Abstract

This project implements a triangular mesh simplification algorithm based on the work of Garland and Heckbert[2]. Using a system that prioritizes edges for removal based on maintaining the model's shape, the program can transform high facecount models into simpler approximations. Ultimately, these simplified meshes are intended to be used as input to a physics engine, providing a manageable model for the physical interactions of mesh shapes.

Introduction and Motivation

Display of three-dimensional models in real-time is often constrained by the number of triangles present in any particular scene. Though graphics coprocessors have increased rendering performance dramatically, the largest scenes can still be problematic even for modern systems. Often, a large number of polygons are unnecessary, as the model occupies an insignificant portion of the physical screen space, due to the distance from the viewing camera. Moreover, simpler approximations of models may be needed for lower-performance systems, or for non-visible computations made using general geometry (such as computing object physics). As a result, model simplification provides a way to service these needs, providing a close approximation to the original model using much fewer polygons.

Previous Work

In 1976, Dr James Clark presented the idea of simplifying geometric detail to increase performance when rendering detailed 3D surfaces[1]. His description of hierarchically displaying variable levels of detail, using low-polygon models for objects that are perceptually distant from the viewer, set in motion the idea of object simplification. Early approaches to simplification involved an artist recreating the model at various levels of detail. However, the need for automated simplification became important as the decreasing cost of graphics coprocessors made the cost of hiring an artist more expensive relative to the overall project cost.

To meet this need, a number of solutions were proposed in the early 1990s. For example, the idea of “vertex decimation” was put forth as a means to preserve topology while simultaneously reducing the total complexity of the model[6]. In its original form, the average of all centroids surrounding a target vertex was compared with the target vertex. If the distance between these two points was less than a certain threshold, the vertex was removed, and the resulting hole filled in with replacement triangles. Though this was a fast method for simplification, the average of centroids did not preserve overall detail as well as it could.

Another method for simplification that came about at roughly the same time as vertex decimation is “vertex clustering”, which simplified vertices in a 3D grid’s cubes to the most “important” vertex[5]. Importance is defined as being attached to a long edge or being part of a sharp curve; ideally, this preserves features of the model to be simplified, but, in practice, the clustering of vertices without respect to their attached faces cannot guarantee topological consistency.

To produce simpler meshes that closely maintained topology, Hoppe et al. created a mesh optimization algorithm that used an “energy function” to determine how to best fit a small number of faces that most closely matched the input mesh[3]. Though the results were accurate, they were slow in relation to other simplification algorithms. Addressing the issue of speed-versus-quality, Garland et al. presented an algorithm for mesh reduction while minimizing the total mesh error, using a set of squared distances from the planes surrounding each vertex[2]. This algorithm produced high-performance, high-quality renders that had previously been unattainable, and is still the basis for most modern simplification techniques.

In the method outlined by Garland, each *Face* has a “fundamental error quadric” computed from its normal vector and distance from a common *Vertex*. The *Face*’s quadric is added to each *Vertex*’s quadric, giving a summation of planes around the *Vertex*. Between any two *Vertex* objects shared by a common *Edge*, a “best fit” point for contraction of that edge is found, first by attempting to solve the gradient for a minimized value, then using best fit along the line, then using the midpoint, stopping when one of the three methods succeeds.

Algorithm

Model Import and Internal Representation

Models are imported into the application and stored using a fully connected mesh structure, as shown in Figure 1. Each arrow represents a bidirectional pointer, where each object has a pointer to the other. The objects are called *Vertex*, *Face*, and *Edge*. A *Face* contains three pointers to *Edge* objects, and three pointers to *Vertex* objects. An *Edge* contains two pointers to *Face* objects and two pointers to *Vertex* objects. A *Vertex* object contains an STL vector for both *Face* and *Edge* objects surrounding it.

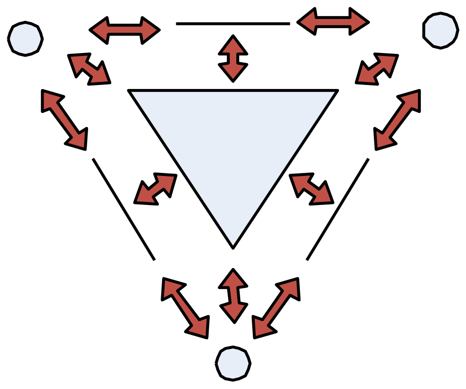
Figure 1 Internal Model Representation

Parsing a file into the application is handled by a factory class called *MeshParser*; it chooses format to parse based on file extension. For the simple model format (SMF, extension .m), two model constructs are currently supported: **Vertex** and **Face**.

