt** provides this, by combining the speed of **C++** with the flexibility of the Qt Object Model.

Qt adds these features to **C++**:

* a very powerful mechanism for seamless object communication called signals and slots;

* queryable and designable object properties;

* powerful events and event filters,

* contextual string translation for internationalization;

* sophisticated interval driven timers that make it possible to elegantly integrate many tasks in an event-driven GUI;

* hierarchical and queryable object trees that organize object ownership in a natural way;

* guarded pointers, QGuardedPtr, that are automatically set to 0 when the referenced object is destroyed, unlike normal C++ pointers which become "dangling pointers" when their objects are destroyed.

Many of these Qt features are implemented with standard C++ techniques, based on inheritance from QObject. Others, like the object communication mechanism and the dynamic property system, require the Meta Object System provided by Qt's own Meta Object Compiler (moc).

The Meta Object System is a C++ extension that makes the language better suited to true component GUI programming. Although templates can be used to extend C++, the Meta Object System provides benefits using standard C++ that cannot be achieved with templates.

2.2.2 Object Trees and Object Ownership

QObjects organize themselves in object trees. When you create a **QObject** with another object as parent, it's added to the parent's **children()** list, and is deleted when the parent is. It turns out that this approach fits the needs of GUI objects very well. For example, a **QAccel** (keyboard accelerator) is a child of the relevant window, so when the user closes that window, the accelerator is deleted too.

The static function **QObject::objectTrees()** provides access to all the root objects that currently exist.

QWidget, the base class of everything that appears on the screen, extends the parent-child relationship. A child normally also becomes a child widget, i.e. it is displayed in its parent's coordinate system and is graphically clipped by its parent's boundaries. For example, when the an application deletes a message box after it has been closed, the message box's buttons and label are also deleted, just as we'd want, because the buttons and label are children of the message box.

You can also delete child objects yourself, and they will remove themselves from their parents. For example, when the user removes a toolbar it may lead to the application deleting one of its QToolBar objects, in which case the tool bar's QMainWindow parent would detect the change and reconfigure its screen space accordingly.

The debugging functions **QObject::dumpObjectTree()** and **QObject-::dumpObjectInfo()** are often useful when an application looks or acts strangely.

2.2.3 Signals and Slots

Signals and slots are used for communication between objects. The signal/slot mechanism is a central feature of Qt and probably the part that differs most from other toolkits.

In GUI programming we often want a change in one widget to be notified to another widget. More generally, we want objects of any kind to be able to communicate with one another. For example if we were parsing an XML file we might want to notify a list view that we're using to represent the XML file's structure whenever we encounter a new tag.

Older toolkits achieve this kind of communication using callbacks. A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback when appropriate. Callbacks have two fundamental flaws. Firstly they are not type safe. We can never be certain that the processing function will call the callback with the correct arguments. Secondly the callback is strongly coupled to the processing function since the processing function must know which callback to call.

In Qt we have an alternative to the callback technique. We use signals and slots. A signal is emitted when a particular event occurs. Qt's widgets have many pre-defined signals, but we can always subclass to add our own. A slot is a function that is called in reponse to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to add your own slots so that you can handle the signals that you are interested in.

The signals and slots mechanism is type safe: the signature of a signal must match the signature of the receiving slot. (In fact a slot may have a shorter signature than the signal it receives because it can ignore extra arguments). Since the signatures are compatible, the compiler can help us detect type mismatches. Signals and slots are loosely coupled: a class which emits a signal neither knows nor cares which slots receive the signal. Qt's signals and slots mechanism ensures that if you connect a signal to a slot, the slot will be called with the signal's parameters at the right time. Signals and slots can take any number of

arguments of any type. They are completely typesafe: no more callback core dumps!

All classes that inherit from **QObject** or one of its subclasses (e.g. **QWidget**) can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to the outside world. This is all the object does to communicate. It does not know or care whether anything is receiving the signals it emits. This is true information encapsulation, and ensures that the object can be used as a software component.

Slots can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt.

You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you desire. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

Together, signals and slots make up a powerful component programming mechanism.

