5-2013

Modified half-edge data structure and its applications to 3D mesh generation for complex tube networks.

Richard Paris
University of Louisville

Modified Half-Edge Data Structure and Its Applications to 3D Mesh Generation for Complex Tube Networks

By

Richard Paris
B.S., University of Louisville, 2012

A Thesis
Submitted to the Faculty of the
University of Louisville
J. B. Speed School of Engineering
as Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Computer Engineering

May 2013

Modified Half-Edge Data Structure and Its Applications to 3D Mesh Generation for Complex Tube Networks

Submitted by:____

Richard Paris

A Thesis Approved On

____

(Date)

by the Following Reading and Examination Committee:

____

Dr. Dar-jen Chang, Thesis Director

____

Dr. Ming Ouyang

____

Dr. John Pani

I. ABSTRACT

Modified Half-Edge Data Structure and Its Applications to 3D Mesh Generation for Complex Tube Networks

Richard Paris

May 2013

In computer graphics 3D mesh generation is an important topic, it is required for a vast number of applications. While there are currently solutions available for the generation of meshes, there is not one that suits our application well that is written in C#, for this reason a C# implementation of the half-edge data structure as well as a C# implementation of the mesh generation algorithms is needed. This document will discuss in detail the method by which the algorithms are implemented, the improvements that are made on the half-edge data structure, and the new features that have been added to the new application. Further this document will evaluate the performance improvement made by the improvements mentioned.

II. TABLE OF CONTENTS

III. LIST OF FIGURES

IV. LIST OF TABLES

V. LIST OF ALGORITHMS

I. INTRODUCTION

Moore’s law states that computational power will move forward at an exponential rate, the problem is that currently it is moving at a linear rate; because of this limitation there is a need to implement efficient data structures in the effort of continuing the pace of moving computing ability forward at that exponential rate. One such area that continues to grow is 3D computer graphics, particularly the representation of 3D models in a virtual environment. Furthermore, the complexity of accurate 3D models results in a tremendous need of storage, some accurate models result in up to 1 billion polygons (Levoy, 2011).

Currently the predominant data structure being used to represent these 3D models is the half-edge data structure. While there are others (such as the quad-edge and doubly linked face list data structures)(Kettner, 2012), many of the commercial and open source geometry libraries use the half-edge data structure as the primary method of storage and analysis (Leadwerks, 2006). The half-edge data structure is an “edge-centered” structure; it is primarily concerned with storing links between half-edges as the main method of traversing the mesh. Each half-edge must at least contain a pointer to its opposite half-edge as well as the next half-edge in the contained face.

The major goal of this paper is to improve the current application, through various means, that is currently being used to visualize a 3D tube structure. Currently the main use of the application is that of medical splines created from medical imaging, but can be expanded to many fields such as game development, physics simulations of structures, etc. The application being replaced is developed using CGAL, a computational geometry library with a built in half-edge data structure, and open-inventor, a 3D graphics API for visualization.

This paper will address three goals set forth for the modification of the half-edge data structure in general and the existing application that is in use today. First and foremost, the data structure being used is an implementation that was developed in C++. This implementation makes use of template classes as well as many other programming concepts that are not easily read and usable. Because of this there is a push to migrate the application to C#, which allows for a more fluid implementation. A C# implementation also allows for the use of free memory management, enumerators, and other techniques that will be discussed in chapter V. The second area of improvement that will be undertaken is to implement support for non-orientable surfaces; the current data structure being used does not support this, and while not necessary for many applications, can be useful in certain instances. Finally the most important aspect of improvement of the data structure is the inclusion of a hash table of unlinked half-edges. The key of this hash table is the connected vertices, allowing for O(1) time access to a half edge provided it currently has no opposite. This will allow for reduced computational time for many of the algorithms necessary for completion of the process.

These three aspects are the driving force of this paper, which proposes changes to be made to the current implementation and the current data structure that will address those areas, as well as improve upon other minor issues. As stated computational power is a limitation on this system that needs to be addressed, especially computational time as it relates to this application. One point of emphasis was that the new implementation needed to complete the process more quickly and efficiently without sacrificing flexibility or usability. The proposed changes to the current method should improve flexibility of the application, reduce the time that is required for the current process and finally to allow for an application that is modifiable to the needs of the user.

This paper will first address the motivation of mesh generation and its applications, focusing on the impact that can be had from 3D tube mesh generation. These motivations include medical imaging and model reconstruction, virtual environments and procedural level generation, and finite element analysis as it pertains to mesh simulation. Following will be a literature review and overview of current triangular meshes, particularly the data structures being used, and the methods by which one can evaluate the efficacy of an implementation of the half-edge data structure.

Chapter IV will contain a discussion of the various improvements and modifications that will be made to the current data structure, as well as make note of why these improvements are valuable. This chapter will discuss the implementation of the hash table mentioned previously, the method by which non-orientable surfaces are supported, and some of the minor improvements made to usability and flexibility of the data structure implementation. Chapter V will discuss the algorithm that is being used in the application giving more detail into how the mesh is generated and the current techniques being employed to improve upon the quality of the meshes. The generation of the tube mesh has two major parts that will be discussed, that is the creation of a branch and of a node.

In the next chapter, results of the improvements will be discussed; in this section certain time and complexity improvements will be introduced, and the data structure will be evaluated using standard criteria. Finally the paper will go into recommendations for future work of this project, including continued look into special cases for the application, and texture assignment for improved visualization.

II. 3D Modeling Applications and Representations

3D modeling is the field of representing 3D solids using mathematical models. Specifically it is a collection of 2D or 3D objects in a 3D space that are connected using various data structures. Typically the models represent a real-world object, but this does not have to be the case. Representing real-world objects allows for one to perform analysis, simulations, and many other applications on the object. Using objects that are not from the real-world, allows for one tor represent virtual and constructed environments that can be used in games, movies, television, and many other areas of entertainment. 3D modeling has a number of specific applications that will be discussed. Current 3D modeling techniques are being used in the medical field to represent any number of biological parts. One can also develop levels procedurally using computational 3D modeling. Finally 3D modeling allows for one to perform physics simulations on the 3D models.

Currently there are two main techniques to represent a 3D solid; these are boundary representation and constructive solid geometry (CSG). CSG is a referred to as “intelligent geometry” (Leadwerks, 2006), it uses a number of simple solids in conjunction with Boolean operations to form a complex solid. These simple solids are convex objects, meaning that there are created using only intersecting coplanar faces. There are three Boolean operations that are used to allow for this technique to have full representation, they are the intersection, union, and set difference. These two aspects allow for complex solids to be created. The other technique is that of boundary representation; boundary representation is “more explicit” (Marshall, 1997) than CSG and stores information about the solids faces, edges, and vertices in order to completely represent the solid. The benefits of this method are that surface information is more readily available; it allows for much simpler representation, it is effectively a combination of faces rather than a combination of solids. Additionally the information of neighboring vertices, faces, or edges are readily available. Finally boundary representation is useful for determining local normal at each vertex, and quickly transforming into a format usable by current graphics cards. One method of boundary representation is the half-edge data structure; the half-edge data structure stores information mainly about the faces, vertices, and edges of a solid; however it mainly uses the half-edge information for traversal of the solid. Adjoining faces are shown as adjoining using a combination of linked half-edges, for each pair of half-edges the faces that they belong to are also now adjoining. This paper and application will be focused on boundary edge representation, specifically the half-edge data structure.

A. <u>Medical Tube Structures</u>

One primary application of 3D modeling is the representation of important medical images. Currently medical imaging consists of taking a series of 2D images at various depths so that a medical professional can examine them for research, diagnosis, or other clinical reasons. These images can be used to recreate a 3D representation of the volume being imaged. The purpose of this procedure is to allow the examiner to have a better model to study, one that can be examined as a 3D model.

Various biological objects can be represented a series of interconnected tubes, the nervous system, the cardiovascular system, even the skeletal system is a series of bones each of which is effectively a simple tube connected by tendons. It would be of value to have a way to analyze these tube systems. The proposed application will allow for that. Given a directed graph that can be taken from the medical images, this application can create a triangular model that approximates the biological object. Once this model is generated, one can perform simulations, programmatically classify the model based on previous data, or compare the model to a previous model to detect changes and anomalies.

Another problem with medical data is that it is so large; one medical doctor estimated the amount of data to be as high as 150 exabytes (Hughes, 2011), this is because each medical 3D model is hundreds of images stored together. This makes storage costly and causes medical data transfer to be a big endeavor. Storing data as a directed graph would reduce the size of a file by as many as four orders of magnitude. Further, a program that could quickly represent that directed graph as tube structure would allow for medical data to be transferred along with the program significantly reducing the amount of data to be transferred.

B. Procedurally Generated Levels

Another application of 3D modeling is the ability to create and generate levels procedurally using either predefined data along with an algorithm, or allowing for random variation to create unique levels for each user and upon each use. The current predominant method of creating a level in a video game is to have a team create every aspect of a level. One person may create a skeleton of the level, meaning they develop the basic structure of the level, and then a second member will be responsible for texturing and adding detail to the level. This means that every level takes a significant amount of time to create, as well as a significant investment. The second part of this method could be replaced by an intelligent algorithm capable of creating realistic levels from a skeleton; this is one of the possible applications of 3D modeling.

The proposed application will be able to take a skeleton, in this case a directed 3D graph, and procedurally create any tube-like structure. For example a complex cave system could be created by setting up the skeleton of a tunnel system and applying the algorithm presented in chapter V to create the next step in development of the mesh, the high-level detail. The benefit of this algorithm is that it attempts to mimic how the tube system would come together in nature. The benefits of creating the geometry procedurally are more than just the ability to rapidly develop these structures.