Models returned by *MeshParser* support the *IMesh* interface, which provides common functionality for all meshes. This includes the ability to add vertices and faces, as well as functions to get iterators to the beginning and end of the *Vertex*, *Face*, and *Edge* lists. The interface also enumerates common "draw modes" for the model, such as faces, edges, vertices, lit, and shading.

Edge Collapse

Removal of an edge from the mesh structure is a multi-step process that requires updating all attached *Vertex* and *Triangle* objects. In Figure 2, the center (green) *Edge* is being collapsed. Stages of the collapse are as follows (visualization of this process shown in Figure 3).

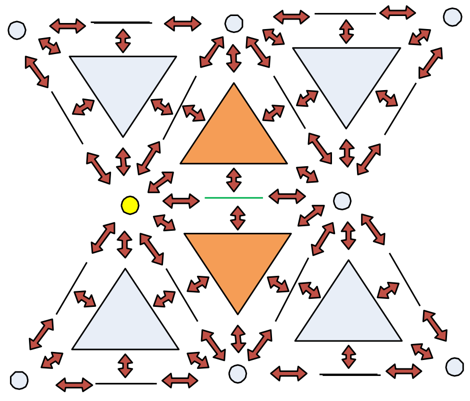
Figure 2 Edge Collapse Visualization

The ***Edge::Collapse()*** function marks the edge as removed, chooses a vertex to be the new common *Vertex*, then calls ***Face::Remove(Vertex* v)*** on each non-NULL *Face*, passing the selected *Vertex* as a parameter. This *Vertex* is needed to ensure the structure remains consistent, with surrounding *Face*s referencing the correct *Vertex*. Each *Face* is marked removed and the surrounding *Edge*s are iterated through, looking for the one being collapsed. The remaining two edges are checked for the passed-in *Vertex*, and the *Edge* not containing that *Vertex* is remapped.

The ***Edge::RemapTo(Edge* eNew, Face* fOld)*** adjusts the pointers around both *Edge*s. To perform this action, the *Edge* gets the "left" and "right" *Face* straddling the removed *Face*. If both are NULL, eNew is removed; otherwise, the pointer to fOld is changed to the "right" *Face* (this choice is arbitrary, as "left" and "right" are used for simplicity based on the above diagram, in reality fOld is changed to the *Face* not containing the chosen *Vertex*). If the "right" *Face* exists, ***Face::RemapEdge(Edge* eOld, Edge* eNew)*** is called, remapping this *Edge* to eNew.

After ***Face::Remove(Vertex* v)*** returns into ***Edge::Collapse()***, the *Vertex* being kept has its position set to the computed location. Then, the *Vertex* along the collapsed edge that is not being kept is remapped to the *Vertex* being kept. The ***Vertex::RemapTo(Vertex* vNew)*** function adds the computed cost and the normal vector to vNew, then iterates through all *Edge*s and *Face*s around it, remapping those that aren’t removed to point to vNew. Finally, all *Edge*s around vNew have their cost recomputed, as it has most likely changed since the remap.

