LAB REPORT: LAB 1

TNM079, MODELING AND ANIMATION

Johan Beck-Norén
johbe559@student.liu.se

Thursday 2nd May, 2013

Abstract

This is the first lab in a series of six labs in the course TNM079 Modeling and Animation given at Linköping University. For this lab a half-edge mesh structure was implemented. A few basic mesh calculations and operations were implemented and performed using the half-edge mesh structure such as mesh curvature, vertex normals, mesh area and mesh volume in order to examine any gain in using it as a mesh structure. The results show that although the half-edge mesh structure uses more memory than a simple mesh and is a bit complex to implement, it is considerably more efficient in performing mesh operations that require access to vertex neighbourhoods.

1 Introduction

1.1 Triangle mesh structure

There are many different structures for storing mesh data and for defining a mesh. The most commonly used mesh format is based on triangles. A triangle is defined by three vertices, which together with a normal direction defines a plane shared by the triangle polygon. The information about the vertices' coordinates are usually stored in a container of some sort, often referred to as a vertex list. The information about which combination of vertices that are used to define a triangle is also stored in a container, often referred to as a triangle list. These lists are used to define the mesh's geometry and topology as a form of boundary representation. A structure like this means that if we, for a vertex, want to find it's neighbouring vertices or which faces the vertex belongs to we have to iterate through the vertex list or triangle list, which is an operation that takes linear time, $O(n)$. However, if we wish to perform this neighbourhood search for every vertex in a mesh structure it becomes a problem of time complexity $O(n^2)$.

This is one of the main reasons why we are looking at alternative mesh structures, such as the half-edge mesh structure that sacrifices memory usage in favour of neighbourhood access and search performance.

1.2 Half-edge mesh structure

As described above, a triangle mesh structure uses a vertex list and a triangle list to define the mesh's topology and geometry. For a half-edge mesh structure we store the same information as we do in a triangle mesh structure, with the addition of edge information. This is to enable the use of a more efficient method of neighbourhood access. Consider "splitting" an edge down its length, creating a half edge (figure 1). Each half-edge only explicitly store information about

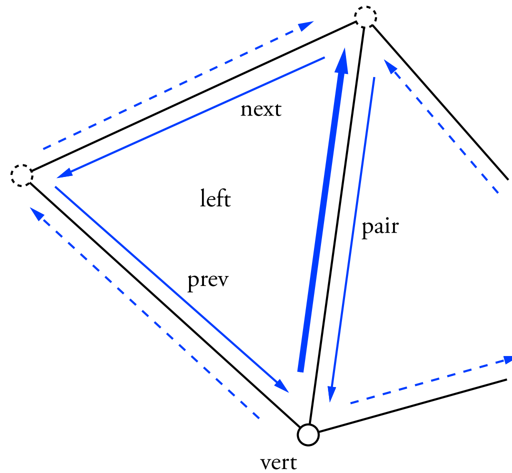

Figure 1: Half edge mesh structure. Explicit information in bold blue lines, implicit information via pair access in dotted blue lines.

the *left* face. This information include pointers *prev* and *next* used to traverse the half edges surrounding the left face in a counter-clockwise fashion, pointer to a vertex used to access the half edge and lastly a pointer the half edge's *pair* which is used to access the *right* face. We can now, given a vertex, quickly access the neighbouring topology through the half edge structure described above.

For this lab we constrain ourselves by assuming the meshes used are of the closed manifold triangle type. All methods mentioned in this report can be found in the file HalfEdgeMesh.cpp.

2 Assignments

This section describes a few geometric properties of meshes for which the operations and calculations might benefit from using a half-edge mesh structure. As described in the introduction of this report, structures using only lists triangle- and vertex-lists for storing mesh topology information perform poorly when accessing neighbourhood information. A situation where we could benefit from fast neighbourhood access is for example when calculate per-vertex normals.

