MESH DATA STRUCTURES

MODELING AND ANIMATION, LAB 1

Mark Eric Dieckmann Erik Junholm Emma Broman Robin Skånberg

Wednesday 22nd March, 2023

Abstract

We learn how to work with mesh data structures and look at how much memory they use. We see that it is sometimes important to sacrifice storage space for more efficient neighborhood search capabilities. We also learn how to classify manifolds according to the Euler-Poincaré equations. Physical attributes such as normals, curvature, area and volume are inspected and discrete formular derived and implemented.

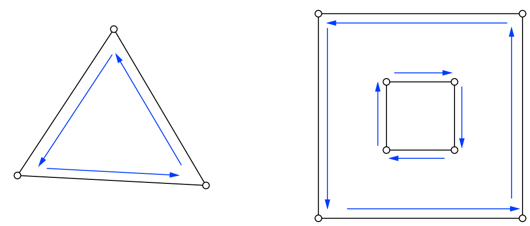

Figure 1: Loop counting for polygons. Left a triangle with one loop, right a polygon with two loops.

1 Introduction

There exists many different data structures for polygon models. Most common are those, which are based on triangles or quadrilaterals. In this lab we will focus on triangles, the far most common way of representing meshes. Triangles are the smallest (explicit) entities to span a plane, they are easy to work with and are well supported in graphics hardware. Triangle meshes are a form of boundary representation that separates topology and geometry. Remember the Euler-Poincaré formula

$$V - E + F - (L - F) - 2(S - G) = 0 \quad (1)$$

With V denoting number of vertices, E number of edges, F number of faces, L number of loops, S number of shells, and G is genus. A loop is a unique ring (along edges) in the mesh. For triangle meshes this simplifies loop counting a lot. Since there is exactly one loop within each face, see figure 1, the number of loops equals the number of faces, $L = F$. Furthermore we normally only work with one shell at a time ($S = 1$). With this we can simplify equation 1 to

$$V - E + F - 2(1 - G) = 0 \quad (2)$$

This allows for easy classification of the genus of a closed manifold mesh.

2 Mesh formats

In the following we will use the above abbreviations, and also assume an entity `Vertex` defined as such

```
struct Vertex
{
  float x,y,z;
};
```

We assume that the storage space for each float is 4 bytes. This is true for most platforms. In the simplest form a mesh can then be described by

Listing 1: Independent face list.

```
struct Face{
  Vertex v1, v2, v3;
};
struct Mesh{
  Face faces[F]; // geometry only
};
```

Where every face stores its corresponding vertices. It is simple to see that the memory footprint of this data structure is $3F \cdot \texttt{sizeof}(\texttt{Vertex}) = 36F$ bytes per mesh. It is also clear that this data structure has a large degree of redundancy, since many vertices will be duplicated.

If we get rid of the vertex redundancy we arrive at the indexed face set (note that the actual vertices in a face have been replaced by pointers to vertices).

Listing 2: Indexed face set.

```
struct Face{
  Vertex* v1, v2, v3;
};
struct Mesh{
  Vertex verts[V]; // geometry
  Face faces[F]; // topology
};
```

This data structure uses $V * \texttt{sizeof}(\texttt{Vertex}) = 12V$ bytes for the geometry and $3 * F * \texttt{sizeof}(\texttt{Vertex*}) = 12F$ for the topology[1]. It can be shown that the relationship between the number of vertices and the number of faces is

$$F \approx 2V \tag{3}$$

for a triangle manifold mesh. Using this relationship we calculate the memory usage to $18F$ bytes per mesh, roughly halving the size. This is considered a lower limit for data structures with random access to individual triangles.

The two above-mentioned mesh data structures are fast when it comes to linear traversal through triangles, for example when rendering. But consider neighborhood information; how can we access neighboring triangles from a given vertex? For the meshes in listing 1 and 2, we need to search through the whole face list and find all triangles containing this vertex. This is $O(F)$ operations, and linear might not be so bad, but if this needs to be done for every vertex it becomes $O(VF)$ which is quadratic in complexity. This has motivated a lot of research into alternative mesh data structures that trade memory consumption for faster neighborhood traversal. Half-edge, winged-edge, and directed edge, are examples of this.