The other benefits arise from being able to store these structures in a much compressed form, i.e. the original directed graph.

After applying the initial tube creation algorithm, the structure must be subdivided a number of times in order to give it the natural appearance, this subdivision process is computationally intensive, this means that the data structure must be efficient so that the subdivision algorithm can run in an appropriate timeframe. This is one of the goals of the paper, to implement this efficient data structure that will allow for the subdivision to be efficient.

There are two other results that arise from the application of subdivision: first the intermediate steps can act as a simplified mesh for collision detection. Mesh collision detection works by the following algorithm (algorithm “ALGO 1”).

```
foreach (Polygon p1 in mesh.polygons)
{
        foreach (Polygon p2 in mesh.polygons)
        {
                if (p1 == p2) continue;
                if (p1.intersects(p2))
                {
                        //collision detected
                }
        }
}
```

Algorithm 1 Collision Detection

This shows that collision detection is on the order of n-squared, thus very inefficient, a simplified mesh allows for collision detection to be done on a mesh with very few polygons to same time without being noticeable to the user. The second result of the subdivision is that it adds not just more detail, but the ability to randomly generate detail. Some structures like blood vessels are typically smooth, but there are tube structures that have detail in them, this detail can be modeled as noise in the mesh, resulting in detail that does not need to be introduced by the designer.

C. Physics Simulations

A great deal of research is done through the use of simulation of dynamical systems, this is because it is much too difficult to recreate many systems and conditions or it is done to forecast potential outcomes based on future occurrences. Forecasting requires extremely accurate models and data which can be provided for through the use of 3D mesh generation. Currently physics simulations on physical objects are done using either rigid body or soft body dynamics, and the interactions are performed on primitives such as triangles, cubes, spheres, etc.

These simulations still can take a great deal of time and computing power which is why it becomes important to have efficient mesh data structures. The process of simulating interactions on a group of discrete primitives is called finite element analysis (FEA). FEA involves applying forces, torques, or heat on a system of discrete objects. Then each of those objects acts upon its local neighbors to determine what was the effect on itself (this could be change in position, velocity, acceleration, temperature, rotation, or any physical property). This is done iteratively to each element in the mesh, calculating the stimuli on itself at that time increment and imparting stimuli on its neighbors for the next time increment. Typically FEA then warrants some analysis, for example an FEA on an engine will result in many stresses, strains, and temperatures, at each stage an analysis should be performed to determine if the part will fail at that point.

FEA is used in nearly every high order simulation, but requires a detailed mesh for the analysis. Currently the mesh can be developed using an artist, a 3D scanner, or a software package such as AutoCAD to create the model from drawings. In the event of large structures such as caves, sewers, or transit tunnels this can be very difficult to achieve. The same is true for small systems that cannot be scanned such as the nervous or cardiovascular system.

III. Overview of Triangular meshes

A mesh is a set of polygons that are linked by common edges and vertices; which together form a 3D model specifically a polyhedral object. For the sake of simplicity each polygon is convex to simplify various operations. Meshes are used for representation to reduce processing time. Individual polygons require processing to be done on each vertex on each polygon, but linking these polygons together allows for processing to be done on many vertices that represent the same point at the same time.

There are many types of meshes that can be used in the representation of a 3D model; one can use a triangular only mesh, a mesh that uses quadrilaterals along with triangles, or any other combination of polygons. However, the triangular mesh is the most commonly used mesh. A triangular mesh is a polygonal mesh that uses only triangles to represent the surface.

The reasoning for using triangular meshes is that any model or object can be broken down into a set of triangles, but the same cannot be said for quadrilaterals and other higher degree polygons. The reason that every model can be decomposed into a set of triangles is the fact that every polygon can be broken down into triangles; however triangles cannot be broken down into anything but other triangles without adding additional vertices and increasing computational time. Further the triangle is always guaranteed to exist on a single plane allowing for all calculations for shading, texture mapping, etc. to be done using only linear interpolation between the three points on the triangle. One should note that while the data structure can force the higher degree polygons to be on a single plane, in the event of mesh modification resulting in vertex movement, this restriction may be violated unless checked after every operation; this can result in significantly more operations. Figure 1 (Kajak, 2011) shows a polygon that has been reduced to a set of triangles using the ear clipping method. Algorithm 2 (Kajak, 2011) shows the procedural method by which the polygon can be reduced into a set of triangles. The algorithm will only reduce the polygon into a list of triangles; a more advanced algorithm is used to maintain a mesh structure. This algorithm is given to explain a simple method of triangular reduction.

```
List<Triangle> ear_clipping(Polygon p)
{
      List<Triangle> T;
      while (P.vertices > 3)
      {
            foreach (Vertex v in P.vertices)
            {
            //test to see if the polygon excluding v contains v
                  if (!InNewPolygon(v, P))
                  {
                        T.add(new Triangle(v, v.next, v.prev));
                        link(v.next, v.prev); //remove vertex v
                                              //from polygon p
                        break;
                  }
            }
      }
      T.add(new triangle(P[0], P[0].Next, P[0].Prev);
      return T;
}
```

Algorithm 2 Ear Clipping

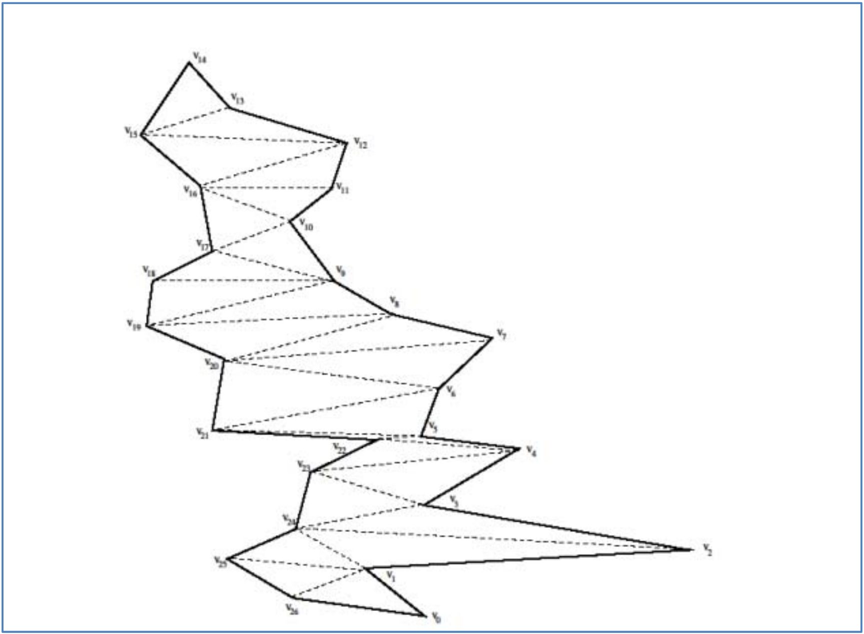

Figure 1 Ear Clipping

A. Triangular Mesh Data Structures

A triangular mesh data structure stores more information that just a list of triangles. The data structure also stores information about links among adjacent triangles and the method by which this is done classifies the type of data structure that is being used. This extra information must be stored in order to allow for various operations to be performed on the mesh, and so that information about the mesh can be quickly accessed in linear or constant time. There are many triangular mesh data structures that are currently being employed. The most commonly used data structures are the face-vertex, winged-edge, and half-edge.

The face-vertex data structure is the simplest of the data structures and is typically used by graphics processors because the processor does not need information about linked edges or linked faces. The face-vertex data structure is simply a list of triangles containing a pointer to the three vertices that make up the triangle. This data structure requires 3 pointers per face.

The winged-edge data structure is an edge based data structure meaning that the links between edges are explicitly defined and the links between vertices are implicitly defined. In the winged-edge data structure each edge points to its head and tail vertices, the two incident faces, and the four edges that are connected to each of its vertices. Figure 2 shows the representation of the winged-edge data structure. (Zorin, 2004) The winged-edge data structure requires eight pointers per edge and one pointer per face and vertex. Though more information can be stored to allow for faster access times, it is unnecessary because the other information is represented implicitly.

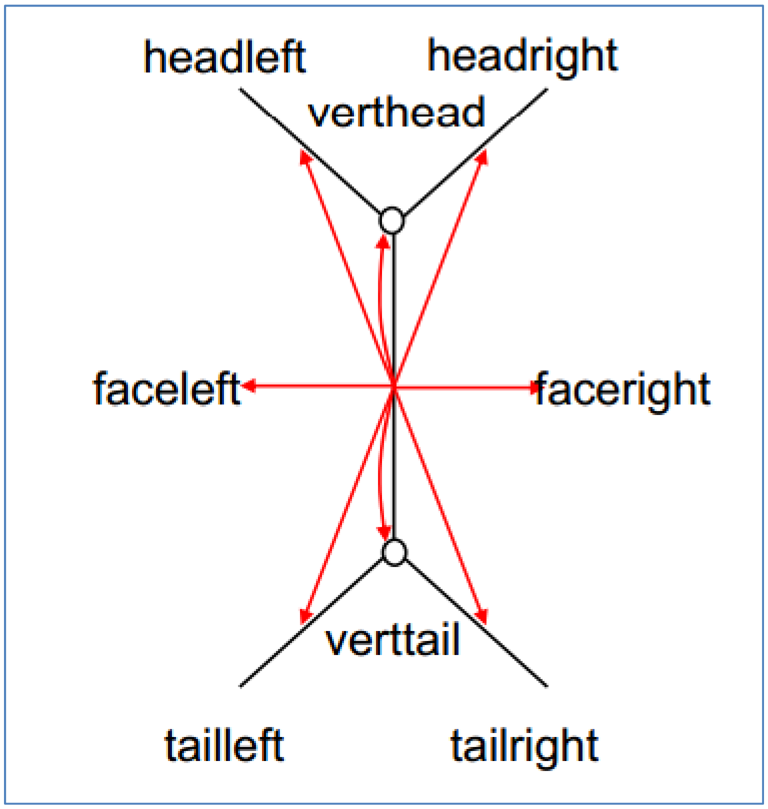

Figure 2 Winged-edge data structure

The half-edge data structure is one of the more powerful data structures and will be the one used for the purpose of this paper. It is a variant of the winged-edge data structure and it implemented by splitting up each edge into two and relying on half-edge traversal for many of the operations. The benefit of the half-edge data structure is that it allows for a consistent orientation among the triangles. The other benefit of the half-edge data structure is that there the traversals can be done without branching, which can lead to reduced traversal time. In the next chapter this paper will go into more detail about the half-edge data structure.