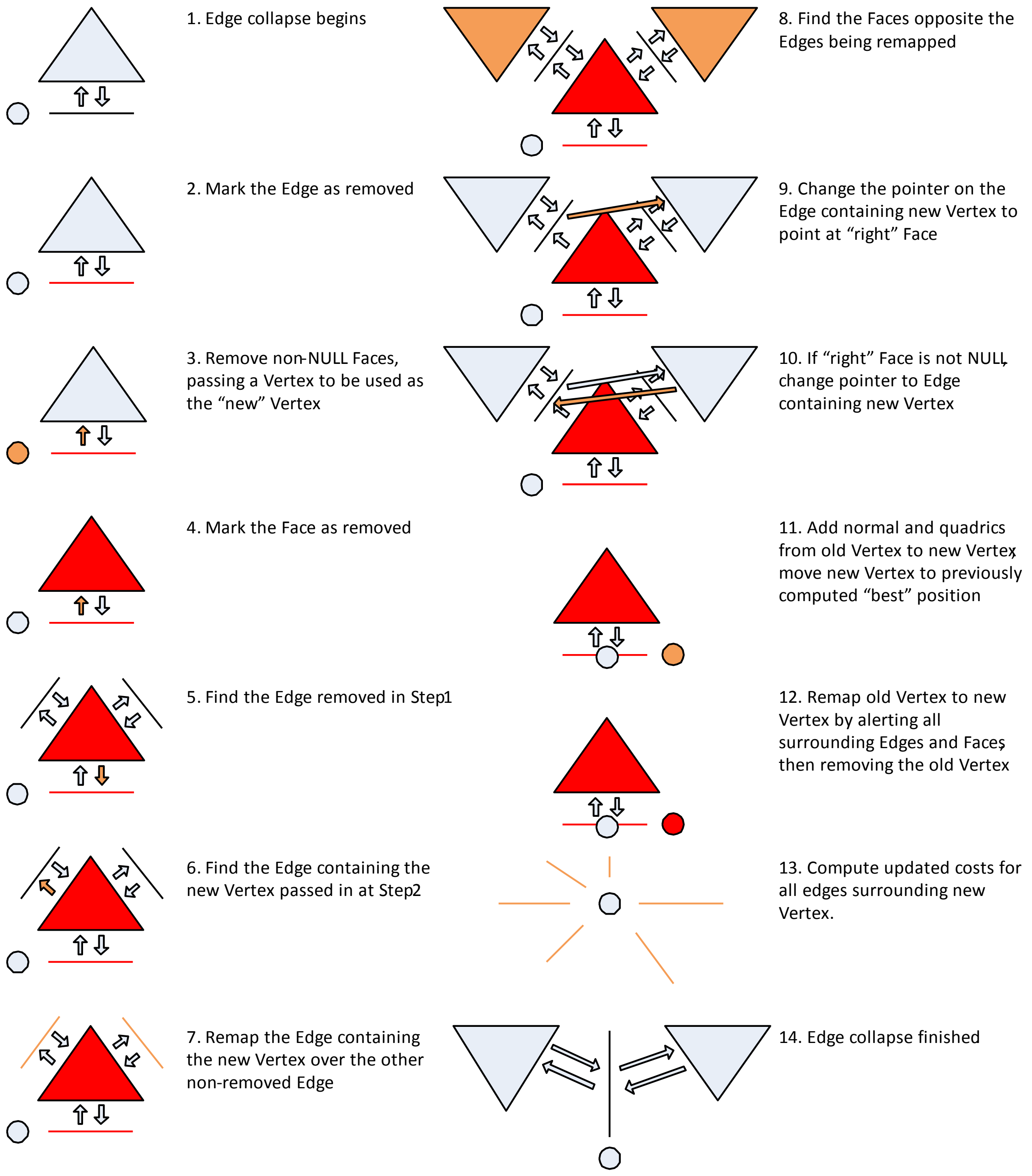
1. Edge collapse begins
2. Mark the Edge as removed
3. Remove non-NULL Faces, passing a Vertex to be used as the "new" Vertex
4. Mark the Face as removed
5. Find the Edge removed in Step1
6. Find the Edge containing the new Vertex passed in at Step2
7. Remap the Edge containing the new Vertex over the other non-removed Edge
8. Find the Faces opposite the Edges being remapped
9. Change the pointer on the Edge containing new Vertex to point at "right" Face
10. If "right" Face is not NULL, change pointer to Edge containing new Vertex
11. Add normal and quadrics from old Vertex to new Vertex, move new Vertex to previously computed "best" position
12. Remap old Vertex to new Vertex by alerting all surrounding Edges and Faces, then removing the old Vertex
13. Compute updated costs for all edges surrounding new Vertex.
14. Edge collapse finished

Figure 3 Visualization of an Edge Collapse

Results

Tests were run on an Intel Core 2 Duo at 3Ghz with 3.5Gb of addressable memory. No simplification took longer than 10 seconds, but no specific timing data is provided, as the overall speed was more than adequate. Meshes tested included “gameguy”, “bunny”, “dragon”, “gargoyle”, and “fandisk”. All meshes except “fandisk” could be simplified using the algorithm; fandisk’s structure caused a breakdown in vertex placement, leading to the broken mesh seen below.