2.1 Implementing the half-edge mesh

In order to create a half-edge mesh structure for a closed manifold triangle mesh we have to associate vertex-, face- and half-edge pointers to each other correctly. The implementation was written in the methods AddFace, using the already implemented methods AddVertex and AddHalfEdgePair in the file HalfEdgeMesh.cpp given in the lab. The following steps are performed for each triple vertices defining a triangle;

1 Add the three vertices to the vertex list using the method AddVertex.

2 Use the three vertices to define the three edges defining the triangle. Generate and connect a half-edge pair for each edge using the method AddHalfEdgePair. This method also associates each generated half-edge with its origin vertex, as well as connecting each vertex to one of its edges.

3 For each generated half-edge belonging to the left face, point it's *prev* and *next* pointers to the correct corresponding half-edges in a counter-clockwise orientation.

4 Create a face. Calculate its face normal and connect the face with a half-edge belonging to the *left* face. Connect the face with the three half-edges from the previous step.

As we stated earlier, we are assuming the mesh used is a closed manifold mesh. This means that each edge is connected to exactly two faces. We can therefore ignore the *right* face in the steps above, since that face will be caught as a *left* face in the algorithm eventually.

2.2 Implementing neighbour access

As mentioned earlier, more efficient neighbourhood access is one of the main reasons for implementing a half-edge mesh structure. Two types of neighbourhood access has been implemented; one method for finding a vertex's neighbouring vertices and another method for find a vertex's neighbouring faces. These were implemented in the methods FindNeighborVertices and FindNeighborFaces respectively.

2.2.1 Finding neighbour vertices

Given a vertex pointer we want to find all the neighbour vertices to that vertex. We accomplish this by traversing around the given vertex using the half-edge structure.

Algorithm 1 Find neighbour vertices of vertex $vertexIn$

1: $edgeCurrent \leftarrow vertexIn.edge.next$
2: $edgeEnd \leftarrow edgeCurrent$
3: **repeat**
4: $\quad foundVertices.append(edgeCurrent.vertex)$
5: $\quad edgeCurrent \leftarrow edgeCurrent.next$
6: $\quad edgeCurrent \leftarrow edgeCurrent.pair$
7: $\quad edgeCurrent \leftarrow edgeCurrent.next$
8: **until** $edgeCurrent = edgeEnd$

2.2.2 Finding neighbour faces

To find a vertex's neighbour faces we take a similar approach as when finding neighbouring vertices.

Algorithm 2 Find neighbour faces of vertex $vertexIn$

1: $edgeCurrent \leftarrow vertexIn.edge$
2: $edgeEnd \leftarrow edgeCurrent$
3: **repeat**
4: $\quad foundVertices.append(edgeCurrent.face)$
5: $\quad edgeCurrent \leftarrow edgeCurrent.prev$
6: $\quad edgeCurrent \leftarrow edgeCurrent.pair$
7: **until** $edgeCurrent = edgeEnd$

2.3 Calculating vertex normals

In order to calculate a vertex normal, we need access to all the faces the vertex belongs to. In the previous section, we talked about fast access to the neighbourhood around a vertex, and this really comes into play now. Although there are many ways to average face normals, we use a simple technique called mean weighted equally (MWE). This technique comprises of summing up all surrounding face normals from faces containing the vertex, and then normalize the resulting vector (equation 1). $N_1(i)$ is sometimes referred to as the *1-ring neighbourhood* of the i:th vertex. In this case it corresponds to all the faces containing vertex v_i, see figure 2.

$$\vec{n}_{v_i} = \sum_{j \in N_1(i)}^{\hat{n}} \vec{n}_{f_j} \tag{1}$$

Using our now implemented method FindNeighborFaces we can quickly gather the data we need for this operation. The resulting normal vector is stored in the Vertex struct. Vertex normal calculations are implemented in the method VertexNormal.