B. Half Edge Data Structure

The half-edge data structure is based on the winged-edge data structure, meaning that it is also an edge-based data structure. This also means that it is very similar to the wing-edged data structure in its representation; each edge stores a reference to its opposite (some data structures take advantage of spatial locality and assume that if ‘i’ is the index of one half-edge, its opposite is ‘i+i%2’), the two half-edges connected to itself, the two vertices that it touches, and the face that it helps make up. Each vertex stores a reference to a single half-edge that points to the vertex; for ease of circulation if there is an incident half-edge that is a boundary edge that half-edge is used. Finally each face stores any half-edge that borders it. Every other necessary access is done in constant time, and traversals done in linear time. Figure 3 (Zorin, 2004) shows the abstract representation of the half-edge data structure, including all the references each edge has. Not all of these references are necessary and can be represented implicitly; for example the tail vertex does not need to be explicitly represented. It can be accessed using ‘e.opposite.head’; one can make use of these implicit representations to for the half-edge data structure in the same amount of memory as the winged-edge data structure.

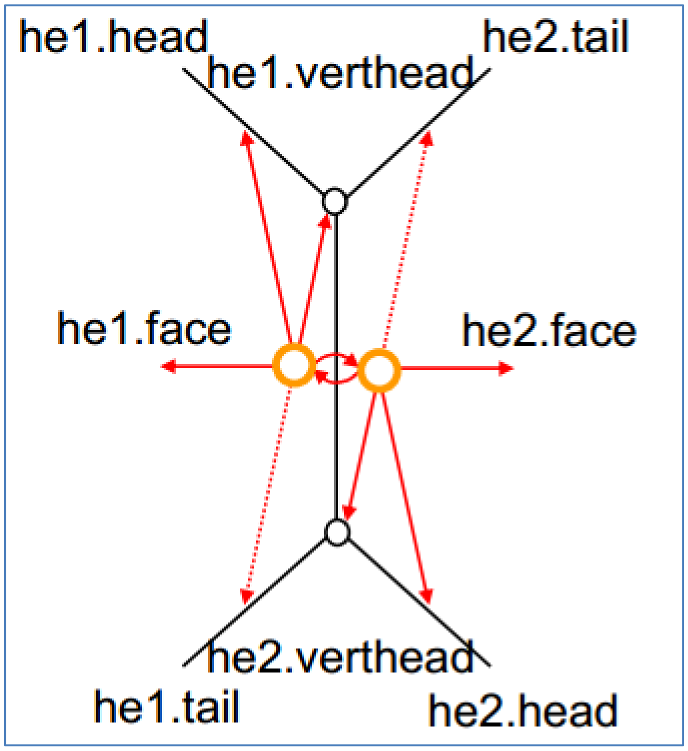

Figure 3 Half-Edge data structure

The half-edge data structure relies on the fact that each edge is bounded by exactly two faces and this allows the edge to be separated into two half-edges that are oriented in opposite direction. As stated this allows for a consistent orientation of the triangles, either clockwise or counter clockwise. The consistent orientation is beneficial in that for orientable surfaces it lets us know immediately which side is the visible side, and it causes the normal to always be oriented outward, given that the correct coordinate system is being used. The reason that the half-edge data structure has a consistent orientation is shown in figure 4 in which both triangles are oriented in a clockwise direction. As can be seen from the figure, when both triangles are oriented in the same direction, the adjacent half-edges are in opposite direction.

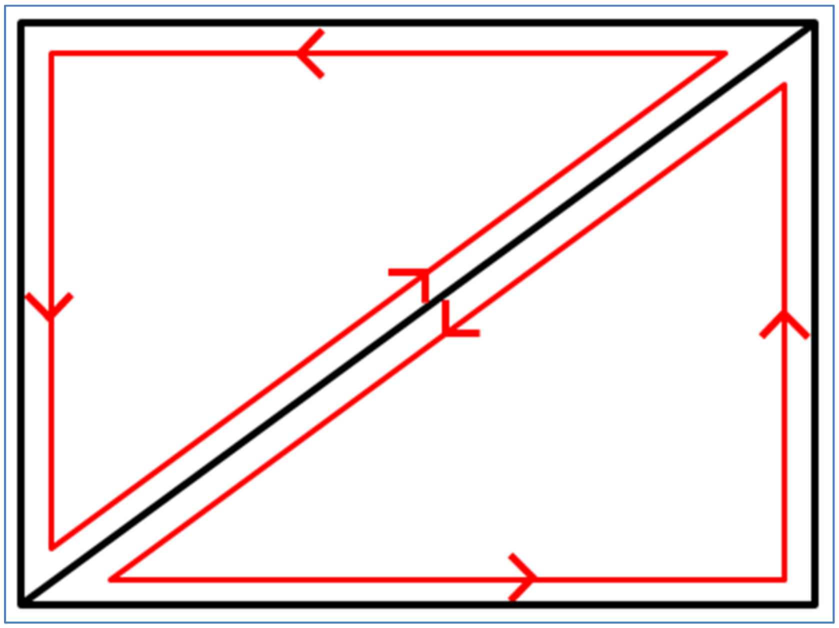
Figure 4 Triangular Orientation

It was also noted that the traversal algorithms no longer require a conditional to determine in which direction the circulator must go. The circulator always knows which way to go because each triangle has the same orientation as the last; so circulation can be performed according to algorithms 3 and 4.

```
List<Face> adjacent_faces(Vertex v)
{
        List<Face> F;
        HalfEdge e =  v.halfedge;
        //e must point to v
        do {
                F.add(e.face);
                e = e.next.opposite;
        } while (e != v.halfedge)
        return F;
}
```

Algorithm 3 Adjacent Face Circulator

```
List<Vertex> neighbor_vertices(Vertex v)
{
        List<Vertex> V;
        HalfEdge e =  v.halfedge;
        //e must point to v
        do {
                V.add(e.tail);
                e = e.next.opposite;
        } while (e != v.halfedge)
        return V;
}
```

Algorithm 4 Neighboring Vertex Circulator

The half-edge data structure contains a number of classes that are necessary to perform all necessary accesses, and calculations. The typical half-edge data structure has the following list of classes.

- Mesh: The main class that contains all other classes, has a list of half-edges, vertices, and if they contain pertinent data, a list of faces.
- Face: Has a reference to a general half-edge, and any pertinent data (e.g. color, normal, mass).

- Vertex: Has a reference to a half-edge that points to the vertex, and any pertinent data (e.g. color, location, normal).
- Half-Edge: Contains the references stated earlier in the paper. This is the most important class, and is used the most.

C. Metrics of Half Edge Data Structure

There are five metrics of a half-edge data structure that should be considered during implementation (Kobbelt). They are access, modification, operations, parameterization, and I/O. In this chapter each metric will be discussed and detailed and will make note of how the two major libraries currently perform in these metrics. Later in chapter VI this paper will discuss how well the proposed implementation performs per these metrics, and specifically what is being done to address each metric.

The first metric is access, meaning how quickly the program can access the vertices, edges, and faces. One should also measure how convenient it is to circulate through neighboring vertices, incident faces, and determine boundary edges of a face. Both libraries perform similarly in this respect; they each have iterators and handles that are used in for loops to access each element. This implementation provides for an easy to use interface for enumeration. They also provide a circulator that does traversal without the need for knowledge of the traversal algorithms. While the circulators and iterators can be easy to use, the move to C# and its enumerators allow for a more abstracted implementation.

The next metric is modification; ideally a mesh should be modifiable by the user without much trouble. One should be able to add or remove vertices and faces quickly without compromising the integrity of the structure. Any implementation must take this into account. Both libraries have methods to add and remove faces and vertices without ruining the mesh. These libraries are sufficient at this job, so there is no need to improve upon this metric; it just needs to be at least as efficient in this endeavor.

The half-edge data structure has certain operations that need to be performed for reasons such as analysis, simulation, or simplification. This means that the implementation should provide methods for completing these operations efficiently. Examples of these operations include ‘half-edge collapse’ for simplification, face/edge splits for subdivision, or specific to this application a method to create a convex hull around a set of points. These operations are provided for by the current libraries, but given the special cases of the application it may be possible to increase efficiency of the calculations.

An important consideration in our application is parameterization which allows for the user to add or remove information from the various objects in the data structure. Ideally faces and vertices should allow for data to be added and removed at runtime. This data should not be limited to a single type though; it should allow for a reference to any object to be placed as a property of the face or vertex. Both libraries have implementations that are general and efficient in their own right, but may be more robust than is necessary for the application.