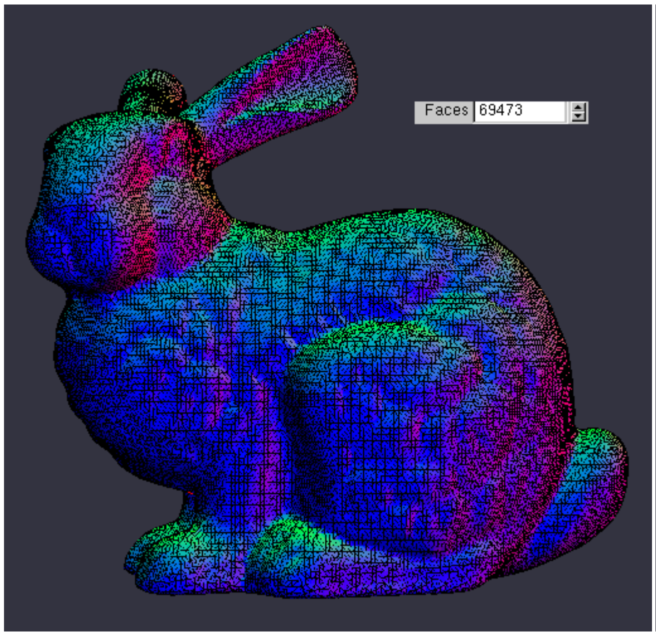

Figure 4 Original Bunny Model - 69,473 faces

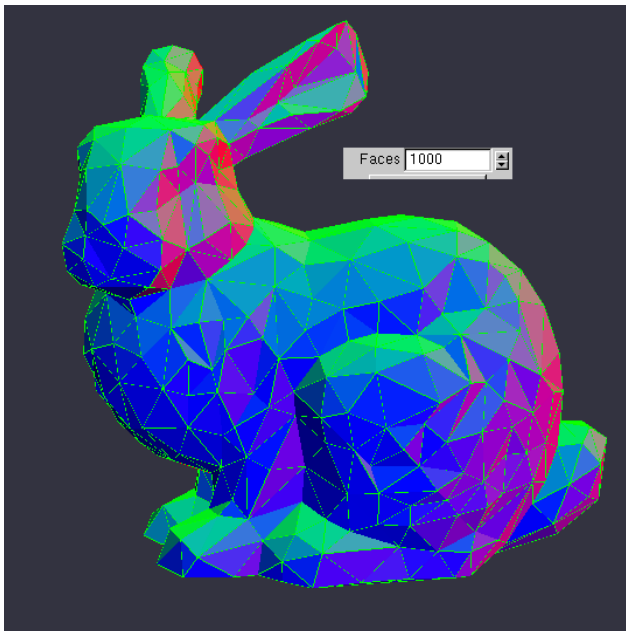

Figure 5 Simplified Bunny Model - 1,000 faces

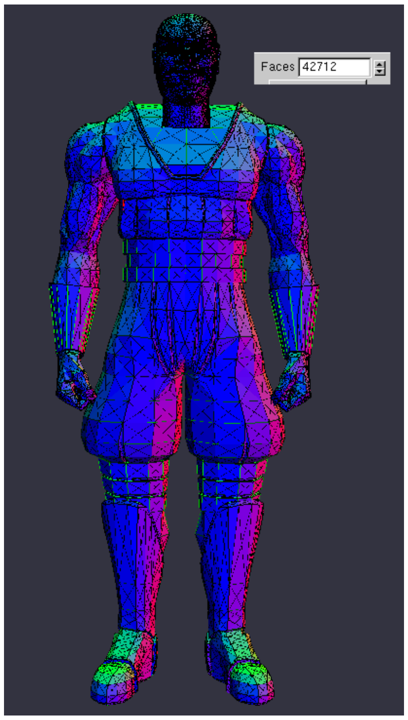

Figure 6 Original Gameguy Model - 42,712 faces

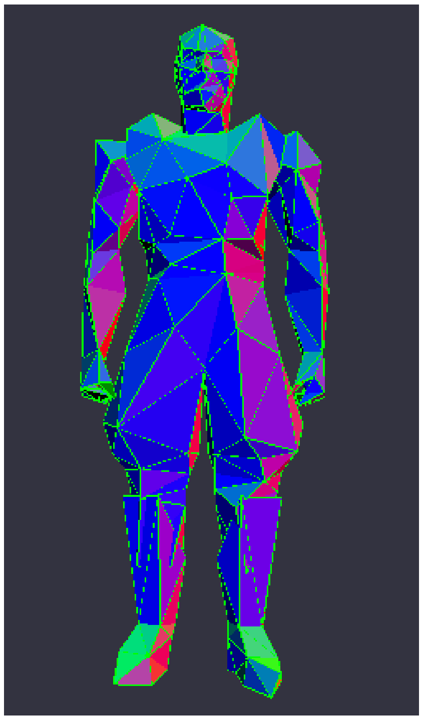

Figure 7 Simplified Gameguy Model - 500 faces

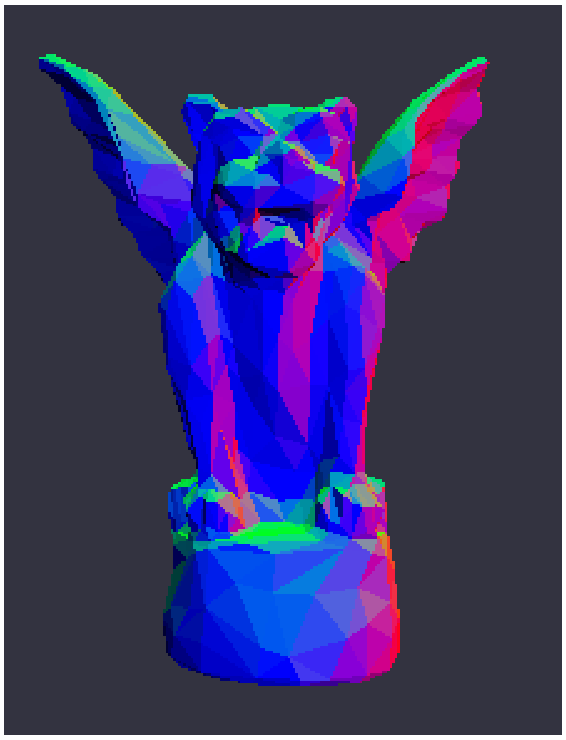

Figure 8 Original Gargoyle Model - 2,000 faces

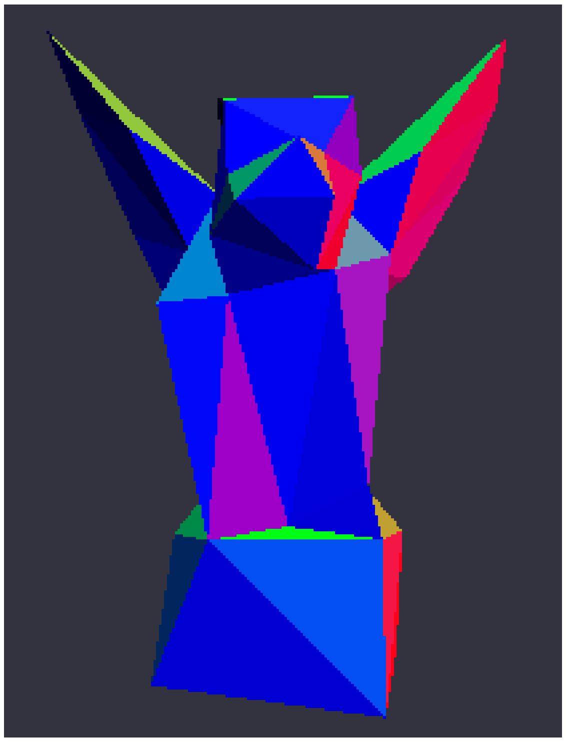

Figure 9 Simplified Gargoyle Model - 100 faces

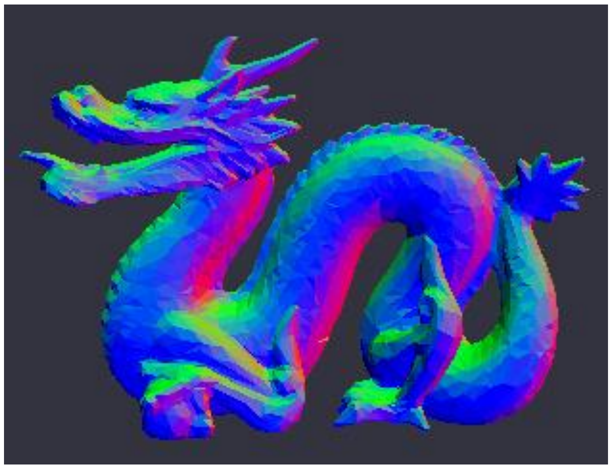

Figure 10 Original Dragon Model - 10,000 faces

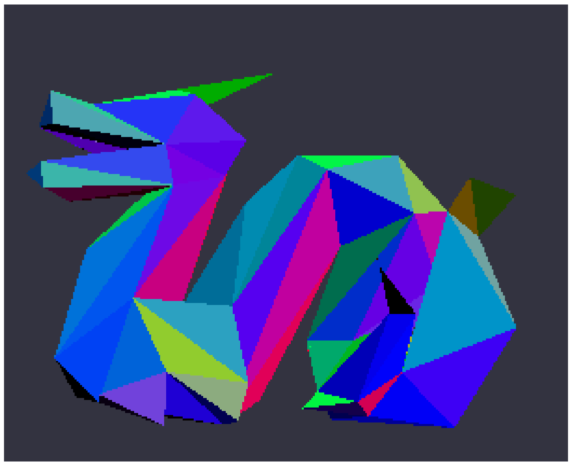

Figure 11 Simplified Dragon Model - 300 faces

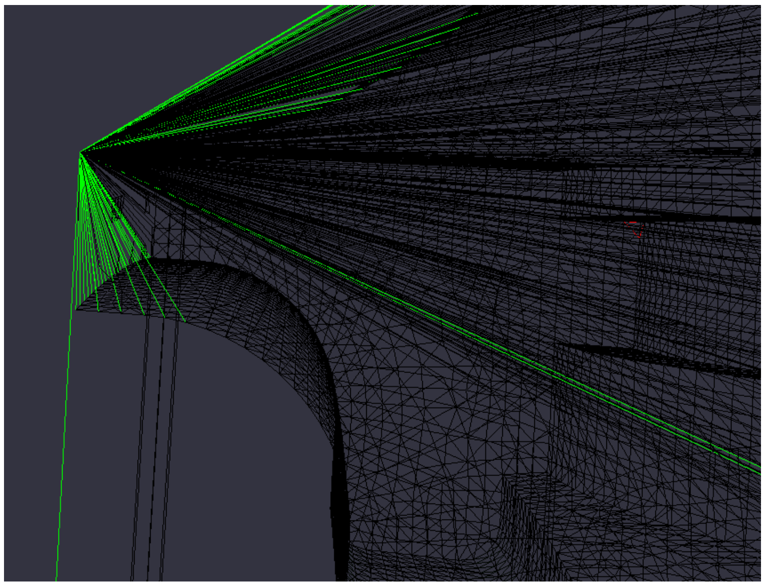

Figure 12 Fandisk Model - broken after attempted collapse

Future Work

Vertex Weighting

Though the current implementation provides a fast method to simplify models, it provides no ability to designate parts of a model as more important. Allowing the user to set arbitrary cost values on vertices or providing a flag to prevent collapse or movement of certain edges would allow