THE HALF EDGE MESH DATA STRUCTURE

MODELING AND ANIMATION

Albert Cervin
albce943@student.liu.se

Wednesday 13th April, 2011

Abstract

In this paper I will present theory and results when implementing mesh data structures, in particular the half edge mesh data structure. This is the first lab in a lab series consisting of six labs in the course TNM079 - Modeling and Animation at Linköping University. Topics that will be covered include area and volume calculations, calculation of genus and number of shells and also different methods for curvature estimations. Methods for calculating vertex normals will also be discussed. The results will then be presented showing that the half edge mesh data structure is very efficient when analyzing meshes in different ways. It uses more memory however, than the more common polygonal mesh representation. The conclusion is that if the extra memory cost is affordable, the half edge mesh data structure is worth considering.

1 Introduction

The course TNM079 - Modeling and Animation consists of six lab sessions to be performed. Three of these sessions are to be presented in a report presenting what has been done and how it has been done. This report describes my work in the first lab sessions covering mesh data structures which is the purpose of this lab. The main data structure presented in this lab is the half edge mesh data structure which is important to know since it is very efficient in terms of topology.

1.1 The polygon mesh data structure

To describe the geometry and topology of an object in computer graphics some kind of data structure is needed. A mesh data structure is one (and the most common) way of describing objects.

One of the most common mesh data structures is the polygon mesh data structure. It is built by vertices that are connected into faces, which form the surface of the object. The popularity of this method can be contributed to the simplicity of it. It is also fast to render and memory efficient since there is no redundancy in vertices. In [1] it can be seen that, with V denoting the number of vertices in the mesh and F the number of faces, the data structure uses $V * \texttt{sizeof(Vertex)} = 12V$ bytes for geometry and $3 * F * \texttt{sizeof(Vertex*)} = 12F$ for the topology and it can be shown that the relationship between number of faces and vertices is $F \approx 2V$. This means that the memory usage for the polygon mesh data structure is $18F$ bytes per mesh, which can be considered a lower bound for mesh data structures with random vertex access.

The problem with this data structure appears when facing the problem of finding adjacent vertices and faces of a specific vertex. The whole data structure has then looped over, which gives a linear time complexity. The real problems occur when it is needed to find neighbors for each vertex in the mesh, then the time complexity is $O(n^2)$. This fact leads us to the next question: Is there other data structures that solves this more efficiently?

1.2 The half edge mesh data structure

The answer is yes. The half edge mesh data structure address the above problem. Besides storing vertices and faces, it also stores information about edges in the mesh. The edges are defined as a vertex, a face and three connected edges as shown in figure 1.

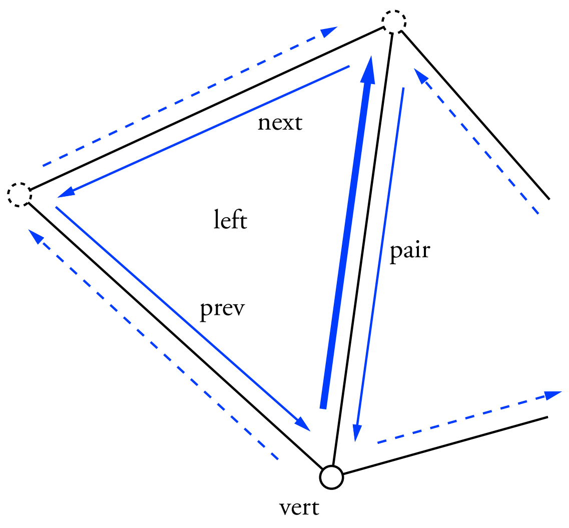

Figure 1: The half edge mesh data structure. (Image courtesy of [1])

This gives a much more efficient way of finding neighboring faces and vertices of a given vertex, by traversing the edges. Since this data structure stores more information it will consume more memory than the polygon mesh data structure. It can be shown that the size of the half edge mesh in terms of memory usage is $72F$ `bytes` where F is the number of faces in the mesh.