Finally this structure should be easily converted to and from standard file formats. One file format that is used is the .off file format, which stores the data in a way similar to the face-vertex structure mentioned previously. It is a list of vertices containing their information, and a list of faces that contains its own information as well as indices corresponding to the vertices that make up that face. This is the file type that will be used for the purposes of this paper. The current libraries do again succeed in the metric, they both are able to export the data and import the data correctly. Thus the improvements will have to be due to an improvement in the time required for conversions.

IV. Half Edge Data Structure Improvements

The current implementation of the half-edge data structure has various areas that can be improved upon such as the method linking of the half-edges or the lack of support for non-orientable surfaces. There is also an absence of important methods for simplification, manipulation, and access. This chapter will discuss how each of the areas is addressed and improved upon.

A. Hash Table for Unlinked Half Edges

Currently when a face is added its opposite is found by iterating through every previous half-edge and checking for the correct opposite. As a result the time complexity of face insertion is linear. However this can be reduced to constant time by using a hash table to store each unlinked half-edge.

A hash table is an array of objects whose position is based on the hashing value of the key. This allows for constant time lookup of an object if the key is known. The key value of each object in this hash table is an ordered pair of integers which corresponds to the ids of the head and tail vertices of the half-edge respectively. It should be noted here that any half-edges opposite half-edge will have its head and tail vertex reversed; this means that to find the opposite half-edge of a given half-edge the key is simply the ordered pair of ids of the tail and head vertices respectively.

1. Subdivision

Subdivision is the recursive process of smoothing a given mesh. This is typically done by splitting a face into some number of newer triangles and adding some number of vertices. The location of the new vertices is determined using either an approximating or interpolating method. This method may also move the already existing vertices based on the locations of its neighbors. The current application uses loop subdivision which splits each face into four new faces, and each half-edge into two new half-edges.

This subdivision scheme results in each half-edge being unlinked during the process; the old method would ignore the half-edge opposite links until the end of the subdivision process then go through each half-edge and search for its opposite. This would cause the subdivision process to run in quadratic time complexity. The addition of the hash table reduces the time complexity to quadratic, but it introduces a new problem into the data structure. In some cases the hash table may grow to be very large, eliminating the constant time lookup of the hash table. This problem is mitigated by searching for half-edge links after each face is subdivided. An alternative method of linking is also available that does not use the hash table for finding opposite half-edges, it makes use of the fact that each edge is initially comprised of 2 half-edges and then is split into 4 half-edges allowing for linking to be done as soon as both adjacent faces are subdivided. Figure 5 below shows how this is done.

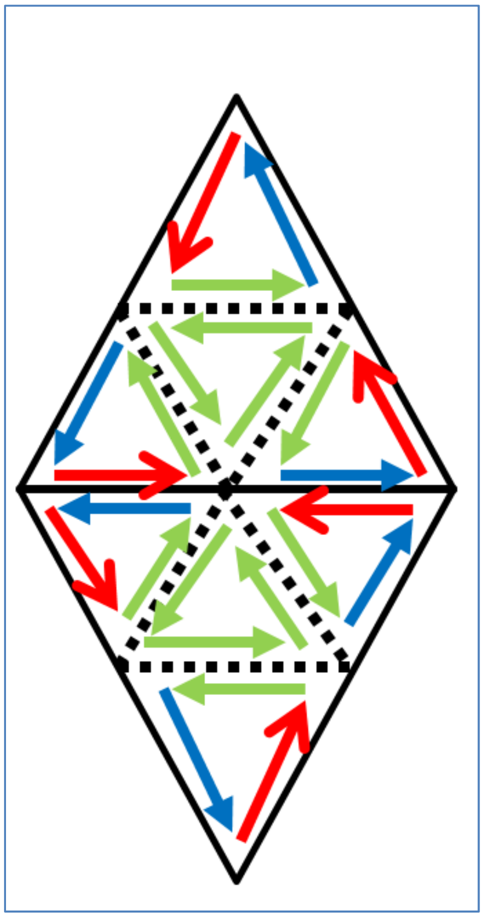

Figure 5 Loop Subdivision

In figure 5 the red arrows represent the already existing half-edges; the blue arrows represent the half-edges that are ‘children’ of existing half-edges, and the green arrow represent completely new half-edges. In each original triangle the ‘next’ and ‘previous’ half-edge reference as well as the ‘opposite’ half-edge reference for the internal half-edges (green) are set immediately following subdivision.

Assume that the top triangle is subdivided first then the bottom triangle, all of the ‘next’, ‘previous’, and internal ‘opposite’ half-edge references of the top triangle will be set, then the ‘next’, ‘previous’, and internal ‘opposite’ half-edge references of the bottom triangle will be set. At this point all original (red) and child (blue) half-edges do not have references to their opposites, instead each original (red) half-edge has a reference to their child’s (blue) opposite. From this we can check each of the three bounding faces to determine if they have already been subdivided, in this example only the two shown triangles have been subdivided so only one set of four half-edges are ready to be paired, Table 1 shows the how to access each of the four half-edges and its pair, assuming the original is called e.

Table 1 Post-Subdivision Half-Edge Links

Half-edge 1	Half-edge 2
e	e.opposite.child
e.child	e.opposite
e.opposite	e.child
e.opposite.child	e

2. Mesh Conversion

It is necessary to convert the half-edge data structure to and from the face-vertex structure for storage in an off file, or to transfer the mesh to the graphics processor. It is therefore an efficient method of conversion is required. The conversion consists of first placing the vertices into the mesh, then adding the faces one by one to the mesh. Using a linear time approach for adding faces to the mesh would result in a quadratic time complexity of the mesh conversion.

However, this hash table allows for face insertion to run in constant time which in turn allows for the mesh conversion to run in linear time, a significant improvement over the alternative. Algorithm 5 shows the pseudo-code for improved mesh conversion.

```
for each face f to be added
      create internal halfedges of f and link them
      for each internal halfedge e
            key := (e.tail.id, e.head.id)
            if the key is in the hash table
                  e.opposite := hash_table[key]
            else
                  key := (e.head.id, e.tail.id)
                  value:= e
                  add the key-value pair to the hash table
            end
      end
end
```

Algorithm 5 Improved Mesh Conversion

3. Boundary Tracing

The hash table at any time keeps track of all unlinked half-edges in the mesh and each of these half-edges is a boundary edge of the surface. This means that enumerating the boundary edges can be done very easily without searching through every half-edge. Additionally testing to see if the surface is fully closed is easily done by testing for the number of elements in the hash table.

Using the unlinked half-edge hash table in conjunction with the circulator allows for any hole in the object to be found in linear time (proportional to the number of half-edges that border the hole). This combined with the linear time enumeration of boundary edges means that the boundary tracing time can be reduced by a factor equal to the ratio of boundary edges to non-boundary edges.

B. <u>Inclusion of non-orientable surfaces</u>

A non-orientable surface is one in which the two sides of the surface are indistinguishable from each other, for example the Mobius strip. What this means is that for a non-orientable surface mesh it is valid to say either side of any primitive is the outside or visible side. This raises a problem for the half-edge data structure because each primitive’s half-edges are oriented in a counter clockwise direction, but in the case of a non-orientable surface a clockwise orientation is also valid. This can be avoided by not culling either face (counter clockwise or clockwise) but there is still a problem with the linking of the half-edges. Consider figure 6, a modification of figure 4, which has two adjacent triangles, one oriented counterclockwise, one clockwise. The two adjoining half-edges are in the same direction which results in an invalid half-edge data structure. It is for this reason that half-edge data structures do no support non-orientable surfaces. By removing this restriction, and implementing the algorithms discussed in the following sections, this problem can be eliminated.

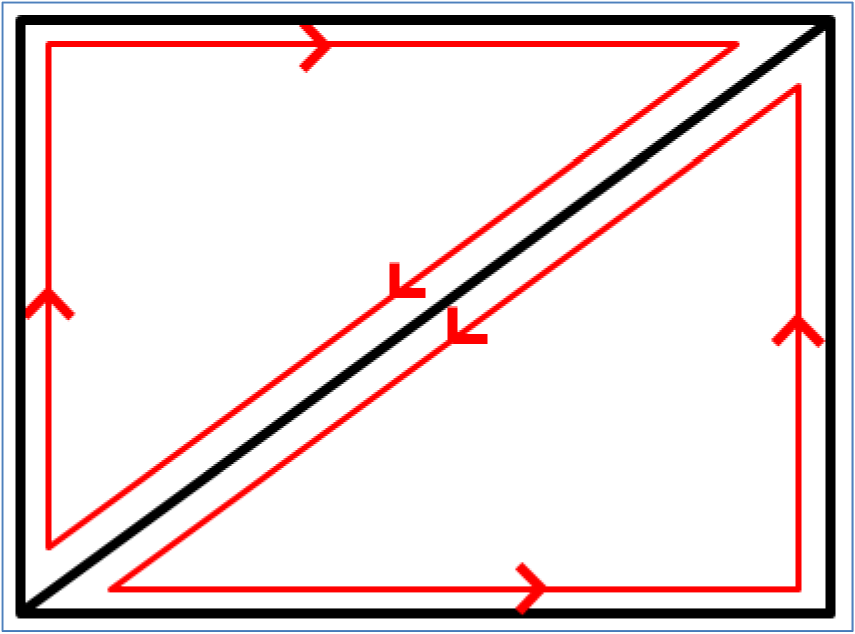
Figure 6 Non-orientable Triangular Orientation

C. Mesh Manipulation and Simplification

A simplified mesh has many benefits, in particular it allows for faster collision detection, less memory expense, and faster analysis. There are two ways to simplify the mesh, the first is to use the mesh before subdivision or processing, but in some cases the mesh does not have the desired visual quality or it contains redundant vertices, half-edges, or faces. The second option is to apply the methods described below to the mesh after subdivision so that it is simplified yet maintains its shape, and smoothness.