[1]Assuming 4 bytes for the Vertex pointer.

2.1 Half-edge

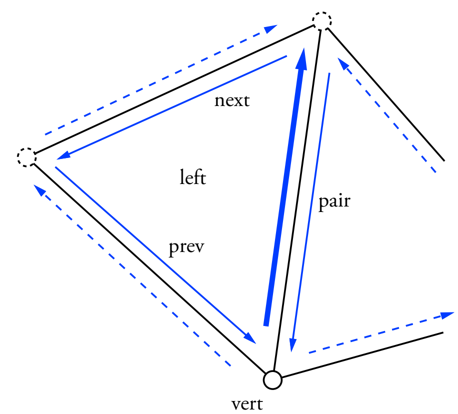

Figure 2: The half-edge data structure as seen from the bold half-edge.

The mesh formats considered so far have stored information about vertices and faces (triangles). To gain more efficient access to neighborhood, we may also add *edge* information to the mesh. One of the most common ways of doing this, is using the so-called half-edge data structure [4], see figure 2 and listing 3. It is called half-edge because every edge is "split" down its length, so that only information about the `left` face is stored explicitly. Information about the "right" face can be accessed through the half-edge's `pair`. To walk around the `left` face, you can use the `next` and `prev` pointers which will take you to the half-edges surrounding that face. Explicit information is indicated by blue lines in the figure, whereas dashed lines show implicit information which can be accessed through the pairs. Note the convention of using counter-clockwise orientation of faces.

Considering listing 3, we augment each `Vertex` with a half-edge pointer `edge`. Note that there are several possible half-edges connected to a single vertex, but it does not matter which half-edge we choose. In any choice, given a vertex, we can now access information about the topological neighborhood through this half-edge. Similarly, each `Face` is also augmented with a half-edge pointer `edge`, which can be any of the face's edges. For a triangle mesh, this structure needs

$$\begin{aligned} V * \texttt{sizeof}(\texttt{Vertex}) + 3F * \texttt{sizeof}(\texttt{Halfedge}) \\ + F * \texttt{sizeof}(\texttt{Face}) \\ \approx \frac{F}{2} * 16 + 3F * 20 + F * 4 = 72F \end{aligned}$$

bytes per mesh. Furthermore, it is limited to manifold meshes since every edge must have two faces. In order to represent non-closed geometry with borders we need to allow storage of empty faces, usually indicated by `NULL` pointers as `left` face. Note that this data structure contains redundancies. For example, since we are dealing with triangles we know that edge `prev` equals `next->next`. Another possible data reduction is to not store faces explicitly but instead letting every three edges implicitly define a triangle. However, this does not allow us to store additional face information like colors, etc.

Listing 3: Half-edge data structure. Note that this is a conceptual representation and does not correspond exactly with the lab code.

```
struct Face;
struct Vertex;

struct HalfEdge { // topology
  Vertex* vert;
  HalfEdge* next;
  HalfEdge* prev;
  HalfEdge* pair;
  Face* left;
};

struct Vertex { // geometry
  float x,y,z;
  HalfEdge* edge;
};

struct Face {
  HalfEdge* edge;
};

struct Mesh {
  Vertex verts[V];
  Face faces[F];
  Halfedge edges[3F];
};
```

3 Geometric Properties

Differential properties deal with how a surface changes and can for example be used to classify and/or modify the surface. Differential properties describe shape in a concise mathematical way, for example when we loosely say that a surface "is smooth", it really means that the second order differential property curvature is low.

Normals

The simplest of the geometric differentials is the normal vector which has the property of being perpendicular to a small local surface neighborhood. Approximating the triangle as sufficiently small we can use it as a support plane. Given counter-clockwise orientation of vertices in a triangle, when viewed from the outside of the manifold, we can construct a plane normal as the cross product of $(\mathbf{v}_2 - \mathbf{v}_1)$ and $(\mathbf{v}_3 - \mathbf{v}_1)$. This forms a vector with the desired properties: perpendicular to the plane spanned by the three vertices and pointing outwards.

$$\mathbf{n} = (\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1) \tag{4}$$

Evaluation of a shading model based on this normal and subsequent interpolation of the result leads

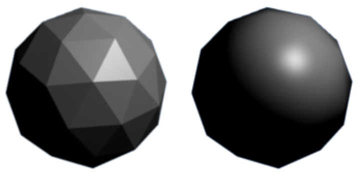

Figure 3: Flat and Phong shading.