2.2.4 Signals

Signals are emitted by an object when its internal state has changed in some way that might be interesting to the object's client or owner. Only the class that defines a signal and its subclasses can emit the signal.

A list box, for example, emits both clicked() and currentChanged() signals. Most objects will probably only be interested in currentChanged() which gives the current list item whether the user clicked it or used the arrow keys to move to it. But some objects may only want to know which item was clicked. If the signal is interesting to two different objects you just connect the signal to slots in both objects.

When a signal is emitted, the slots connected to it are executed immediately, just like a normal function call. The signal/slot mechanism is totally independent of any GUI event loop. The emit will return when all slots have returned.

If several slots are connected to one signal, the slots will be executed one after the other, in an arbitrary order, when the signal is emitted.

Signals are automatically generated by the moc and must not be implemented in the .cpp file. They can never have return types (i.e. use void).

2.2.5 Slots

A slot is called when a signal connected to it is emitted. Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them. A slot's arguments cannot have default values, and, like signals, it is rarely wise to use your own custom types for slot arguments.

Since slots are normal member functions with just a little extra spice, they have access rights like ordinary member functions. A slot's access right determines who can connect to it:

A public slots section contains slots that anyone can connect signals to. This is very useful for component programming: you create objects that know

nothing about each other, connect their signals and slots so that information is passed correctly, and, like a model railway, turn it on and leave it running.

A protected slots section contains slots that this class and its subclasses may connect signals to. This is intended for slots that are part of the class's implementation rather than its interface to the rest of the world.

A private slots section contains slots that only the class itself may connect signals to. This is intended for very tightly connected classes, where even subclasses aren't trusted to get the connections right.

You can also define slots to be virtual, which we have found quite useful in practice.

The signals and slots mechanism is efficient, but not quite as fast as *"real"* callbacks. Signals and slots are slightly slower because of the increased flexibility they provide, although the difference for real applications is insignificant. In general, emitting a signal that is connected to some slots, is approximately ten times slower than calling the receivers directly, with non-virtual function calls. This is the overhead required to locate the connection object, to safely iterate over all connections (i.e. checking that subsequent receivers have not been destroyed during the emission) and to marshall any parameters in a generic fashion. While ten non-virtual function calls may sound like a lot, it's much less overhead than any 'new' or 'delete' operation, for example. As soon as you perform a string, vector or list operation that behind the scene requires 'new' or 'delete', the signals and slots overhead is only responsible for a very small proportion of the complete function call costs. The same is true whenever you do a system call in a slot; or indirectly call more than ten functions. On an i586-500, you can emit around 2,000,000 signals per second connected to one receiver, or around 1,200,000 per second connected to two receivers. The simplicity and flexibility of the signals and slots mechanism is well worth the overhead, which your users won't even notice.

2.2.6 Meta Object System

Qt's Meta Object System provides the signals and slots mechanism for inter-object communication, runtime type information, and the dynamic property system.

The Meta Object System is based on three things:

1. 1. the **QObject** class;
2. 2. the **Q_OBJECT** macro inside the private section of the class declaration;
3. 3. the **Meta Object Compiler** (moc).

The **moc** reads a C++ source file. If it finds one or more class declarations that contain the Q_OBJECT macro, it produces another C++ source file which contains the meta object code for the classes that contain the Q_OBJECT macro. This generated source file is either #included into the class's source file or compiled and linked with the class's implementation.

In addition to providing the signals and slots mechanism for communication between objects (the main reason for introducing the system), the meta object code provides additional features in QObject:

- the **className()** function that returns the class name as a string at runtime, without requiring native runtime type information (RTTI) support through the C++ compiler.
- the **inherits()** function that returns whether an object is an instance of a class that inherits a specified class within the QObject inheritance tree.
- the **tr()** and **trUtf8()** functions for string translation as used for internationalization.
- the **setProperty()** and **property()** functions for dynamically setting and getting object properties by name.
- the **metaObject()** function that returns the associated meta object for the class.