2 Method

2.1 Implement the half edge mesh

To implement the half edge mesh, several connections have to be done correctly and this can be a bit complicated. Some of the code was already implemented in the code that was given for this lab. To finish the implementation we followed the steps outlined below, adding code in the function `AddFace` located in `HalfEdgeMesh.cpp`.

1. Start by adding the three vertices sent to the function using the `AddVertex` function and store the three indices returned.
2. For each pair of indices returned in step 1, add a half edge pair with the function `AddHalfEdgePair`. The two indices returned from this function is saved.
3. Use the indices returned in step 2 to connect the edges by setting the indices `next` and `prev` to the correct values.
4. Create a new `Face` and insert it in the `mFaces` vector. Connect the face to the edge constructed vith the first vertex.
5. Calculate the normal for the new face and store it in the struct.
6. Set the `face` field of each of the cunstructed half edges, to the newly created face.

2.2 Implement neighbor access

Implementing neighbor access is a matter of finding adjacent faces or vertices of a given vertex and store that in a list of neighbors. The introduction of half edge meshes greatly simplifies this operation and pseudo-code for the algorithm is given below.

```
edge = current_vertex.edge.next
end = edge

do
{
  one_ring.add(edge.vertex)
  edge = edge.next
  edge = edge.pair
  edge = edge.next
}while( end != edge )
```

When `end == edge` the ring is closed and we can stop searching.

Neighboring faces are found in a similar way:

```
edge = current_vertex.edge
end = edge

do
{
  one_ring.add(edge.face)
  edge = edge.prev
  edge = edge.pair
}while ( end != edge )
```

These two functions were implemented in `FindNeighborVertices` and `FindNeighborFaces` respectively, in the file `HalfEdgeMesh.cpp`.

2.3 Calculate vertex normals

In the code, face normals is already iplemented as above and with face normals, only flat shading is possible. To produce smoothed shading (Gourad or Phong etc.) vertex normals are needed. Vertex normals are calculated using a method called mean weighted equality which is the normalized sum of adjacent face normals [1].

$$n_{v_i} = \widehat{\sum_{j \in N_1(i)}^{n}} n_{f_i} \tag{1}$$

In equation 1 $N_i(i)$ is the adjacent faces of a given vertex v_i. This equation was realized in the function `VertexNormal` and adjacent faces were found using the function `FindNeighborFaces`.

2.4 Calculate surface area of a mesh

Since an integral can be approximated by a Riemann sum, the surface area of a mesh can be calculated as a sum of the areas of each of the faces in the mesh.

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

Equation 2 describes the calculation of the area where $A(f_i)$ is the area of face i. $A(f_i)$ is calculated as half the magnitude of the cross product of any two edges of the face.

$$A(f_i) = \frac{1}{2} |(v_2 - v_1) \times (v_3 - v_1)| \tag{3}$$

.

2.5 Calculate volume of mesh

To calculate the enclosed volume of a mesh, the *divergence theorem* is used.

$$\int_s \mathbf{F} \cdot \mathbf{n} \quad dA = \int_V \nabla \cdot \mathbf{F} \quad d\tau \tag{4}$$

Equation 4 relates area and volume saying that the surface integral of a vector field times the unit normal equals the volume integral of the divergence of the same vector field [1]. Here, F is assumed to have constant divergence $\nabla \cdot \mathbf{F} = c$. This gives the volume integral in equation 5

$$\int_v \nabla \cdot \bar{\mathbf{F}} d\tau = 3V = \int_S \bar{\mathbf{F}} \cdot n dA \tag{5}$$

Equation 5 is then discretized by a Riemann approximation. The vector field is evaluated at the centroid of the face giving equation 6 describing the enclosed volume of a mesh.

$$3V = \sum_{i \in S} \frac{(v_1 + v_2 + v_3)_{fi}}{3} \cdot \mathbf{n}(f_i) A(f_i) \tag{6}$$

The volume calculations were implemented in the `Volume` method in `HalfEdgeMesh.cpp`.