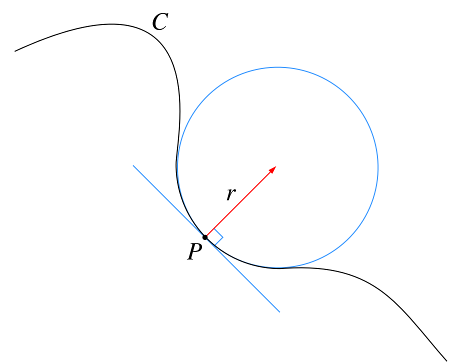

Figure 4: Osculating circle for a plane curve. Image from Wikipedia.

to, so-called, flat shading, and clearly shows the linear properties of the mesh (figure 3). By calculation of vertex-based normals, independent evaluation of the shading model and interpolation of the result across each triangles surface would result in a so-called Gouraud shading. The smoothest result is achieved by interpolation of the normal and fragment-based evaluation of the shading model. This approach is called Phong shading. [2]

To acquire per vertex normals there exist many techniques for averaging easy to calculate face normals, see for example [3]. The easiest variant is called mean weighted equally (MWE) and is defined as the normalized sum of the adjacent face normals.

$$\mathbf{n}_{v_i} = \widehat{\sum_{j \in N_1(i)}^{n} \mathbf{n}_{f_j}} \tag{5}$$

$N_1(i)$ is called the 1-ring neighborhood, and is simply all the faces sharing vertex v_i. For more examples of interpolated normals, like weighting according to triangle area, edge lengths, or incident angle, see [3]. Note that interpolated normals do not change the surface, they only affect lighting calculations; we can still see the linear outline of the smooth shaded sphere in figure 3.

Curvature

The next differential discussed is curvature. Curvature is currently one of the most important mesh quality measures and is used in many algorithms. Curvature describes the smoothness of the surface and how the normal at point p changes as we move the point along the surface. The curvature of a plane curve is inversely proportional to the radius of the osculating circle, see figure 4. In dimensions higher than 2D there exists different types of curvature; in this text we will discuss the two most frequent: *Gaussian curvature* and *mean curvature*. They can both be composed from the two principal curvatures κ_1, κ_2.

$$\begin{array}{ll} \text{Gaussian curvature} & K = \kappa_1 \kappa_2 \\ \text{Mean curvature} & H = \frac{1}{2}(\kappa_1 + \kappa_2) \end{array} \tag{6}$$

The principal curvatures are found as follows: let p be a point on the surface S. Consider all plane curves C_i on S passing through the point p on the surface. Every such C_i has an associated curvature κ_i given at p, as in figure 4. Of those curvatures κ_i, at least one is characterized as maximal κ_1 and one as minimal κ_2. For a unit sphere, or a circle, all curvatures are identically 1 (including the principal curvatures), which means that both Gaussian and mean curvature has magnitude of $\frac{1}{r}$.

Gaussian curvature is fairly straightforward to implement, and has a direct and intuitive formula:

$$K = \frac{1}{A}\left(2\pi - \sum_{j \in N_1(i)} \theta_j\right). \tag{7}$$

[2]In our application the lighting calculations are carried out by vanilla OpenGL, which implements either flat or Gouraud shading, depending on the developers configuration. Any shading calculation requires given normals. In our case vertex normals are required.

It is the deviation from 2π weighted by the area of the 1-ring neighborhood as shown in figure 5. A positive value means that the sum of the angles are less than 2π and the umbrella is pointy, i.e. the surface is non-smooth. A value of zero is equivalent to a plane or a surface that is flat along one direction. For a saddle point one of the values κ_1, κ_2 is negative and so is the curvature.