While it is possible to use QObject as a base class without the Q_OBJECT macro and without meta object code, neither signals and slots nor the other features described here will be available if the Q_OBJECT macro is not used. From the meta object system's point of view, a QObject subclass without meta code is equivalent to its closest ancestor with meta object code. This means for example, that className() will not return the actual name of your class, but the class name of this ancestor.

2.2.7 Short description

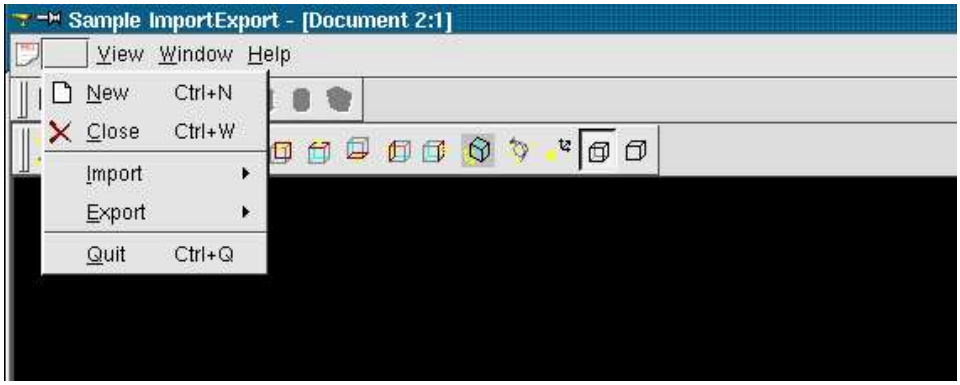

Figure 2.2: Detail of File menu.

Sample interface has 3 options menus:

File

Choosing **New** we open a new graphic window with 5 options menus and 2 toolbars.

File menu fig. 2.2 has 2 important options:

Import: In one of its submenus we find the abbreviations for the formats we can import. Accepted input formats are:

- BRep

```
void Translate::importBREP(const Handle(AIS_InteractiveContext)
                       theContext, const QString& filter)
{
    QString file = selectFileName( filter, TRUE );
    if ( !file.isNull() ) {
        QApplication::setOverrideCursor(Qt::waitCursor);
                if(!importBREP(theContext,
                  (const Standard_CString) file.latin1()))
                {
                   QApplication::restoreOverrideCursor();
                   QMessageBox::information
                     (qApp->mainWidget(),tr("TIT_ERROR"),
                     tr("INF_TRANSLATE_ERROR"), tr("BTN_OK"),
                   QString::null, QString::null, 0, 0);
                   qApp->processEvents();/* update desktop */
                }
                else
                   QApplication::restoreOverrideCursor();
    }
}
```

- **Csfdb**
- **Iges**
- **Step**

Export: In addition to import formats there are these possible output formats fig. 2.3:

- **Stl**
- **Vrml**
- **bmp**
- **gif**
- **xwd**

View

View menu has two visualizatin possibilities:

Tiled: with more than one window on the screen

Cascade: with just a window on the screen and the others minimized

toolbar

The toolbar holds the icons indicating the points of view.

Front: front view

Back: back view

Top: top view

Bottom: bottom view

Left: left view

Right: right view

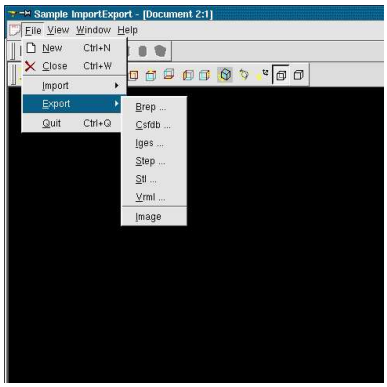

Figure 2.3: View of export options.