2.6 Implement and visualize curvature

The curvature estimation was already implemented in the source code of the lab, in form of a Gauss estimation.

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

Equation 7 shows the formula for Gauss estimation of the curvature where A is the area of the one-ring neighborhood and θ is the angle between the current and the next vertex in the neighborhood ring. This estimation is however inaccurate and to improve the estimate, Voronoi area can be used as stated in[1].

2.7 Improve the curavture estimate

The accuracy of equation 7 can be improved further by using the Voronoi area of the neighborhood.

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

A in equation 7 is then replaced by A_v in equation 8. To use this it is necessary to have the previos, next and current vertex current when looping through the neighborhood ring. The current vertex is v_j, the previous is α_j and the next is β_j as seen in figure 2.

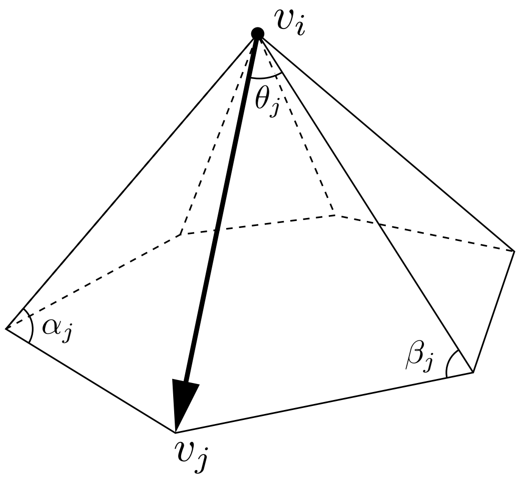

Figure 2: One-ring of neighboring vertices.

This curvature estimate was implemented in the function `VertexCurvature` in `HalfEdgeMesh.cpp`.

2.8 Classify the genus of a mesh

The genus of a mesh tells how much holes there are in the mesh. A mesh with genus $G = 1$ means that there is one hole in the mesh. Genus for a mesh can be calculated with the *Euler-Poincaré* formula (eq 9).

$$V - E + F - (L - F) - 2(S - G) = 0 \tag{9}$$

In equation 9 V is the number of vertices, E is the number of *unique* edges, F is the number of faces, L number of loops, S is the number of shells and G is the genus for the shell. A loop in this formula is a unique ring along edges in the mesh. For triangle meshes there are always one loop per face giving $L = F$. If we furthermore only consider one shell at a time ($S = 1$) the calculation of genus can be simplified and rewritten as

$$G = 1 - \frac{(V - E + F)}{2}. \tag{10}$$

Equation 10 was implemented in the function `Genus` in `HalfEdgeMesh.cpp`.

2.9 Compute the number of shells

Computing the number of shells (and thus *not* assuming $S = 1$) is implemented by a flood fill algorithm for each shell. We use a `map` for tagging each shell and then inserting a set of the vertices of that shell. We also hold an array where we flag visited vertices and when we have flood filled one shell, we advance to the next by finding the next unvisited vertex in the vertex list.

The constant 1 in equation 10 is then replaced by the result of the shell calculation. The algorithm was implemented in the function `Shells` in `HalfEdgeMesh.cpp`. Pseudo-code for the algorithm is as follows.

```
initialize visited array

next_object = 0
num_shells = 0

while (next_object < num_verts)
{
    push next_object -> vertex_queue_set

    while (!vertex_queue_set.empty())
    {
        v <- vertex_queue_set.pop()
        push v -> vertex_tagged_set
        mark v as visited

        for (all v_i in find_neighbor_vertices(v))
        {
            if v_i is not in vertex_tagged_set
                push v_i -> vertex_queue_set

        }

    }

    next_object = next unvisited vertex
    num_shells++
}
```

3 Results

3.1 Loading of meshes

As said before, analyzing meshes becomes much faster with the half edge mesh data structure and table 1 shows the increase in speed when analyzing a simple polygonal mesh versus using the half edge mesh data structure. The times are calculated using `ctime` and measures the time spent in calculating normals and estimating curvature for the mesh (vertex and face both for normals and curvature).