The new half-edge data structure now allows more a great deal more in the way of mesh manipulation. This application requires the ability to dynamically add and remove faces and vertices to the mesh. The previous implementation was not capable of these dynamic removals, and the dynamic addition of a face was much too slow.

As mentioned previously, the time complexity of adding a face to the mesh is constant where it was previously linear. This was accomplished using the hash table for unlinked half-edges. In addition to improving the time to add a face to the mesh the new implementation allows for a face to be removed from them mesh. During the creation of the branch structures it becomes necessary to remove a number of faces from the mesh thus a reliable method to remove faces and maintain a valid half-edge data structure was needed. Face removal is done by releasing the half-edges opposite of each half-edge in the face back to the unlinked hash table. Then, if necessary, a new half-edge is chosen for the vertices on the face. The new implementation also includes methods to remove vertices from the mesh, though currently it simply removes the faces incident upon that vertex.

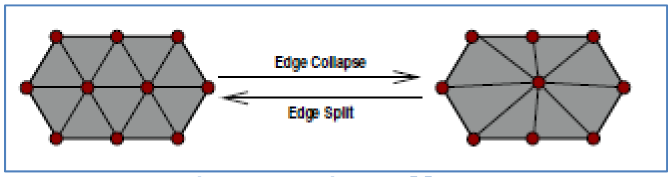

Figure 7 Edge Collapse

It is also valuable to collapse faces or half-edges in order to simplify the mesh. A half-edge collapse is shown in figure 7 (Widas, 1997) and involves merging the two vertices that make up the edge. This operation results in two degenerate triangles that are removed from the mesh, there are also two new vertices at the same position meaning one can be removed. Finally six half-edges can be removed from the mesh. The half-edge collapse is valuable when the two vertices provide very little information more than a single vertex would. Another reason to collapse a half-edge is if it is very short in comparison to the rest of the object.

A second valuable method that was implemented is the face collapse method; a face collapse is related to the half-edge collapse because it collapses all three edges of a triangle to the centroid of that triangle. The face collapse results in 4 degenerate triangles (3 lines and 1 point) allowing them to be removed from the object. The face collapse also results in removal of two vertices and 12 half-edges.

D. Circulator

There are many calculations that are based upon all incident faces or neighboring vertices (e.g. the vertex normal) and it is necessary to enumerate each of these faces or vertices. A circulator is an interface that allows for this, given a vertex it can find the incident faces or neighboring vertices in linear time. It makes use of algorithms 3 and 4 and abstracts them to an easy to use interface.

In the C# language there is a construct called the enumerator which allows for iteration using the ‘foreach’ loop. The enumerator interface is used to simplify iteration over a set of objects, in this case faces or vertices, using the following syntax.

```
foreach (Vertex v in Mesh.Vertices) {
      //do something to each vertex
}
```

The new data structure implements these enumerators for both the incident faces, and the neighboring vertices. This improves the usability of the data structure as well as the readability because it more closely resembles human speech than the typical for loop. Additionally it abstracts the initialization and iteration away from the user preventing errors from arising in the case that the initialization or iteration is performed incorrectly.

As mentioned the new data structure is capable of handling non-orientable surfaces. To accommodate these surfaces, a modification had to be made to the circulators; these modifications can be seen in algorithms 5 and 6. Each circulator works under a similar structure to the winged-edge circulators; specifically the modified circulators use the vertex they are circulating about to determine which direction to go next, rather than always going counter clockwise as they previously did.

```
List<Face> adjacent_faces(Vertex v)
{
      List<Face> F;
      HalfEdge e =  v.halfedge;
      //e must point to v
      do {
            F.add(e.face);
            if (e.head == v)
                  e = e.next.opposite;
            else
                  e = e.prev.opposite;
      } while (e != v.halfedge)
      return F;
}
```

Algorithm 6 Improved Adjacent Face Cicrulator

```
List<Vertex> neighbor_vertices(Vertex v)
{
      List<Vertex> V;
      HalfEdge e =  v.halfedge;
      //e must point to v
      do {
            V.add(e.tail);
            if (e.head  == v)
                  e = e.next.opposite;
            else
                  e = e.prev.opposite;
      } while (e != v.halfedge)
      return V;
}
```

Algorithm 7 Improved Neighboring Vertex Circulator

V. 3D Tube Networks Mesh Generation

A graph is a set of vertices, some of which are connected in pairs, these connections are called edges. A directed graph is a special type of graph in which the pairs of connected points are ordered. The purpose of this application is to start with a directed graph and, through the algorithm that follows, build a 3D tube mesh that is representative of that directed graph. This is a multistep process that will be discussed throughout this chapter. This chapter will also discuss in detail the implementation of the algorithm, and how the data structure is used to allow for the creation of the tube mesh.

A. Algorithm

The creation of a tube mesh starts first with a conversion from the directed graph into a new structure called a ‘tube’ each tube consists of branches, nodes, and the directed connections between them. The tube construct is very similar to that of a directed graph, the main differences being that vertices of degree 3 or higher are considered branches (the rest being nodes) and no two branches can connect; instead they must have a node placed between them. The algorithm to do this was provided prior to the project and as such this paper will not cover it in depth. However, it is important to know that each branch has in-nodes and out-nodes much like a directed graph. Algorithm 7 below shows the process by which this tube construct is converted into a tube mesh.

```
mesh convert_tube(tube)
      foreach (Branch b in branches)
            convert_branch(b);

      foreach (Branch b in branches)
            foreach (Node n in b.out_nodes)
                  convert_node_path(n);

      foreach (Node n in nodes)
            if (n is not converted and n has no in node)
                  convert_node_path(n);

      foreach (Node n in nodes)
            if (n is not converted)
                  convert_node_path(n);


convert_node_path(n)
      convert_node(n);
      if (n has no out node) return;
      if (n’s out node has been converted) return;
      n2 := n's out node;
            convert_node_path(n2);
```

Algorithm 8 Tube Conversion

Once the tube construct is created, the mesh conversion process takes place. First, since no two branches connect each one of them can be converted individually with no interaction; this is step 1 in figure 8 and the exact process will be detailed in the next subsection. Once each branch has been created the paths must now be constructed between each branch. Recall that each branch has both in-nodes and out-nodes, only the paths beginning with out-nodes need to be converted, this is because the in-nodes will either be the end of a path between branches, or will be handled at a later stage.

Each path is created by recursively converting nodes along the direction of flow in the tube until a destination branch is reached. In figure 8, the first node to be converted would be node 2 because it is the first out node of the first branch node. Next the algorithm would convert the node path beginning with node 3 (step 3A). Since it is not the end of a path the path conversion method will recursively convert the path beginning at node 4 (step 3B). At this point the path has been fully converted, and the algorithm moves onto step 4, the conversion of the final out-node. This process will continue until each out-node and its path have been converted, more detail on how each node is converted will be given in a later subsection.

At this point each out-node has been converted, and all that is left are the special cases where the flow starts at a node (i.e. a node with no in-node) and the nodes that are not connected to any branches. This is step 5 in the example provided by figure 8. Node 1 will be converted and the process will continue as described in algorithm 6.

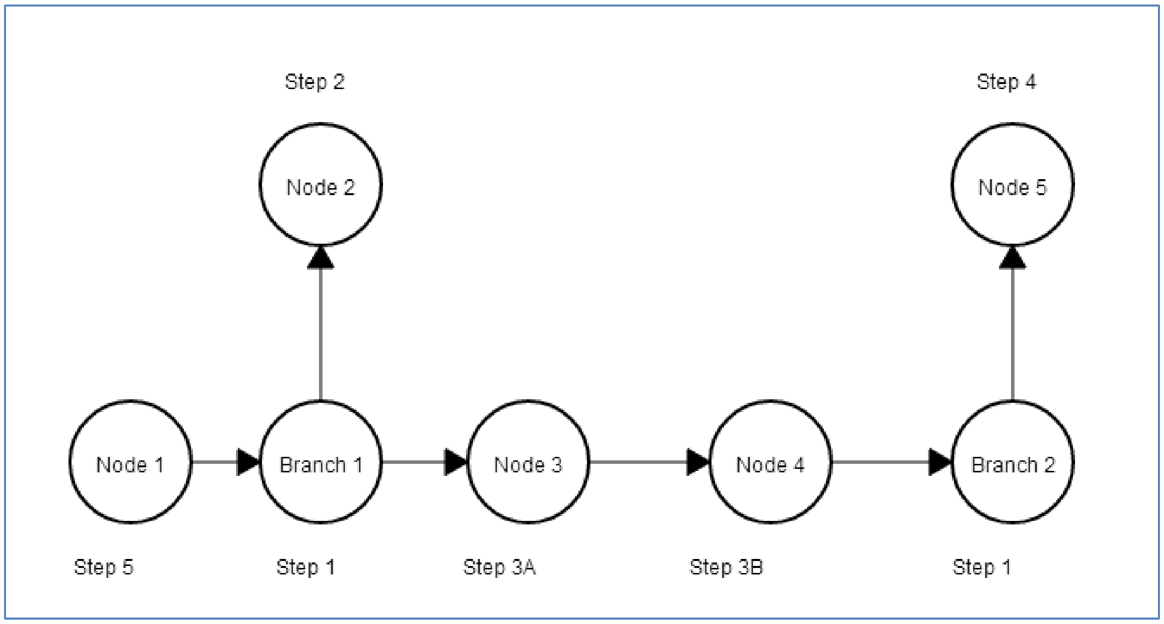

Figure 8 Sample Branch Structure

Now that each node connected to a branch has been converted the nodes which are disconnected from the branches are converted. This process is similar to step 5 in figure 8, a node with no in-node is converted and it simply reverts back to the path conversion portion of algorithm 6.