for specialized feature preservation. It is important to note that only weighting the Vertex objects can be permanent, as an Edge object can be affected by the collapse of another Edge sharing a Vertex. By affixing the Vertex, all edges associated with that Vertex also become affixed.

Collapse History

Tracking the stages of a collapse allows a user to test the effects of custom weights. It also would provide a visualization of the stages of collapse, allowing a user to slide between collapse states. This could be done using progressive meshes[4], as described by Hoppe et al., allowing for smooth, visible transitions between mesh complexity states.

Mesh Inversion and Improved Feature Detection

The current method has no inversion prevention. This means that a mesh fold-over can occur, inverting the normal on a face due to a poor choice of collapse. To fix this, Face normals should be saved and compared before and after collapse. Any normal changing by too great of a value will cause the collapse to be reverted and a new collapse to be chosen. Collapse history must be implemented before mesh inversion prevention. Also, improved detection of notable architectural features can be added using Face normals.

References

[1] Clark, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Commun.*

[2] Garland, M. and Heckbert, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and interactive Techniques* International Conference on Computer Graphics and Interactive Techniques.

[3] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. 1993. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '93.

[4] Hoppe, H. 1996. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '96.

[5] Low, K. and Tan, T. 1997. Model simplification using vertex-clustering. In *Proceedings of the 1997 Symposium on interactive 3D Graphics* (Providence, Rhode Island, United States, April 27 - 30, 1997). SI3D '97.

[6] Schroeder, W. J., Zarge, J. A., and Lorensen, W. E. 1992. Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and interactive Techniques* J. J. Thomas, Ed. SIGGRAPH '92. Garland, M. and Heckbert, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and interactive Techniques* International Conference on Computer Graphics and Interactive Techniques.