Modeling and Animation (TNM079)
Lab 1: Mesh data structures

Gunnar Johansson   Ola Nilsson   Andreas Söderström

March 29, 2007

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 fomulae derived and implemented.

1 Introduction

There are many different formats for polygon models, sometimes called polygon soup. Most common are formats 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) entity 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 (B-rep) that separates topology and geometry. Remember the Euler-Poincaré formula

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

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, $L = F$.

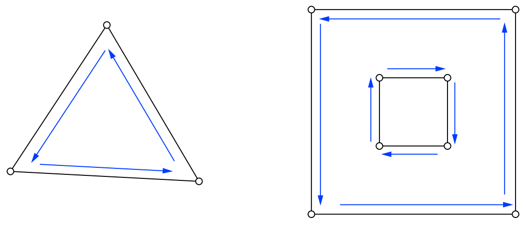

Figure 1: Loop counting for polygons. Left a triangle with one loop, right a polygon with two loops.

Furthermore we normally only work with one shell ($S = 1$).

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

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 triangle 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 $3*F*$`sizeof(Vertex)` $= 36F$ bytes per mesh. It is also clear that this data structure has a large degree of redundance, since many vertices will be duplicated.

If we get rid of the vertex redundance we get another mesh structure

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 $V *$ `sizeof(Vertex)` $= 12V$ bytes for the geometry and $3 * F *$ `sizeof(Vertex*)` $= 12F$ for the topology[1]. It can be shown that the relationship between the number of vertices and 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 abovementioned 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, directed edge, are examples of this.

[1]Assuming 4 bytes for the Vertex pointer.

2.1 Half-edge

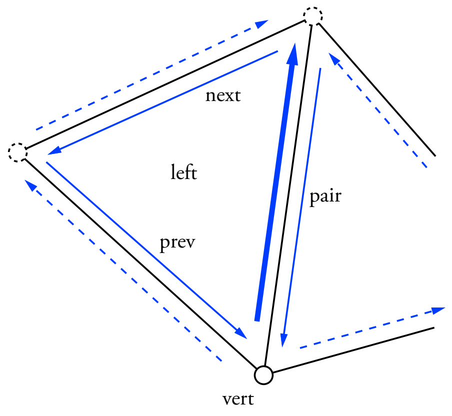

Figure 2: The half-edge data structure as seen from the bold half-edge.

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 [3], 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

$$V * \texttt{sizeof}(\texttt{Vertex}) + 3F * \texttt{sizeof}(\texttt{Halfedge}) + F * \texttt{sizeof}(\texttt{Face}) \approx \frac{F}{2} * 16 + 3F * 24 + F * 4 = 84F$$

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.

```
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];
};
```

2.1.1 Non-closed half edge meshes

3 Geometric properties for meshes

Differential properties deal with how a surface changes. 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.

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

This is however only enough to do flat shading, and

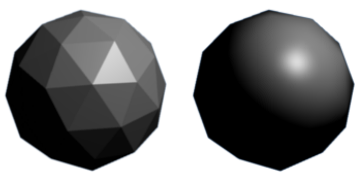

Figure 3: Flat and smooth shading.

clearly shows the linear properties of the mesh, figure 3. By assigning normals at vertices and interpolate across the triangle we can squeeze some extra smoothness out of the linear surface. There exist many techniques for doing so, see for example [2], the easiset one being mean weighted equally (MWE) 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 [2] for an exhaustive comparison. Note that interpolated normals do not change the surface, they only effect lighting calculations; see the linear outline of the smooth shaded sphere in figure 3.

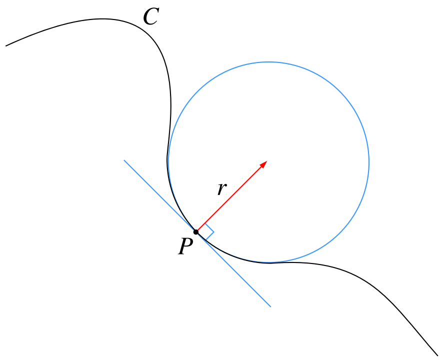

Figure 4: Osculating circle for a plane curve. Image from Wikipedia.