Finally we have the special case in which there is a ring of nodes, meaning each node has an in-node. If this case occurs any node in the ring is chosen and converted, then the nodes are converted in order around the ring.

1. Create Branch

The most important part of the process is the creation of the branch meshes; this not only takes the most time, but allows for the greatest variation in quality of the mesh. There are six major steps in this process which can be seen in algorithm 7.

```
mesh convert_branch(b)
Shpere s
foreach (Node n in neighbors)
      c = Create_Cross_Section(n)
      Project_Cross_Section(c, s)
Create_Convex_Hull(s)
Remove_Faces(s)
foreach (Node n in neighbors)
      Create_Extrusion(n, s)
      return s.mesh
Manipulate_Sphere(s)
```

Algorithm 9 Branch Conversion

The first step is to create a cross-section for each node, this can be of any shape and will be representative of the node that neighbors the branch. This node is then projected onto a sphere of sufficient size so that there can be no overlap of vertices among cross sections. Next a convex hull is created containing each of the vertices, the resulting mesh contains each of the original cross-section projections that were created in step 2. This must be ensured so that each of these projections can be removed in step 4 which will allow for the extrusions to be created in step 5, these extrusions represent the links between each node and the current branch node. The final step is to manipulate the sphere so that it becomes small enough to not envelop the extrusions but large enough to maintain consistency. This is where much of the variability comes from, the amount and method to manipulate the branch is what allows for many different results from the same algorithm. The method that is used in this implementation is unchanged from the previous application, but should be examined in the future.

2. Create Node

The final part of the algorithm is to create the node extrusions for every remaining node, this is done by connecting sequential nodes together using an extrusion similar to the extrusion created between each node and its branch. There are two steps that are involved in this process, the first is to create a cross section at each of the two consecutive nodes if one does not already exist, and the second is to actually create the extrusion.

B. Implementation

The visualization portion of this project is being done using XNA which is a free platform that makes it much easier to create games or graphical environments. For this project the XNA framework was placed into a control so that the program can have a menu system as well, this menu system allows for many things to be done using the half-edge data structure and mesh creation algorithm. The previous application used open-inventor for its menu system which will be replaced using window forms which would allow the application to be closed source if desired. Additionally since XNA is being used for the visualization only, the important parts of the application can be run without using any external libraries which satisfies one of the conditions this project had at its inception.

1. <u>Debugging</u>

One useful addition to the application was a method for debugging visually using incremental construction of the mesh. The added method allowed for one to view the step by step process of a the creation of a single branch or to see the step by step process to create the overall structure, essentially one can step through either algorithm 8 or algorithm 7 to see if there are any errors during construction or to determine when the error occurs. This has proven very valuable during the initial implementation because it is much easier to determine if there are error visually rather than to analyze the mesh structure at each step.

C. <u>Reduction of twisting</u>

One problem that was encountered during development is that along certain paths, the tube structure would become twisted resulting in meshes that intersected in on themselves or looked like figure 9, in that they were now concave in some cases, when they should always be convex.

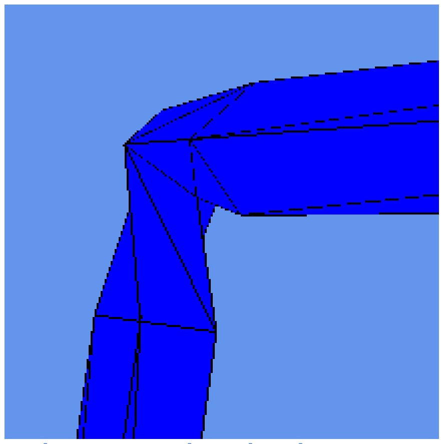
Figure 9 Twisted Tube

This problem was solved using a technique called local alignment which attempted to rotate the vertices of each cross-section to have as close as possible orientation to the previous cross-section, this is done by rotating the cross-section about the axis created between the two an amount that minimizes the angle between two chosen vertices, there is a second technique called global alignment that would cause all cross-sections to have the same alignment, but that technique has not been implemented and should be considered a future possibility.

D. Improved creation of two-branch pipes

In some cases the path created between three nodes can have very poor mesh quality, this can be due to any number of reasons, but is usually attributable to a sharp bend between the three, this results in the cross-sectional area being reduced no nearly zero in some cases which would not happen in nature. To prevent this defect three alternative methods were investigated, the first method was to use an additional virtual node that was equidistant with maximum distance from the first two nodes, however for this node no extrusion was made. This had the effect of creating a three-branch with only two extrusions and resulted in a mesh with more consistency and natural appearance as shown in figure 10.

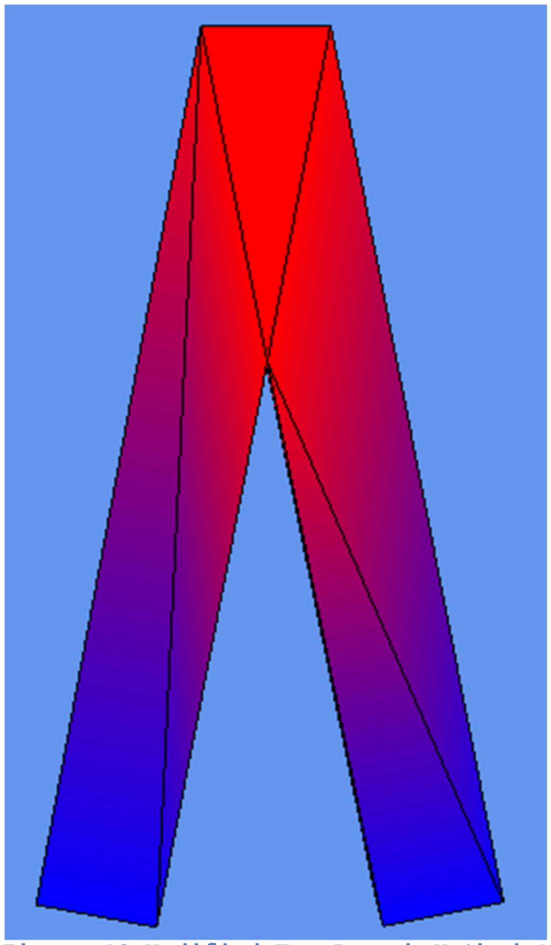
Figure 10 Modified Two-Branch Method 1

A second method investigated was to use an additional cross section in the same location as the previous method, but for this method the cross-sections are manipulated using a different method. First instead of moving each vertex independently the vertex groups were moved together to maintain a square cross-section. Next the real branches that will soon have extrusions were moved inward the maximum possible amount without causing overlap. Finally the virtual cross section was moved to be a distance away from the actual cross-sections that at its shortest is equal to the radius of either the largest node, or the radius of the branch. An example can be seen in figure 11.

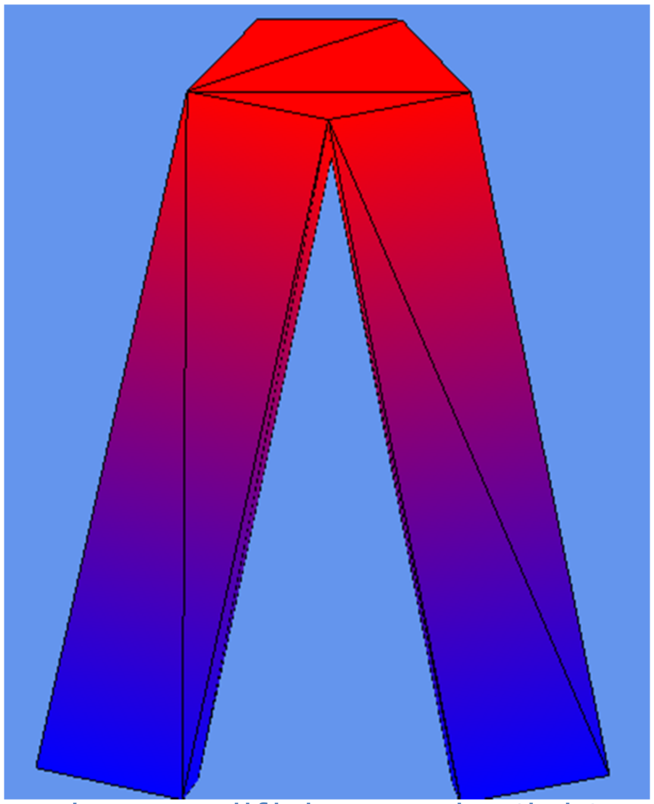

Figure 11 Modified Two-Branch Method 2

The next attempt was directed toward making each branch node more spherical, and this was done by attempting to add more vertices to the convex hull creation. This method did little to solve the problem resulting from having too tight a bend in the path, but it did have a nice effect for some other structures, one such structure can be seen in figure 12 below. In order to determine the vertices that will be added to the hull a set of vertices that are approximately equally distributed was created, next each of those vertices was tested to see if it would cause a cross section to not be present in the convex hull. If this is the case the vertex was removed and the process carried out normally. The green vertices in figure 12 are the vertices that were added by this process.