Axo: axonometric view

Other option-buttons on the toolbar are:

Fit_All: the fitting of the object regarding to the window

Zoom_Window: zoom about a mouse-selected region

Dynamic_Zooming: dynamic zoom with mouse, going left the object decrease, going right the object increase

Dynamic_Panning: object translation with mouse

Global_Panning: close-up positioning of the object

Dynamic_Rotation: object rotation with mouse

Reset: restore the original situation after import step

Hidden_Off: hides the edges at the object back

Hidden_On: shows the edges behind the object

```
void MDIWindow::createViewActions()
{
  // populate a tool bar with some actions

  QToolBar* aToolBar = new QToolBar(this,"view operations");
  aToolBar->setLabel( tr( "View Operations" ) );

  QList<QAction>* aList = myOperations->getViewActions();

  for(QAction* a = aList->first();a;a = aList->next())
     a->addTo(aToolBar);

  connect(myOperations,SIGNAL(fitAll()),myView,SLOT(fitAll()));
  connect(myOperations,SIGNAL(fitArea()),myView,SLOT(fitArea()));
  connect(myOperations,SIGNAL(zoom()),myView,SLOT(zoom()));
  connect(myOperations,SIGNAL(pan()),myView,SLOT(pan()));
  connect(myOperations,SIGNAL(rotation()),myView,SLOT(rotation()));
```

```
  connect(myOperations,SIGNAL(globalPan()),myView,SLOT(globalPan()));
  connect(myOperations,SIGNAL(front()),myView,SLOT(front()));
  connect(myOperations,SIGNAL(back()),myView,SLOT(back()));
  connect(myOperations,SIGNAL(top()),myView,SLOT(top()));
  connect(myOperations,SIGNAL(bottom()),myView,SLOT(bottom()));
  connect(myOperations,SIGNAL(left()),myView,SLOT(left()));
  connect(myOperations,SIGNAL(right()),myView,SLOT(right()));
  connect(myOperations,SIGNAL(axo()),myView,SLOT(axo()));
  connect(myOperations,SIGNAL(reset()),myView,SLOT(reset()));
  connect(myOperations,SIGNAL(hlrOn()),myView,SLOT(hlrOn()));
  connect(myOperations,SIGNAL(hlrOff()),myView,SLOT(hlrOff()));

  aList->at(ViewOperations::ViewHlrOffId)->setOn(TRUE);

}
```

It is possible to visualize another pop-up menu clicking on the object with the mouse right button. The options are in fig. 2.4:

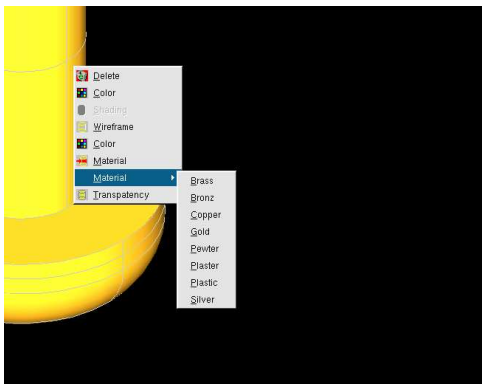

Figure 2.4: Materials options.

Delete: Deletes the object

Color: Changes the color of the object

Shading: Shades the object

Wireframe: Visualizates the wireframe of the object (default)

Material: Indicates the material properties of the object, we can choose between:

plaster

brass

```
b = new QPushButton(tr("BTN\_BRASS"),this);
sm->setMapping(b,(int) Graphic3d\_NOM\_BRASS);
connect(b,SIGNAL(clicked()),sm,SLOT(map()));
b->setToggleButton (TRUE);
```

```
connect(b,SIGNAL(toggled(bool)),this,SLOT(updateButtons(bool)));
myButtons.append(b);
```

bronze

copper

gold

pewter

plastic

silver

Trasparency: Adjusts the object transparency

Help

Appendices

A Open Source Philosophy

Science is ultimately an Open Source enterprise [17]. The scientific method rests on a process of discovery, and a process of justification. For scientific results to be justified, they must be replicable. Replication is not possible unless the source is shared: the hypothesis, the test conditions, and the results.

The process of discovery can follow many paths, and at times scientific discoveries do occur in isolation. But ultimately the process of discovery must be served by sharing information: enabling other scientists to go forward where one cannot; pollinating the ideas of others so that something new may grow that otherwise would not have been born.