Object	Number of vertices	Simple mesh	Half edge mesh
`sphere1.0.obj`	994	0.015s	0.01s
`bunnySmall.obj`	17518	4.45s	0.171s
`bunnyLarge.obj`	140130	473.757s	1.17s

Table 1: Time spent analyzing for different objects.

3.2 Calculation of vertex normals

The vertex normals are shown in green and the face normals are shown in red in figure 3.

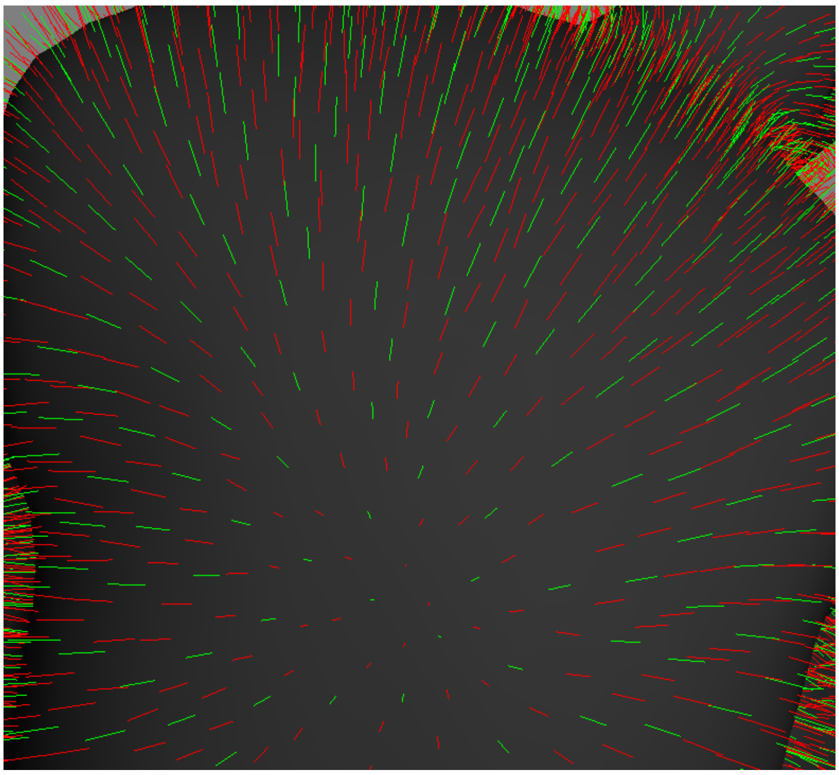

Figure 3: Vertex normals in green and face normals in red.

3.3 Volume and area calculations

Results of the area and volume calculations are shown in table 2 and 3 for the three spheres of different radii. The reason for choosing the spheres is that it is very hard to evaluate the approximations for the other meshes. The correct values for area and volume are also provided for comparison.

Object	Calculated area	True area
`sphere0.1.obj`	0.12511	0.12566
`sphere0.5.obj`	3.12775	3.14159
`sphere1.0.obj`	12.511	12.5664

Table 2: Area calculations for spheres of different radii.

Object	Calculated volume	True volume
`sphere0.1.obj`	0.0041519	0.004189
`sphere0.5.obj`	0.51899	0.52360
`sphere1.0.obj`	4.15192	4.18879

Table 3: Volume calculations for spheres of different radii.

3.4 Curvature estimate

The (Gaussian) curvature for a sphere is $1/R$ where R is the sphere radius. Furthermore, a sphere with radius $R = 1$ has a principal curvature of 1. This can be compared with the actual results for the `sphere1.0.obj`: [0.99839, 1.00609]. Visualization of the curvature can be seen in figure 4.