It is often important to distinguish between convex and concave curvature, and relate them to positive and negative magnitudes respectively. This can be accomplished by looking at the mean curvature. The mean curvature of a mesh can be found by looking at the gradient of the area of the mesh[2] :

$$H\mathbf{n} = \frac{\nabla A}{2A}. \tag{8}$$

The discrete version is:

$$H\mathbf{n} = \frac{1}{4A} \sum_{j \in N_1(i)} (\cot \alpha_j + \cot \beta_j)(\mathbf{v}_i - \mathbf{v}_j). \tag{9}$$

The area A is the total area of the 1-ring neighborhood.

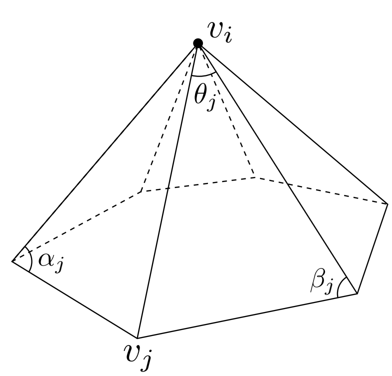

Figure 5: The 1-ring neighborhood of the discrete curvature operator.

The accuracy of the equations 7 and 9 can be improved by choosing the Voronoi area of the neighborhood for the division.

$$A_v = \frac{1}{8} \sum_{j \in N_1(i)} (\cot \alpha_j + \cot \beta_j)|(\mathbf{v}_i - \mathbf{v}_j)|^2 \tag{10}$$

If you are interested [1] contains a whole chapter on discrete curvature on meshes.

Area

Recall that any integral can be approximated by a Riemann sum and it follows that the area of a mesh is the sum of the areas of each individual face.

$$A_S = \int_S dA \approx \sum_{i \in S} A(f_i) \tag{11}$$

The area of the i:th face $A(f_i)$ is half the magnitude of the cross product of any two edges in the triangle, $\frac{1}{2}|(v2 - v1) \times (v3 - v1)|$. In this case the Riemann sum is exact, but this is not generally true.

Volume

To calculate the enclosed volume of a closed manifold mesh it is sufficient to know the faces of the mesh. The *divergence theorem* (alt. Gauss' theorem) relates the volume and surface integrals.

$$\int_S \mathbf{F} \cdot \mathbf{n} \, dA = \int_V \nabla \cdot \mathbf{F} \, d\tau \tag{12}$$

The theorem states that the surface integral of a vector field times the unit normal equals the volume integral of the divergence of the same vector field. As this is true for *any* vector field, we can choose a vector field with special properties. Assume that $\mathbf{F}$ has constant divergence, that is $\nabla \cdot \mathbf{F} = c$. Then the volume integral becomes

$$\int_V \nabla \cdot \mathbf{F} \, d\tau = \int_V c \, d\tau = c \int_V d\tau = cV \tag{13}$$

We can compute the volume (times a constant) with this integral and equation 12 links this integral to the surface integral. We can calculate the divergence with $\mathbf{F} = \bar{\mathbf{F}} = (x, y, z)$:

$$\begin{aligned} \nabla \cdot \bar{\mathbf{F}} &= \\ &= \nabla \cdot (x, y, z) \\ &= \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \cdot (x, y, z) \end{aligned} \tag{14}$$

$$\begin{aligned} &= \frac{\partial x}{\partial x} + \frac{\partial y}{\partial y} + \frac{\partial z}{\partial z} \\ &= 3 \end{aligned}$$

We note that our related surface integral will compute $3V$ ($c = 3$) and finally we develop a discrete formula by approximating the integral as a Riemann sum over each face:

$$\begin{aligned} 3V &= \int_V \nabla \cdot \bar{\mathbf{F}}\, d\tau = \int_S \bar{\mathbf{F}} \cdot \mathbf{n}\, dA \\ &\approx \sum_{i \in S} \bar{\mathbf{F}}(f_i) \cdot \mathbf{n}(f_i) A(f_i) \end{aligned} \tag{15}$$

Here we used $\bar{\mathbf{F}}(f_i)$ to denote the vector field evaluated at the i:th face (f_i), and similarly for the normal and area. At this point we have not yet decided where to evaluate the vector field, it is only specified that it should be done somewhere on the face. Remember that the face normal and area are constant over the face. The vector field is not though; but it can be shown that any point (on the face) will do in the limit. We chose the centroid of the triangle giving rise to

$$3V = \sum_{i \in S} \frac{(\mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3)_{f_i}}{3} \cdot \mathbf{n}(f_i) A(f_i), \tag{16}$$

where $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ are the vertices of the i:th triangle. The error of this approximation decreases as the area of the largest triangle goes to zero. Try to compose a formula that evaluates $\bar{\mathbf{F}}$ at other points as well and see if you can improve on the results.