Where scientists talk of replication, Open Source programmers talk of debugging. Where scientists talk of discovering, Open Source programmers talk of creating. Ultimately, the Open Source movement is an extension of the scientific method, because at the heart of the computer industry lies computer science.

Computer science, though, differs fundamentally from all other sciences. Computer science has only one means of enabling peers to replicate results: share the source code. To demonstrate the validity of a program to someone, you must provide them with the means to compile and run the program.

Replication makes scientific results robust. One scientist cannot expect to account for all possible test conditions, nor necessarily have the test environment to fully test every aspect of a hypothesis. By sharing hypotheses and results with a community of peers, the scientist enables many eyes to see what one pair of eyes might miss.

In the Open Source development model, this same principle is expressed as *"Given enough eyes, all bugs are shallow"*. By sharing source code, Open Source developers make software more robust. Programs get used and tested in a wider variety of contexts than one programmer could generate, and bugs get uncovered that otherwise would not be found. Because source code is provided, bugs can often be removed, not just discovered, by someone who otherwise would be outside the development process.

Open Source Software can be studied, altered and distributed freely without restriction other than the guarantee that those freedoms will never change. In contrast, proprietary software is usually not free. Users have to pay for a license, but cannot see how the software works and cannot change what it does without the permission of the owner. In fact, because the source code is not available to users, they cannot make any change even with the owner's permission. Users are tied to the owner's upgrade schedule, or they may find that their version is no longer supported. The most important benefits using open source can be resumed in:

- **Benefits for Businesses:** Companies that use Open Source software can customize and distribute the software without the need to acquire new licenses or permission. Regardless of where they obtained the software, businesses can find programmers and developers to provide them with support. Businesses can use Open Source software to expand or link to other services with a freedom that is often impossible with proprietary software. As an example, a company can create Open Source software

that connects and interfaces with a service. The software can be altered and improved upon and ported to various other operating systems creating more business.

- **Benefits for Developers:** Developers of Open Source software have the access and the right to alter the source code to make the changes and improvements their business needs, whenever they need them. Security fixes are one of the best examples of this. Security holes, when found, have fixes quickly published and shared freely by other developers for the benefit of all. Additionally, Open Source Software broadens the potential of jobs for programmers and developers since most of them work on customized software systems for businesses.

- **Benefits for Users:** Open Source software is often available for free, or at a minimal cost. If a user pays for Open Source software, it usually comes with support and printed documentation. If a user installs a free copy, then the user can often get the documentation for free online. Like users of proprietary software, users have the benefit of upgrades. But unlike proprietary software, users can get support even if the originator no longer provides for the service. The user can modify the software according its needs.

B Open Cascade

Open Cascade, or simply OCAS, by MATRA is a general-purpose CAD (Computer Aided Design) software. This system gives a complete application development environment, made up of about 10.000 classes written in C++ and organized in about 400 project files. In particular, an "OCAS project" is a collection of classes that share common functionality in some semantic area of geometry and/or graphics.

The classes of Open Cascade offers the infrastructure *(Rapid Application Framework)* for rapid development of geometric computing applications oriented or "design" in some specific areas of interest, say in advanced CAD tools, design databases, simulation systems or graphics rendering of complex assemblies.

CAS.Cade project (this name derives by the anagram of the words CASE and CAD) started in 1990 when Matra Datavision, which was at that time the producer of another traditional CAD called **Euclid**, wanted to develop one totally new modelling software.

It was decided to adopt the latest technology of the moment: all the application would have been based on the *object oriented* approach; in the specific it was adopted the C++ programming language. All geometric (and not only geometric) algorithms were completely rewritten.

In 1995 the *modelling kernel* was finished and Matra started the first debugging and testing session.

In 1999 the Matra decided to completely change its policy of software distribution: from software house it became a vendor of services (training courses, phone assistance etc.) and CAS.CADE was freely distributed under the *Open Source* licence (therefore it is now possible to use freely in the contest of the development of non commercial software).