2.4 Calculating surface area of a mesh

Let us recap a little. In most cases, a polygon mesh is an approximation of a shape. A polygon mesh can for example resemble the shape of a sphere, but no matter how high the polygon count is it will still never become a completely round sphere since it still consist of individual polygons. Keeping this in mind, we realize that to calculate the surface area of a mesh we simply need to sum the surface area of each individual face on the mesh. This also follows that an integral can be approximated discreetly by a Riemann sum of the surface areas of each individual face. In equation 2, $A(f_i)$ is the surfaces area of the i:th face. The area of a triangle face is calculated by taking half the magnitude of the cross product between any two edges of the triangle.

$$A_S = \int_S dA \simeq \sum_{i \in S} A(f_i) \tag{2}$$

This is implemented using the quick neighbourhood access of the half-edge mesh structure.

Algorithm 3 Surface area of mesh

1: $areaSum \leftarrow 0$
2: **for** all faces in mesh **do**
3: $\quad edge \leftarrow face_i.edge$
4: $\quad v1 \leftarrow edge.vertex1$
5: $\quad v2 \leftarrow edge.next.vertex2$
6: $\quad v3 \leftarrow edge.prev.vertex3$
7: $\quad areaSum \stackrel{+}{=} \frac{1}{2} LengthOfVector(CrossProduct(v2 - v1, v3 - v2))$
8: **end for**

The area calculations are implemented in the method Area.

2.5 Calculating volume of a mesh

To calculate the volume of a mesh, we use an approach similar to when we calculated the surface area of a mesh. The volume integral normally used can in this case be replace with a Riemann sum over each face by using *Gauss' theorem* (equation 3). Gauss' theorem relates the volume and surface area integrals by stating that the surface integral of a vector field times the unit normal gives the volume integral of the divergence of the same vector field.

$$\int_S \vec{F} \cdot \vec{n} dA = \int_V \nabla \cdot \vec{F} d\tau \tag{3}$$

Because this holds true for any vector field, we choose a vector field $\vec{F}$ with constant divergence, e.g. $\nabla \cdot \vec{F} = c$ where c is some constant. Insertion into the right hand side of equation 3 gives us

$$\int_V \nabla \cdot \vec{F} d\tau = \int_V c d\tau = c \int_V d\tau = cV. \tag{4}$$

After choosing a suitable vector field $\vec{F}$ and then approximating the volume integral as a Riemann sum we end up with

$$3V = \sum_{i \in S} \frac{(\vec{v}_1 + \vec{v}_2 + \vec{v}_3)_{f_i}}{3} \cdot \vec{n}(f_i) A(f_i). \tag{5}$$

The fraction refers to the centroid of the i:th face, and $A(f_i)$ is the surface area of the i:th face.

2.6 Implementation and visualization of Gaussian curvature

The curvature of a mesh basically describes how smooth the mesh is. More precisely it describes how much the normal for a point on the mesh changes as we move the point along the surface of the mesh. The *Gaussian curvature* K is multiplicatively dependent on the two principle curvatures κ_1 and κ_2.

$$K = \kappa_1 \kappa_2 \tag{6}$$

The principal curvatures are defined as the maximal and minimal curvatures passing through any given point on the mesh surface. K is calculated by equation 7 where A is the area of the 1-ring neighbourhood seen in figure 2.

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

The code for Gaussian curvature was already implemented in the files provided and works well with the half-edge mesh structure, since the implementation uses the method FindNeighborVertices.

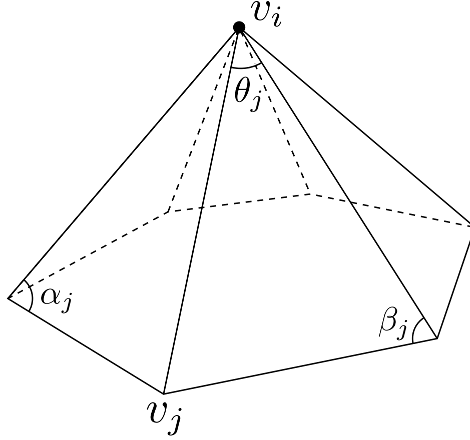

Figure 2: 1-ring neighbourhood for vertex v_i used in discreet curvature calculations.