The next differential discussed is curvature. Intuitively curvature describes the smoothness of the surface, or how the normal at point p changes as we move the point along the surface. In this context the curvature of a plane curve is inversely proportional to the radius of the osculating circle, see figure 4. We are interested in the concept of *mean curvature* defined as the mean of the principal curvatures κ_1, κ_2

$$H = \frac{1}{2}(\kappa_1 + \kappa_2) \tag{6}$$

The principal curvatures are found as follows: let p be a point on the surface S. Consider all 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. Of those curvatures κ_i, at least one is characterized as maximal κ_1 and one as minimal κ_2. For a sphere or a circle all curvatures are equal including the principal curvatures which means that the mean curvature has magnitude of $\frac{1}{r}$. It has lately been shown [1] that the mean curvature of a mesh can be found by

$$H\mathbf{n} = \frac{\nabla A}{2A} \tag{7}$$

The discrete version is, through a somewhat cumbersome derivation

$$H\mathbf{n} = \frac{1}{4A} \sum_{j \in N_1(i)} (\cot \alpha_j + \cot \beta_j)(\mathbf{v}_i - \mathbf{v}_j) \tag{8}$$

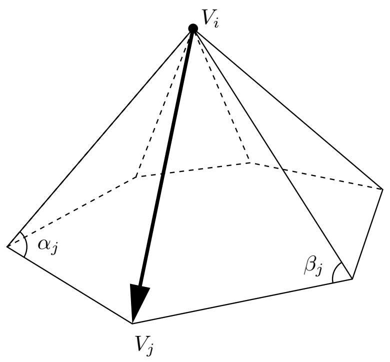

Figure 5: The 1-ring neighborhood of the discrete curvature operator.

The area A is the total area of the 1-ring neighborhood. This approximation can be further improved by choosing the voronoi area of the neighborhood for the division.

$$A_{voronoi} = \frac{1}{8} \sum_{j \in N_1(i)} (\cot \alpha_j + \cot \beta_j)|(\mathbf{v}_i - \mathbf{v}_j)|^2 \tag{9}$$

3.1 Integrals involving meshes

Recall that any integral can be approximated by a Riemann sum and we see that the area of a mesh is readily computed as the sum of the areas of each individual face.

$$A_S = \int_S dA \approx \sum_{i\in S} A(f_i) \tag{10}$$

Where $A(f_i)$ is the area of the i:th triangle, which 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 approximation is exact, but this is not generally true.

3.1.1 Volume

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

$$\int_S \mathbf{F} \cdot \mathbf{n}\, dA = \int_V \nabla \cdot \mathbf{F}\, d\tau \tag{11}$$

The therorem states that the surface integral of a vector field $\mathbf{F}$ times the unit normal equals the volume integral of the divergence of the same vector field. Noting that we are free to choose whatever vector field we like we assume a vector field with constant divergence, $\nabla \cdot \mathbf{F} = c$. Then the volume integral becomes

$$\int_V \nabla \cdot \mathbf{F}\, d\tau = \int_V c\, d\tau = c \int_V d\tau = cV \tag{12}$$

We see that we can compute the volume (times a constant) with this integral and therefore also with the corresponding surface integral in equation 11. We choose $\mathbf{F} = (x, y, z)$, and calculate the divergence as

$$\begin{aligned} \nabla \cdot \mathbf{F} &= \\ &= \nabla \cdot (x, y, z) \\ &= (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}) \cdot (x, y, z) \\ &= \frac{\partial x}{\partial x} + \frac{\partial y}{\partial y} + \frac{\partial z}{\partial z} \\ &= 3 \end{aligned} \tag{13}$$

We note that our related surface integral will compute $3V$ and finally we develop a discrete formula by approximating the integral as a Riemann sum.

$$3V = \int_V \nabla \cdot \mathbf{F}\, d\tau = \int_S \mathbf{F} \cdot \mathbf{n}\, dA \quad \approx \quad \sum_{i\in S} \mathbf{F}(f_i) \cdot \mathbf{n}(f_i) * A(f_i) \tag{14}$$

In this approximation we are free to decide where to evaluate the vector field at each triangle, $\mathbf{F}(f_i)$, a natural choice is to evaluate the function at the centroid of the triangle giving

$$3V = \sum_{i\in S} \frac{(\mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3)_{f_i}}{3} \cdot \mathbf{n}(f_i) * A(f_i) \tag{15}$$

Where v_1, v_2 and $v3$ are the vertices of the i:th triangle. The error of this approximation decreases as the area of the largest triangle goes to zero.

4 Assignments

Start by downloading the code base using svn[2]

```
mkdir moacode;
cd moacode;
svn co https://gg.itn.liu.se/svn/courses/moa/lab1 mylab1;
```

Use `tnm079` for both username and password. This will check out (i.e. download) the code in directory lab1 located at the url into a directory called mylab1. You should be able to compile and run the code straight away.

```
cd mylab1;
make;
./main;
```

This will start the viewer, and you will see a rotating square. The basic controls are:

```
w: Accelerate camera forward
s: Accelerate camera backwards
a: Accelerate camera to the left
d: Accelerate camera to the right
```

[2]More info about Subversion at `http://subversion.tigris.org`

```
W: Accelerate camera upwards (i.e. shift+w)
S: Accelerate camera downwards (i.e. shift+s)

1: Load bunny mesh
2: Load the cube as half edge and validate

space: Stop all camera motion

Looking:
Left click with your mouse in the GUI
window and move the mouse to change
where you are looking.
```

Examine the function `keyboardFunc` in `GUI.cpp` to find out more about what the different keys correspond to and how to add your own input. By pressing `1` you can add a predifined mesh using the SimpleMesh datastructure. This also calculates normals and curvature which is *slow* for big meshes.