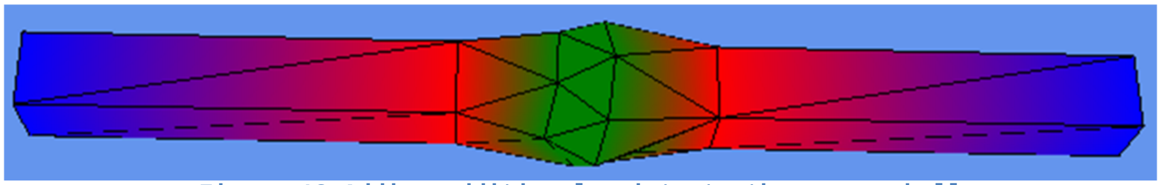
Figure 12 Adding additional points to the convex hull

E. <u>Convex Hull</u>

One of the important steps of the creation of the 3D tube mesh is the convex hull algorithm, the convex hull algorithm accounts for a majority of the time for the entire process to complete, this is because a quadratic algorithm is being used because it allows for easy debugging. Additionally the small number of points for most cases means that it may be beneficial to continue using the incremental algorithm rather than introduce a divide and conquer algorithm that may have more overhead.

It should be noted that the convex hull creation is actually a special case in that each point will be part of the final hull and each point already lies on a known sphere. This means that there may be a more efficient algorithm that applies only to this special case that can be used.

VI. Results

One of the major goals of this project was to decrease the time it takes for the steps involved in the creation of these tube structures. There are two applications that this new implementation will be compared against, the original implementation in C++, and the initial implementation in C#. The latter only being suitable to make comparisons in subdivision of surfaces, and general surface creation time due to the early nature of the implementation (at the time this project started the tube creation had not yet been implemented in the C# version).

The first comparison will be that of the general surface creation time, this is the time to load in an off file and convert it to the half-edge data structure. The comparison will be made for four spheres having 64, 256, 1024, and 4096 vertices. Table 1 shows that with very few vertices the improvement is very little but as the number of vertices increases the time cost savings becomes extremely large. This is because the new algorithm is completed in linear time rather than quadratic.

Table 2 Comparison of Mesh Creation Times

Vertices	Initial (ms)	Improved
64	7	6
256	30	8
1024	385	13
4096	5412	40

A. Subdivision

Next we will compare the results of the subdivision algorithm being used in the two C# implementations, both use loop subdivision which allows us to compare the two on the basis of their implementations. For this comparison four spheres will again be used this time 64, 144, 256, and 400 vertices.

Table 3 Comparison of Subdivision Times

Vertices (before)	Vertices (after)	Initial (ms)	Improved (ms)
64	270	25.5166	14.4952
144	598	101.8767	16.1378
256	1054	291.7180	18.9798
400	1638	701.2819	22.9138

The improved implementation offers an extreme time cost savings over the initial implementation due to its linear running time as opposed to the quadratic running time of the initial implementation. Table 4 shows that the new implementation also offers a significant speed up over the C++ implementations, the speed up factor is in some cases as high as 3.47.

Table 4 Comparison of Subdivision Times

File Name	C++ (ms)	C# (ms)	Speed Up
eyespline	51.93	14.92	3.47
Lattice_10_8_3	2436	1072	2.26
Mobius_16_16	902.6	321.5	2.81
polyhedron	203.6	60.88	3.23
Tree3	260.6	76.83	3.39

B. Tube Creation

In the previous two sections comparisons were being made to an unfinished implementation and largely the improvement was due to improvements in the implementation rather than improvements in the data structure. For this section a comparison between a finished implementation and the improved version being presented will be made. This comparison is to ensure that the new version is at least as good as the previous version which would show that the new version is an acceptable alternative that runs using C# and can be easily modified and used (as shown previously).

For this section five directed graphs are used, each one goes through the entire process in each application and the total time to completion will be compared. Additionally both resulting tube structures will be shown in an effort to prove the efficacy of the improved implementation. Table 5 shows the speed up offered by this new implementation in the creation of the tube structures. The newer implementation offers as much as 7x speed up over the C++ implementation.

Table 5 Comparison of Tube Creation Times

File Name	C++ (ms)	C# (ms)	Speed Up
eyespline	68.62	9.540	7.193
Lattice_10_8_3	3231	654.3	4.939
Mobius_16_16	1354	221.8	6.104
polyhedron	320.5	47.09	6.805
Tree3	255.9	49.89	5.130

C. Non-Orientable Surfaces

The final improvement over previous attempts is that the proposed implementation has support for non-orientable surfaces such as the Mobius strip, the Klein bottle, and any other constructed surface that does not have neighboring faces with the same orientation. This section will present each of these types of surface as well as the result of these surfaces after subdivision.

The purpose of showing these surfaces after subdivision is so that it can be seen that the faces stay connected and can reference each other (if they were not connected the subdivision algorithm would shrink them apart in a way similar to what will be seen at the edges of the surface. Figure 13 shows the original surface with no culling and figure 14 shows counter clockwise culling, this is done to show which faces are oriented in which directions. Loop subdivision is performed and the results are shown in figures 15 and 16, as you can see all of the connections remain indicating that the components of subdivision (including the data structure) do not break when using non-orientable surfaces. Figures 17-20 are included to further demonstrate this achievement; additionally, for each surface the data structure was checked to ensure that it was valid and connected as it should be.