4 Assignments

Download the code from the course page if you have not already gotten it.

4.1 Grading

The assignments marked as (3) are mandatory to complete in order to pass the lab. You will need some of that code for the later labs. Completing assignment (4) gives you the grade 4 and completing assignments (4,5) gives you a grade 5.

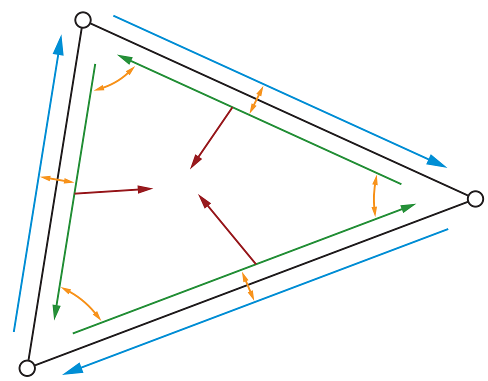

Figure 6: An incomplete halfedge mesh consisting of one triangle.

4.2 Assignments

Start with **implementing the half-edge mesh** and use your mesh for all subsequent tasks. If you get stuck please check the readme of the repository for the listed lab notes.

Implement the half-edge mesh (3)

The simple mesh data structure that you already have is very inefficient when accessing neighbour data. This is for example necessary when calculating per-vertex normals. Open the file `HalfEdgeMesh.cpp` and study the functions `AddVertex`, `AddFace` and `AddHalfEdgePair`. Your task is to implement the `AddFace` function. For a working example, study the implementation in `SimpleMesh`.

If we allow general polygons in an open mesh (non-closed manifold), the implementation is fairly complex. We restrict ourselves to closed triangle meshes which simplifies the implementation.

Start by looking at figure 6; we have a triangle and its incomplete half-edge mesh. The “inner loop” edges (CCW orientation) are drawn in green, the “outer loop” (CW orientation) is drawn in blue. The connections between edges, that is the `next`, `prev` and `pair` pointers, are drawn in orange. Finally the inner edges are connected to the face enclosed inside, pointers drawn in red. The outer edges are not yet connected and have no assigned faces.

You will add code to `AddFace` that does the following for each triangle:

1. Add its three vertices to the data structure with `AddVertex`. You get their indices back. This function removes redundant vertices by only inserting unique values.

2. For each edge in the triangle, add a half-edge pair (`AddHalfEdgePair`). You get two indices back. This function additionally:

   - Removes redundant half-edges by only inserting unique edges
   - Connects the half-edge pair
   - Connects each half-edge to its origin vertex
   - Connects each vertex to one of its edges

3. Connect the inner ring of edges. That is the `next` and `prev` indices. Note the convenience functions `e()`, `f()` and `v()` that you can use instead of `mEdges.at()`, `mFaces.at()` and `mVerts.at()` or their bracket counterparts (e.g. `mEdges[]`).

4. Create a face, connect it to one of the edges. Calculate its normal vector and push the face (`push_back`) on the `mFaces` vector.

5. Connect the inner edges to the newly created face.

Since we have assumed that the mesh is closed, all loops will eventually be inner loops, therefore we need not care about the outer loops, it is enough to set the half-edge pointers for the inner loop. Think about this and make sure you understand why.

All geometric data in the HalfEdgeMesh is automatically initialized to `UNINITIALIZED`; when you connect the edges and faces you will update these values. A simple validation of the data structure can be done by trying to load a mesh. When a Geometry object is loaded the function `Initialize` is automatically run. For the half-edge mesh this function tries to validate the mesh, via the `Validate` member function. This function proceeds to see that all the mesh data is set to something else than `UNINITIALIZED`. It is not a bulletproof test but it will catch many errors. Try to load a simple mesh such as the cube, which is small enough so that you can inspect the geometry "by hand" in a text file viewer (obj is a text based file format).