Open Cascade Architecture

Ocas internal code organization is quite simple: all C++ *classes* are grouped under *packages*; every package belongs to an Ocas *library*; finally a group of libraries forms a *module*. The main reason of this structure is to link together services and algorithms which operate in the same semantic domain.

Ocas geometric services are divided into 6 categories :

[Foundation Class] Foundation Classes provide a variety of general-purpose services such as: primitive types, strings and various types of quantities, automated management of heap memory, exception handling, classes for manipulating aggregates of data, math tools etc.

[Modelling Data] Modelling Data supplies data structures to represent 2D and 3D geometric models. These services are organized in the following libraries: 2D geometry, 3D geometry, geometry Utilities, topology.

[Modelling Algorithms] The Modelling Algorithms module groups together a range of topological algorithms used in modelling. Along with these tools, it is possible to find the geometric algorithms which they call.

[Data Exchange Processor] The Data Exchange classes provide services that allow Open CASCADE applications to exchange data with other software

applications using the following interfaces: STEP AP203 AP214, IGES (it is also possible to import and export using some simpler vector data format such as VRML and BRep boundary representation).

[Visualization] For visualizing data structures, Open CASCADE provides ready-to-use algorithms which create graphic presentations from geometric models. These data structures may be used with the viewers supplied, and can be customized to take into account the specificity of your application.

[OCAF] The Open Cascade Application Framework provides modelling services, aiming to connect user data, even non-geometric, to parametric geometric models. It also provides further functionalities for storing and editing the history of a work session, and not only its results. This approach is embedded into an automatic mechanism for document and application generation, called application template.

Non Geometric Services

Whereas Open Cascade is a software environment specifically oriented for geometric modeling, it also gives other non-geometric services, aiming to support the application programmer along all the life-cycle of his application.

The non-geometric services may be classified in three main subsystems:

CDL The Component Definition Language (CDL) is the Open Cascade language for defining the interface of software components. A CDL file establishes the internal structure of C++ programs and data. For example a CDL class describes class constructors, instance or static methods and state variables. The following CDL link details important features about this language.

WOK The Workshop Organization Kit is the set of tools for the development of CDL based applications. In particular it contains: a system shell for command invocations; some tools for code generation through suitable program extractors; commands for automatic generation of project Makefile and for library compilation. This link explains how to get WOK working under Windows.

EDL The Open Cascade EDL language supports automatic generation of textual files from textual template to which we apply variable bindings. It is the main tools that permit extractors (CPPExtractor and SchemeExtractor) working. The following EDL link details important feature about this language.

Handles

Open CASCADE provide a smart pointer mechanism called **handle** which allows automatic memory management.

Handle definition

There are two types of classes:

- Classes of objectsused with handles
- Classes of objects used with values

The type of a C++ object determines the class category to which it belongs. A C++ object from a class manipulated by value contains an instance of that class. However, a C++ object from a class type manipulated by handle contains the identifier of the instance it references. In this case, the C++ object is called **handle**.

A handle can be compared with a C++ pointer. Several handles can reference the same object. Also, a single handle may reference several objects, but only one at a time. To have access to the object which it refers to, the handle must be dereferenced just as with C++ pointer.

Declaring a handle creates a null handle. To initialize it, either create a **new** object or assign the value of another handle to it, on condition that they are compatible.

C STL format

An Stl *"StereoLithography"* file is a triangular representation of a 3-dimensional surface geometry. The surface is tessellated or broken down logically into a series of small triangles (facets). Each facet is described by a perpendicular direction and three points representing the vertices (corners) of the triangle. These data are used by a slicing algorithm to determine the cross sections of the 3-dimensional shape to be built by the fabber.

Format Specifications

An StL file consists of a list of facet data. Each facet is uniquely identified by a unit normal (a line perpendicular to the triangle and with a length of 1.0) and by three vertices (corners). The normal and each vertex are specified by three coordinates each, so there is a total of 12 numbers stored for each facet.