4.1 Grading

Of the following assignments the ones marked with a single star (*) are mandatory to complete in order to pass the lab, grade 3. To achieve grade 4 you need to additionally complete **all** assignments marked with two stars (**). For grade 5 you need to pass the grade 4 requirements and additionally complete **one** of the assignments marked with (***).

4.2 Assignments

Start with **implementing the halfedge mesh** and use your mesh for all subsequent tasks.

Implement the half edge mesh *

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`, `addTriangle` and `addHalfEdgePair`. Your task is to implement the `addTriangle` 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. However, we restrict ourselves to closed triangle meshes, which simplifies the implementation drastically.

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 `addTriangle` that does the following for each triangle:

1. Add its three vertices to the data structure with `addVertices`. 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 their 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. I.e. the `next` and `prev` indices.
4. Create the face, connect it to one of the edges and push it back on the `mFaces` vector.
5. Connect the inner edges to the newly created face.

Since the mesh is assumed to be closed, all loops will eventually be inner loops, therefore it is enough to set the half-edge pointers for the inner loop.

All geometric data in the HalfEdgeMesh is initilized to `UNINITIALIZED`; when you connect the edges and faces you will update these values. A simple validation of the datastructure can be done

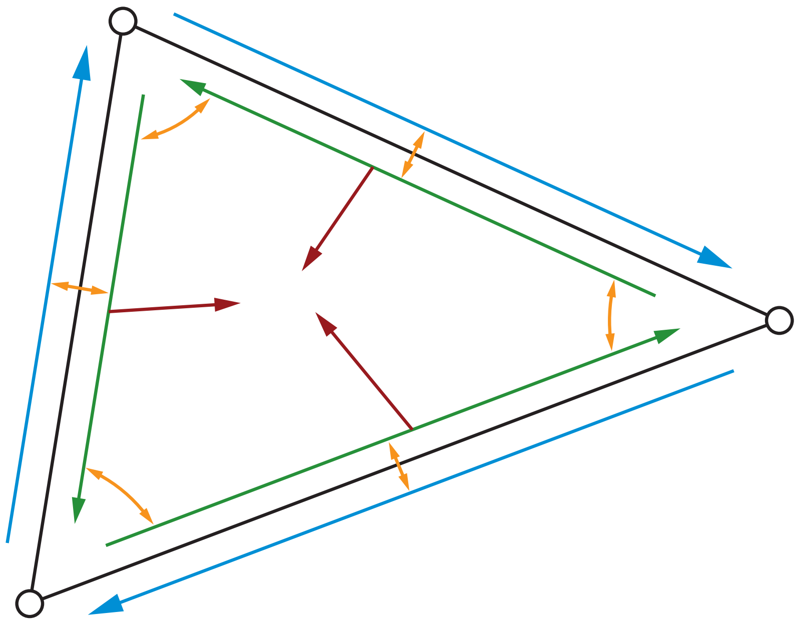

Figure 6: An incomplete halfedge mesh consisting of one triangle.

by trying to load a mesh in `GUI.cpp` and run the member function `validate`. 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. We have prepared a cube mesh to be loaded and validated when you press '2' in the viewer.

Calculate normals *

When the mesh is built in the previous assignment it contains no normal information, only connectivity. Using the neighbour information provided by your half edge mesh it is simple to implement fast calculation of per-vertex normals.

First calculate the face normals, this is done in the function `calculateFaceNormals` in `HalfEdgeMesh.cpp`. Remember that the vertices in each triangle are oriented counter clock wise. Try loading the cube again! Now the shading should look better? If not then there is probably an error somewhere.

Next implement the `findNeighbourTriangles` in `HalfEdgeMesh.cpp`. See `SimpleMesh.cpp` for a working example. This function is used to find the 1-ring neighborhood around a vertex, which is exactly what we need in order to calculate the vertex normals. Use this in the `calculateVertexNormals` function in the file `HalfEdgeMesh.cpp`. With access to every face in the 1-ring you also have access to their normals. Implement equation 10 and store the resulting normal in each Vertex struct. You need to add a member variable for this.