2.7 Implementation and visualization of mean curvature

Mean curvature, unlike Gaussian curvature, depends additively on κ_1 and κ_2. The mean curvature H is defined as

$$H = \frac{1}{2}(\kappa_1 + \kappa_2). \tag{8}$$

H can be found by looking at the gradient of the area seen in figure 2. The discrete version of this equation is outlined in equation 9, where α and β are seen in figure 2.

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

The approximations in equation 7 and equation 9 can be improved by choosing the *voronoi* area of the 1-ring neighbourhood. The *voronoi* area is given by equation 10. Replacing A with the *voronoi* area A_v in equations 7 and 9 greatly improves the curvature results. The implementation can be found in the method VertexCurvature.

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

2.8 Classifying the genus of a mesh

The genus of a mesh is loosely defined as the number of holes a mesh has. A solid cube would for example have a genus of 0, while a torus would have a genus of 1 and so on. To calculate the genus of a mesh we use the *Euler-Poincaré* formula described in equation 11. V, E and F represent the number of vertices, whole (non-half) edges and faces while L, S and G represent the number of loops, shells and the genus of the mesh.

$$V - E + F - (L - F) - 2(S - G) = 0 \tag{11}$$

Realizing that we are only dealing with triangle meshes in this lab, the number of loops will be the same as the number of faces. Setting $L = F$ and rearranging equation 11 gives us

$$G = \frac{-V + E - F + 2S}{2} \tag{12}$$

which will result in the correct genus classification. The implementation was done in the method Genus.

2.9 Computing the number of shells

In equation 11 used in the section above, S is defaulted a value of 1. Shells can be described as parts of a mesh with no topological connections to each other.

We compute the number of shells by using a flood-fill algorithm. Starting at any given vertex we traverse the vertices of the mesh using the half-edge structure, ”tagging” vertices as visited as we go. For each visited vertex, we collect the 1-ring neighbourhood of that vertex. If any 1-ring neighbour vertex has not been ”tagged” it is put in a queue to be visited. When we have visited every vertex possible by half-edge traversal, e.g. the queue is empty, we compare our list of ”tagged” vertices to the mesh vertex list. If they are not of equal size it means the mesh consist of two or more shells. In that case we pick an arbitrary ”untagged” vertex from the vertex list and repeat the process until the size of the ”tagged” list equals the number of total vertices for the mesh.

Algorithm 4 Compute number of shells by flood fill algorithm

```
1: shells ← 0
2: currentVert ← 0
3: vertQueue.push(currentVertex)
4: while tagged.size < numberOfVerts do
5:    while vertQueue ≠ empty do
6:       currentVert ← vertQueue.pop
7:       if currentVert not in tagged then
8:          tagged.add(currentVert)
9:          for all v_i in 1-ring of currentVert do
10:            if v_i not in tagged then
11:               vertQueue.push(v_i)
12:            end if
13:         end for
14:      end if
15:   end while
16:   if tagged.size < numberOfVerts then
17:      currentVert ← unvisited vertex
18:   end if
19:   shells += 1
20: end while
```

3 Results

3.1 Vertex normals

In the program given in the lab, vertex normals are visualized as green lines and face normals as red lines as seen in figure 3.

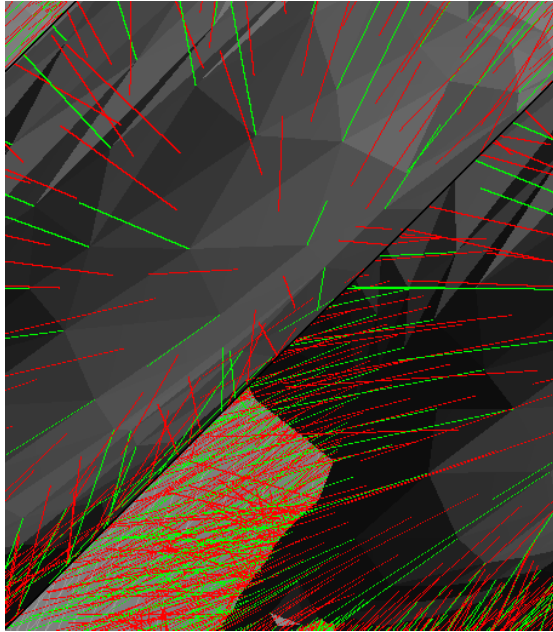

Figure 3: Face normals in red and vertex normals in green.