The facets define the surface of a 3-dimensional object. As such, each facet is part of the boundary between the interior and the exterior of the object. The orientation of the facets (which way is "out" and which way is "in") is specified redundantly in two ways which must be consistent. First, the direction of the normal is outward. Second, the vertices are listed in counterclockwise order when looking at the object from the outside (right-hand rule).

Each triangle must share two vertices with each of its adjacent triangles. In other words, a vertex of one triangle cannot lie on the side of another. [Vertex-to-vertex rule in an StL file] The object represented must be located in the all-positive octant. In other words, all vertex coordinates must be positive-definite (nonnegative and nonzero) numbers. The StL file does not contain any scale information; the coordinates are in arbitrary units.

The official 3D Systems StL specification document states that there is a provision for inclusion of "special attributes for building parameters", but does not give the format for including such attributes. Also, the document specifies data for the "minimum length of triangle side" and "maximum triangle size", but these numbers are of dubious meaning.

Sorting the triangles in ascending z-value order is recommended, but not required, in order to optimize performance of the slice program.

Typically, an StL file is saved with the extension ".stl" case-insensitive. The slice program may require this extension or it may allow a different extension to be specified.

The stl standard includes two data formats, ASCII and binary. These are described separately below.

StL ASCII Format

The ASCII format is primarily intended for testing new CAD interfaces. The large size of its files makes it impractical for general use.

The syntax for an ASCII StL file is as follows:

```
facet normal -9.710413e-01 -1.289588e-01 -2.011177e-01
  outer loop
    vertex  1.679896e+01  2.067231e+01  0.000000e+00
    vertex  1.688584e+01  2.001808e+01  0.000000e+00
    vertex  1.663303e+01  2.036216e+01  1.000000e+00
```

```
   endloop
 endfacet
```

Words as vertex, face, loop, etc indicate a keyword; these must appear in lower case. Note that there is a space in "facet normal" and in "outer loop", while there is no space in any of the keywords beginning with "end." Indentation must be with spaces; tabs are not allowed.

The numerical data in the facet normal and vertex lines are single precision floats, for example, 1.23456E+789. A facet normal coordinate may have a leading minus sign; a vertex coordinate may not.

StL Binary Format

Binary (.STL) files are organized as an 84 byte header followed by 50-byte records each of which describes one triangle facet. The syntax for a binary StL file is as follows:

```
80 Any text such as the creator's name
4 int equal to the number of facets in file
facet 1
4
4
4
4
4
4
4
4
4
4
4
2 float normal x
float normal y
float normal z
float vertex1 x
float vertex1 y
float vertex1 z
float vertex2 x
float vertex2 y
float vertex2 z
float vertex3 x
float vertex3 y
float vertex3 z
unused (padding to make 50-bytes)
facet 2
4
4
4
4
4
```

```
4
4
4
4
4
4
4
2 float normal x
float normal y
float normal z
float vertex1 x
float vertex1 y
float vertex1 z
float vertex2 x
float vertex2 y
float vertex2 z
float vertex3 x
float vertex3 y
float vertex3 z
unused (padding to make 50-bytes)
facet 3
...
```

A facet entry begins with the x,y,z components of the triangle's face normal vector. The normal vector points in a direction away from the surface and it should be normalized to unit length. The x,y,z coordinates of the triangle's three vertices come next. They are stored in CCW order when viewing the facet from outside the surface. The direction of the normal vector follows the "right-hand-rule" when traversing the triangle vertices from 1 to 3, i.e., with the fingers of your right hand curled in the direction of vertex 1 to 2 to 3, your thumb points in the direction of the surface normal.

Notice that each facet entry is 50 bytes. So adding the 84 bytes in the header space, a binary file should have a size in bytes = 84 + (number of facets) * 50. Notice the 2 extra bytes thrown in at the end of each entry to make it a nice even 50. 50 is a nice number for people, but not for most 32-bit computers because they store values on 4-byte boundaries. Therefore, when writing programs to read and write .STL files the programmer has to take care to design data structures that accommodate this problem.