Calculate surface area of a mesh *

Find the method `area` in `HalfEdgeMesh.cpp` and add code that calculates and returns the area of your mesh. In the directory `Objs` you will find three spheres with the radii 1.0 , 0.5 and 0.1 that you may find helpful. The spheres are stored in the files `sphere1.0.obj`, `sphere0.5.obj` and `sphere0.1.obj`.

Calculate volume of a mesh *

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.

Classify genus of a mesh **

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 so you need to find a way of asserting these conditions. Alternatively you can calculate the number of disconnected (closed) compononents. Closed means that every edge in the mesh is bordering exactly two faces and is properly initialized. The number of shells can be found by means of flood fill.

```
add first vertex to vertexQueue
while vertexQueue != empty do
  $v$ = vertexQueue.pop()
  insert $v$ into vertexSet
  for all tris in findNeighbourTriangles(v, ...) do
    for all $v_i$ in tri do
      if $v_i \in$ vertexSet then
        do nothing
      else
        push $v_i$ onto vertexQueue
      end if
    end for
  end for
end while
```

Comparing the number of vertices in the vertexSet with the total number of vertices in the mesh should tell you something... Tip: The standard template library (stl) data structures std::queue and std::set implement a FIFO queue, and a unique set respectively.

Convert quads to triangles **

In the file `ObjIO.cpp` you will find the method `splitQuad`. This method will be called if the input file reader detects that a quad has been read. The method recieves an input stream and two vector references as arguments. The stream contains the four indices to the vertices in the quad. Use this information to split the quad into two triangles and write the vertex indices for each of the triangles into `tri1` and `tri2` respectivly. There will always be two ways of splitting a quad to triangles; motivate your choice.

Extend half edge mesh to handle non-closed surfaces***

In order to represent non-closed surfaces with boundaries, you need to correctly set the outer half-edge loop after adding a triangle. We quickly see that there are 4 different cases, depending on the number of boundary edges.

- The new triangle has 3 border edges. This triangle is the entire mesh, see figure 6
- The new triangle has 2 border edges.
- The new triangle has 1 border edges.
- The new triangle has 0 border edges. This corresponds to a triangle that fills a hole in the mesh

We can easily test which border edges a triangle has since the `addHalfEdgePair` returns a boolean to tell us whether the edges already existed or not. Consider case one above (3 border edges), looking at figure 6 we see that all we need to do in order to merge the mesh is to connect the outer edge loop (blue edges) and mark the border. That corresponds to setting `e->next = e->pair->prev->pair`, similiarly for `prev`, and `e->face=BORDER`. Now we can walk around vertices again, all members are set and we have the additional descriptor `BORDER` to mark the boundary.

If the triangle is connected to one or more other triangles then it gets more complicated, but if we focus on the addition of a single border edge at a time the concept generalizes. We have a valid boundary mesh (which may be empty), and we have new triangle with zero or more boundary edges[3]. For each boundary edge merge it into the boundary of the mesh by setting `next`, `prev`, and `face=BORDER`. Now the question is, what edges should it connect to?

Note that while walking the mesh at this stage you can't use the `prev` and `next` indices for edges that have `face=UNITIALIZED`, but you can of course use their `pairs`. Implement the `mergeBoundaryEdge` function for this purpose and do the merge last in `addTriangle`.

Present a lower memory bound for the halfedge data structure***

How much is it possible to compress the topology of the halfedge data structure? We gave a few pointers in the text, and the question is how much information can you remove and still retain constant time neighbour access. This is a theroetical assignment, but you are encouraged to implement your findings. Especially interesting is to compare the actual speed of a topology-compressed halfedge against the version presented in listing 3.

Calculate curvature for a vertex ***

Use the neighbour information provided by your half edge mesh to implement fast calculation of pervertex curvature. Open the file `HalfEdgeMesh.cpp` and enter your code into the `curvature` function. Store the result in the `mCurvatureArray` member varuable (found in `HalfEdgeMesh.h`. Curvature value nr X in `mCurvatureArray` should correspond to vertex nr X in the vertex array. In the directory `Objs` you will find three spheres with the radii 1.0 , 0.5 and 0.1 that you may find helpful. The spheres are stored in the files `sphere1.0.obj`, `sphere0.5.obj` and `sphere0.1.obj`.

[3]We assume that the inner edges already are connected.

5 Examination

In order to pass the lab you will have to **present your results to a lab assistant**. You will also need to **hand in a lab report together with your code**. The report is required in order to recieve a grade on the lab. Though you work in pairs **the report and your grade is individual**. The lab report should contain a short description as to how you solved each assignment and a motivation of why you chose to solve the task the way you did. Your report and code should be sent to **Andreas.Soderstrom@itn.liu.se**

You are required to pass all single star (*) assignments before your next lab since you will reuse some of your code in later labs.

References

[1] 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.

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

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