Implement neighbor access (3)

Implement the functions `FindNeighborFaces` and `FindNeighborVertices`. See `SimpleMesh.cpp` for a working example. Note that the `SimpleMesh.cpp` does not store any neighborhood information which requires looping over the mesh. The functions in `HalfEdgeMesh.cpp` can be written more efficiently.

Calculate vertex normals (3)

When the mesh is built in the previous assignment it calculates one face normal as each face is added. Using the neighbour information provided by your half-edge mesh it is simple to implement fast calculation of per-vertex normals. This step is performed in the general update function (`Update`).

In order for this we need fast access to our neighbors. `FindNeighborFaces` is used to find the neighboring faces around a vertex, which is exactly what we need in order to calculate the vertex normals. Implement equation 5 in the `VertexNormal` function and store the resulting normal in each Vertex struct.

Calculate surface area of a mesh (3)

Find the method `Area` in `HalfEdgeMesh.cpp` and add code that calculates and returns the area of your mesh. In the sample data you will find three spheres with the radii 1.0 , 0.5 and 0.1. A sphere with the radius $r = 1.0$ has an area of $A = 12.56637$ and a volume of $V = 4.18879$. The spheres are stored in the files `sphere_1.0.obj`, `sphere_0.5.obj` and `sphere_0.1.obj`.

Calculate volume of a mesh (3)

Find the method `Volume` in `HalfEdgeMesh.cpp` and add code that calculates and returns the volume of your mesh. Again you might find the sphere meshes in `Objs` useful.

Implement and visualize Gaussian curvature (3)

The function `VertexCurvature` in `SimpleMesh` implements a basic version of Gaussian curvature. Since the two mesh classes share a similar interface you can copy this code into your HalfEdgeMesh. In the directory `Data/Objs` you will find three spheres with the radii 1.0 , 0.5 and 0.1. These are helpful to visualize and inspect the result. Discuss your findings when visualizing the curvature - what are the expected analytical curvatures of the spheres knowing their radii? How well does the Gaussian curvature estimate this?

Implement and visualize mean curvature (4)

Gaussian curvature cannot differentiate between concave and convex curvature. Comment out your Gauss curvature code and implement mean curvature (equation 9) instead. Use the Voronoi area for area estimation as described in equation 10. `Util.h` contains a function for stable computations of cotangents (given three vertices) called `Cotangent`. Compared to the Gaussian curvature in the previous assignment - how well does the mean curvature estimate the curvature of the spheres?

Classify the genus of a mesh (4)

Implement a method for calculating the genus of your half-edge mesh. Open the file `HalfEdgeMesh.cpp` and write your code in the `Genus` function. Equation 2 only holds for closed manifold meshes with one shell, and for now we can assume that $S = 1$.

Compute the number of shells (5)

By default the function `Shells` returns 1. Implement a way of finding the number of shells. Any one shell can be "tagged" by means of a flood fill.

push first vertex $\rightarrow$ vertexQueueSet
while vertexQueueSet $\neq$ empty **do**
  $v \leftarrow$ vertexQueueSet.pop
  push $v \rightarrow$ vertexTaggedSet
  for all v_i in findNeighborVertices(v) **do**
    if $v_i \notin$ vertexTaggedSet **then**
      push $v_i \rightarrow$ vertexQueueSet
    end if
  end for
end while

Use the new information to compute the genus of a mesh with more than one shell. **Tip**: The standard template library (stl) data structure std::set and std::map might useful.

5 Acknowledgements

The lab scripts and the labs were originally developed by Gunnar Läthén, Ola Nilsson, Andreas Söderström and Stefan Lindholm.

References

[1] Mario Botsch, Mark Pauly, Leif Kobbelt, Pierre Alliez, Bruno Levy, Stephan Bischoff, and Christian Rössl. Geometric modeling based on polygonal meshes. In *SIGGRAPH Course Notes*, 2007.

[2] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 317–324, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[3] Shuangshuang Jin and Robert R. Lewis. A comparison of algorithms for vertex normal computation. *The Visual Computer*, 21:71–82, 2005.

[4] Martti Mäntylä. *An introduction to solid modeling*. Computer Science Press, Inc., New York, NY, USA, 1987.