3.2 Surface area of mesh

By looking at spheres with radii of *0.1*, *0.5* and *1* we can calculate the analytically correct areas of these spheres and compare it to the results from our mesh representations of the same spheres. See table 1 for result comparisons.

Sphere radius	True area	Calculated area
0.1	0.12566	0.12511
0.5	3.14159	3.12775
1.0	12.5664	12.5110

Table 1: Comparison between true and calculated area values for spheres of different radii.

3.3 Volume of mesh

We again use the spheres as an example for mesh volume calculation. See table 2 for results.

Sphere radius	True volume	Calculated volume
0.1	0.0041888	0.0041519
0.5	0.523360	0.518988
1.0	4.18879	4.15190

Table 2: Comparison between true and calculated volume values for spheres of different radii.

3.4 Curvature calculations

As discussed in [1] the curvature of a sphere is inversely proportional to the radius of that sphere, $\frac{1}{R}$ where R is the radius of the sphere. This implies that a sphere with a radius of *1* will have a curvature of *1*. Using the *voronoi* area approximation discussed in section 2.7 results in principle curvature values in the interval $[0.997816, 1.00227]$ for $[\kappa_1, \kappa_2]$.

3.5 Genus classification

Using *cube.obj* yields a result of $genus = 0$ while using *genuscube.obj* yields a result of $genus = 5$.

3.6 Number of shells

Using *cube.obj* returns $shells = 1$. Using *genusTest.obj* returns $shells = 4$ and $genus = 3$.

4 Conclusion

As discussed in [1] a simple mesh structure with only a vertex list and triangle list has a memory usage of $18F$ bytes for a mesh with F faces. [1] goes on and states that a mesh with a half-edge structure has a memory usage of $72F$ bytes for a mesh with F number of faces. This translates to an increase in memory storage by a factor of 4, which can be considered pretty steep. In return we get a mesh structure with fast random access to vertex neighbourhoods which can be beneficial for some applications. The simple mesh is very fast for linear traversal of triangles e.g. for rendering applications, but performs rather poorly when access to neighbourhood information is needed. As stated in section 1.1, accessing neighbourhood information is an operation of quadric complexity for a simple mesh structure. The complexity for the algorithm used in the half-edge structure to access neighbourhood information, since we assume closed manifold triangle meshes, becomes almost trivial in comparison.

When the *voronoi* area approximation was implemented for the curvature approximations in sections 2.6 and 2.7 the results were greatly improved. It might therefore be worth the extra effort it takes to implement.

The reason for the volume values in section 2.5 being slightly smaller that the real volume is because the meshes we use are only approximations with a finite number of vertices. Each vertex may lie on the surface of the real sphere, but when these vertices are connected to form triangles these triangle faces does not lie on the surface of the sphere. If we let the number of vertices increase the volume integral approximation by the Riemann sum would become more accurate and thus provide a more accurate result.

In conclusion; even though the half-edge mesh structure is a little trickier to implement it is very well suited for modeling applications. Considering memory is comparatively cheap it could be worth the effort to implement and use this structure for high performance modeling applications.

5 Lab partner and grade

The lab was completed in collaboration with Hans-Christian Helltegen, hanhe945. Since I have completed all assignments with (*) and (**) as well as one assignment with (***) I should qualify for grade 5.

References

[1] Gunnar Läthén, Ola Nilsson, Andreas Söderström, Stefan Lindholm *Mesh Data Structures*. TNM079 Modeling and Animation, Linköping University, 2013.