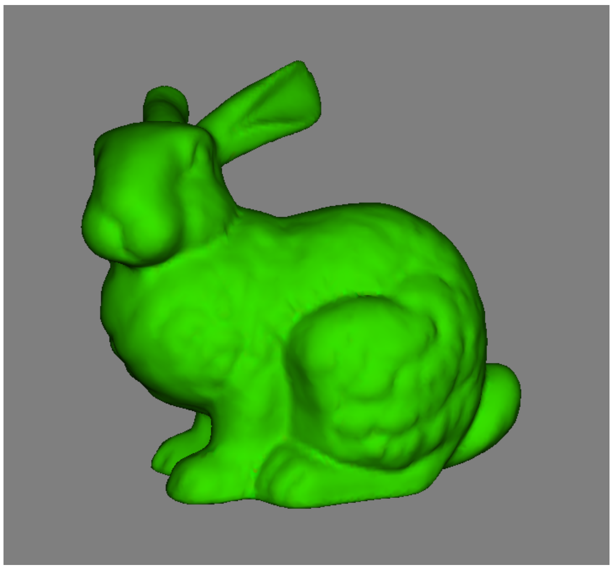

Figure 4: Vertex curvature visualized on the model `bunnySmall.obj`.

3.5 Calculation of genus

For the sphere models: $G = 0$.
For `genuscube.obj`: $G = 5$

3.6 Calculation of number of shells

For the model `genusTest.obj`: $S = 4$ and $G = 3$. The model can be seen in figure 5.

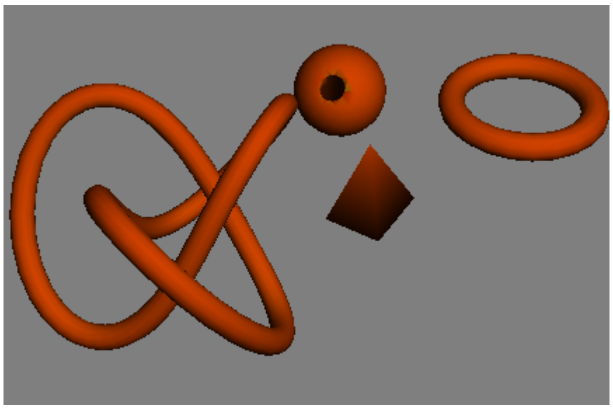

Figure 5: The model `genusTest.obj`.

4 Conclusion

From the results in section 3 it can be seen that the half edge mesh data structure is very effective in analyzing the mesh. The conclusion is that the time complexity for analyzing the simple mesh is of considerably higher order than the complexity for the half edge mesh. However, the complexity of the implementation is higher for the half edge mesh and it also uses more memory than a simple polygon mesh.

In the calculation of area and volume for the three spheres (table 2 and 3) it can be seen that there is small numeric errors. This is since the mesh is only an approximation of a real sphere and the Riemann approximation becomes better the more triangles there are in the mesh. When the number of triangles go towards infinity, the area calculation will converge to an accurate result.

The curvature estimation increases significantly in quality when using Voronoi regions, which leads to the conclusion that it is worth the minor extra work to implment it.

The calculation of shells work well and is pretty fast. The only disadvantage it has is that it is not that memory efficient since it allocates a boolean array with the same size as the number of vertices which can be a lot in complicated meshes. If the mesh then furthermore only consists of one shell, the computational waste is very big.

The overall conclusion of this lab session is that if the extra memory cost is affordable and there are much analyzing of the mesh to be done, the half edge mesh is the way to go.

Lab partner and grade

I did this lab together with Nathalie Ek (natek725). All tasks marked with * was completed, as well as the tasks marked with ** and also one task marked with ***, which should qualify me for grade 5.

References

[1] Gunnar Läthén, Ola Nilsson, and Andreas Söderström. Mesh data structures. Technical report, ITN, 2011.