D Input File Format

Splinetor can read files that are in this format

```
3    number of cylinders
24    number of slices, main trunk
  0.00000000    128    z position, number of points
 16.67968154     21.69347531
 16.67968155     21.49954080
 16.68181197     21.26728675
 16.72706759     21.00147134
 16.78892233     20.71793090
 16.83589675     20.43246458
 16.83769052     20.22031165
...
...
...
...
 16.70328323     22.13026914
 16.68424170     21.99628725
 16.67968155     21.85144776
  3.00000000    128    z position, number of points
 16.46913256     21.67202869
 16.46913256     21.50906235
...
...
...
...
 13.63566729     19.51861262
 13.62010135     19.45360257
 13.60646260     19.39261272
 69.00000000    128    z position, number of points
 13.22942174     19.01031766
 13.23854038     18.93086623
...
...
...
...
 13.27043231     19.27580277
 13.25098770     19.18016276
 13.23617172     19.09329077
```

Bibliography

[1] T. Wohlers. *Wohlers Report 2001, Rapid Prototyping and Tooling State of the Industry Annual Worldwide Progress Report.* Wohlers associates, 2003.

[2] A. Gatto and L. Iuliano. *Prototipazione Rapida.* Tecniche Nuove, 1998.

[3] R. Petzold, H. F. Zeilhofer, and W. A. Kalender. Rapid prototyping technology in medicine - basics and applications. *Computerized Medical Imaging and Graphics*, 23:277–284, 1999.

[4] M. Marongiu, M. Camba, and P. Pili. La prototipazione rapida in italia e nel mondo: stato dell'arte. Technical Report 01/20, CRS4, Center for Advanced Studies, Research and Development in Sardinia, Cagliari, Italy, 2001.

[5] F. Murgia, P. Pili, and G. Pusceddu. Computer assisted surgery and rapid prototyping in medicine. Technical Report 02/08, CRS4, Center for Advanced Studies, Research and Development in Sardinia, Cagliari, Italy, 2002.

[6] G. Franzoni, R. de Leo, F. Murgia, P. Pili, G. Pusceddu, A. Scheinine, and M. Tuveri. Realizzazione di un prototipo di carotide con tecnica fused deposition modelling. Technical Report 02/07, CRS4, Center for Advanced Studies, Research and Development in Sardinia, Cagliari, Italy, 2002.

[7] F. Murgia, G. Pusceddu, and G. Franzoni. Open cascade and rapid prototyping in human carotid lumen reconstruction. *Proceedings of Euro Graphics 2002 - Italian Chapter*, 2002.

[8] P. Pili, F. Murgia, G. Pusceddu, G. Franzoni, and M. Tuveri. Physical human lumen reconstruction: Life-size models by rapid prototyping. *Medical Imaging 2003: Physiology and Function - Proceedings of SPIE 2003*, 5031:504–514, 2003.

[9] G. ABDULAEEV et al. Viva: the virtual vascular project. *Information Technology in Biomedicine*, pages 268–273, December 1988.

[10] Piero Pili Alan Scheinine and Fabrizio Murgia. Lumen carotid segmentation software: Programming user manual. Technical Report 03/09, CRS4, Center for Advanced Studies, Research and Development in Sardinia, Cagliari, Italy, 2003.

[11] R. Li. Data structures and application issues in 3d gis. *Geomatica*, 48 (3):111–130, 1994.

[12] Vuoskoski J. Sulkimo J. Particle tracking in sophisticated cad models for simulation purposes. *Nucl. Instr. Meth. Phys.Res.A.*, 371 (3):434–438, 1996.

[13] M. MORTENSON. *Geometric Modeling.* Wiley and Sons, 1997.

[14] M. MANTYLA. *Solid Modelling.* Computer Science Press, 1995.

[15] AA. VV. *Open Cascade, Foundation classes. User's Guide.* EADS Matra Datavision, 2001.

[16] M. K. Dalheimer. *Programming with Qt.* O'Reilly, 1999.

[17] AA. VV. *Open Sources: Voices from the Open Source Revolution.* Chris DiBona, Sam Ockman, Mark Stone, 1999.