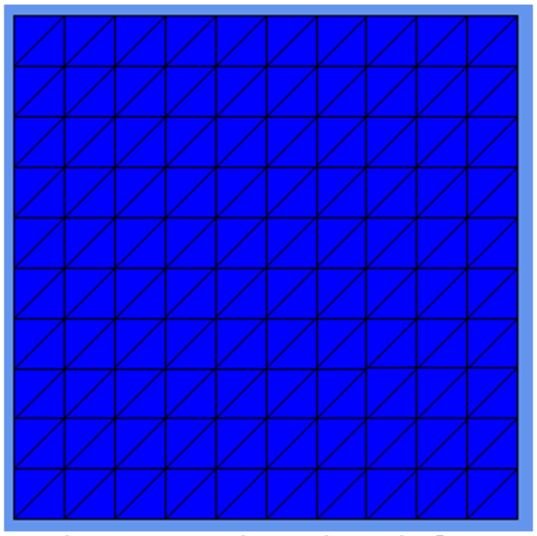
Figure 13 Random Oriented Plane

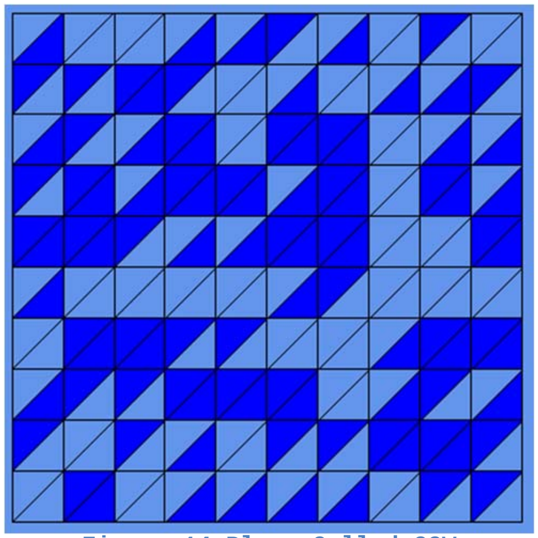
Figure 14 Plane Culled CCW

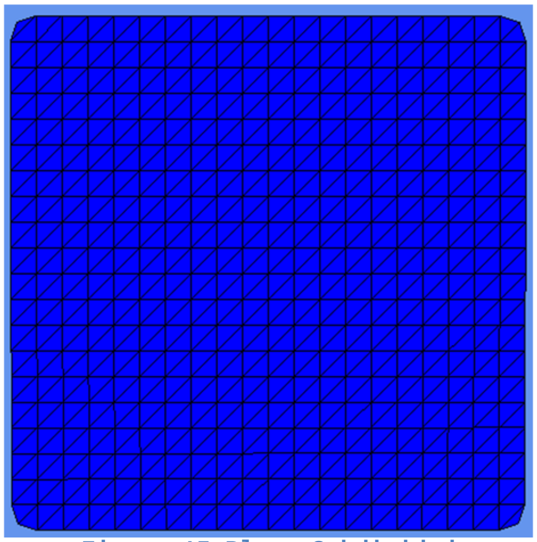
Figure 15 Plane Subdivided

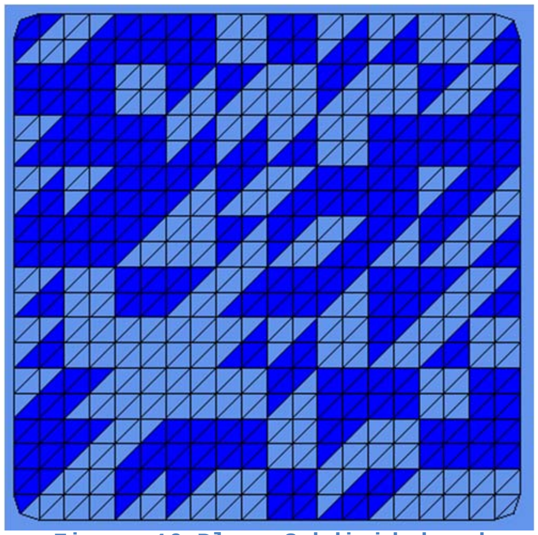
Figure 16 Plane Subdivided and Culled

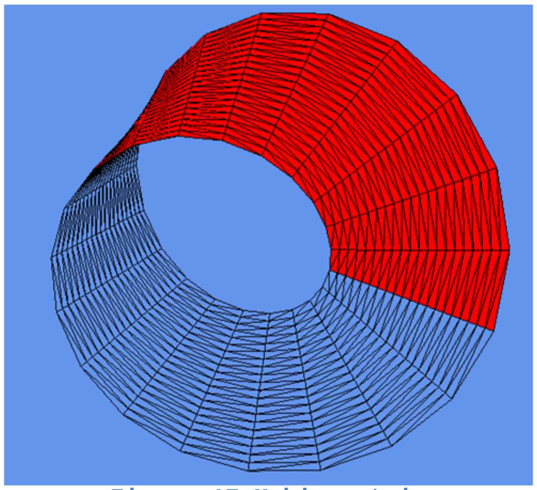
Figure 17 Mobius strip

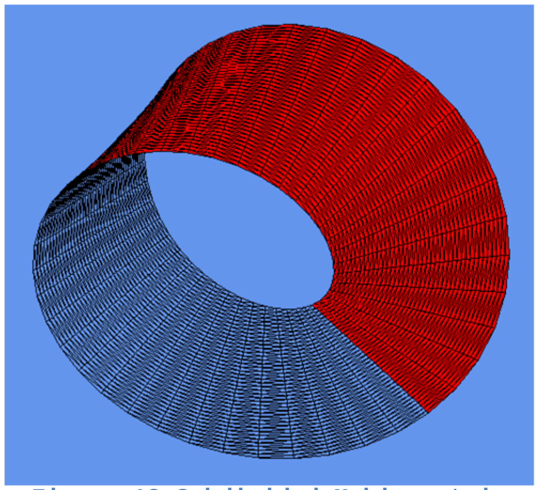
Figure 18 Subdivided Mobius strip

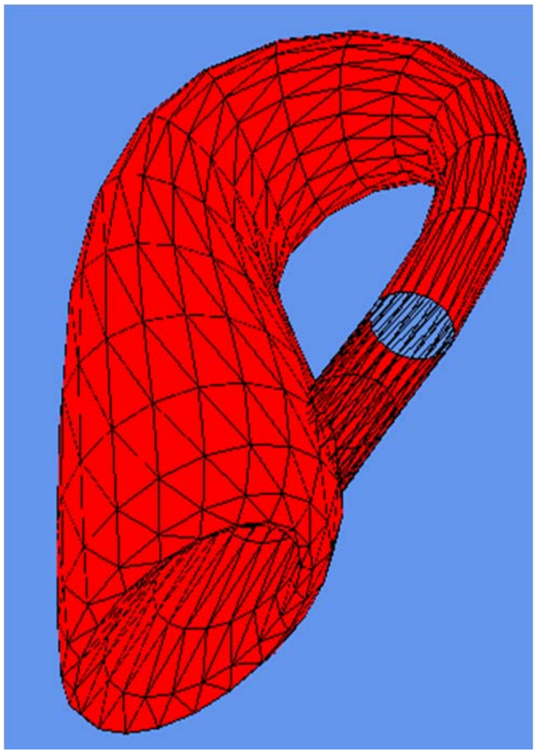
Figure 19 Klein bottle

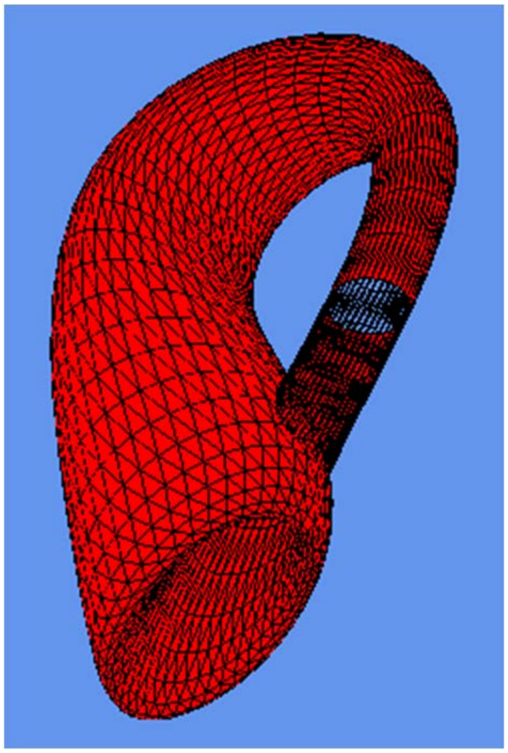
Figure 20 Subdivided Klein bottle

D. Evaluation of proposed Data Structure

In this section the data structure being proposed will be compared to the alternative implementations that are currently in use, namely CGAL and OpenMesh. They will be compared on the basis of the access, modification, operations, parameterization, and input/output.

The first criterion is access, this metric has many simple aspects such as access of vertices, edges, and faces which are the basic elements and many other complex aspects. First we look at the simple, each data structure has references to lists of vertices, edges, and faces. In this regard there is no advantage that could not easily be corrected by adding an additional reference that is not present. The advantage lies in the complex aspects of the access metric; the proposed data structure allows for the quick access to boundary edges using the unlinked half-edge hash-table. A second advantage lies in the ease of use of the circulator in the new implementation which uses an enumerator to make the access much more natural and simple.

The next criterion is modification, which relates to the ease and ability to modify the mesh by adding and removing elements. The requirements of a half-edge data structure is that after any modification by the user, the data structure remain consistent. Each of the three data structures allow for the addition and removal of faces and vertices and each guarantees that the mesh stay consistent after the operations. Each data structure has roughly the same ability to modify the mesh so while there is no improvement here, there is no degradation either.

The most important criterion is the ability to perform necessary operations on the mesh, it is here that a great deal of improvement over the initial C# implementation was done. This metric contains such operations as half-edge collapse, face collapse, and other simplification methods that should be included in the data structure. For the proposed data structure the important simplification methods were included, but time limitations meant that only some could be implemented. More importantly thought the ability to create a convex hull in the half-edge data structure and the ability to perform subdivision were included. These are necessary parts of the application being improved upon and as such were implemented first and carefully. So every necessary operation is included but not every available operation, for this reason CGAL and OpenMesh are still better alternatives in some applications per this metric.

The next criterion is parameterization, which relates to storing necessary additional data inside each vertex/edge/face. It is here where CGAL offers a much better alternative using template classes for additional robustness. However the proposed application needs only to store certain variables for its completion and these can be added prior to compilation time in order to accomplish this goal. In the future it will be valuable to add these aspects but for now the data structure offers enough to complete the process.

The final criterion is input/output which means that the mesh should be easily converted and stored to many file types. This data structure allows for one to save only to an .off file, but allows for reading from file types such as .tub, .gr, and .off. For this reason the data structure again meets the criteria to be a half-edge data structure and offers some benefits above the default implementation of CGAL and OpenMesh.

VII. Recommendations

A. Convex Hull Creation

As mentioned, the convex hull that is created is a special case in which all of the points are on the same sphere. It should also be restated that the convex hull construction is the most expensive method in the implementation and has, by itself, a quadratic time complexity. More research should be done into determining if the special case for the convex hull allows for a simpler method to be used. Even if no such method can be found there are faster convex hull methods than the current implementation; the incremental method was chosen for its simplicity and the ability to see the construction of the convex hull as it occurs. A new method should be implemented in which the goal is speed rather than simplicity.

B. Texture Assignment

Video games require texturing for visual quality; a simple mesh lacks the visual effect afforded through the use of texturing. It may be valuable to find a way to programmatically texture or color the given mesh.

VIII. LIST OF REFERENCES

Hughes, Graham. (2011). How big is 'big data' in healthcare. Retrieved from http://blogs.sas.com/content/hls/2011/10/21/how-big-is-big-data-in-healthcare/

Kajak, Bartosz. (2011). *Improved algorithms for ear-clipping triangulation.* (Master of Science), UNLV, Las Vegas. Retrieved from http://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=2314&context=thesesdissertations

Kettner, Lutz. (2012). Hlafedge Data Structure. 2012, from http://www.cgal.org/Manual/latest/doc_html/cgal_manual/HalfedgeDS/Chapter_main.html

Kobbelt, Mario Botsch; Mark Pauly; Christian R¨ossl; Stephen Bischoff; Leif. *Geometric Modeling Based on Triangle Meshes*.

Leadwerks. (2006). What is Constructive Solid Geometry. 2012, from http://www.leadwerks.com/files/csg.pdf

Levoy, Marc. (2011). The Digital Michelangelo Project. 2012, from http://graphics.stanford.edu/data/mich/

Marshall, David. (1997). Boundary Representation. 2012, from http://www.cs.cf.ac.uk/Dave/Vision_lecture/node57.html

Widas, Peter. (1997). Introduction to Finite Element Analysis. 2012, from http://www.sv.vt.edu/classes/MSE2094_NoteBook/97ClassProj/num/widas/history.html

Zorin, Denis. (2004). Mesh Data Structures. 2012, from http://mrl.nyu.edu/~dzorin/ig04/lecture24/meshes.pdf

IX.

CURRICULUM VITAE

NAME:	Richard Paris
ADDRESS:	1706 Tempest Way Louisville, Kentucky 40216
EDUCATION:	B.S. Computer Engineering & Computer Science University of Louisville 2012
	B.S. Electrical & Computer Engineering University of Louisville 2013