Interactive Geometry Editor

Beth Werbaneth

Introduction

3D modeling software allows artists to realize their visions in a digital format. By manipulating points, edges, surfaces, users can create representations of various objects that can be exported for use in other applications. Modern modeling software like Autodesk Maya are notorious for bombarding the users with rows upon rows of buttons and toolbars and can prove difficult for the amateur user to pick up and use. In response to this, I created a simple program that provided a system of modeling and saving simple objects that could be exported to the Panda3D game engine.

This project aims to extend upon that earlier project by improving the subdivision scheme, implementing a simplification algorithm, and optimizing the performance of these operations to allow for interactive development. In order to provide a GUI for the user, this project was created using SFML 2.0.

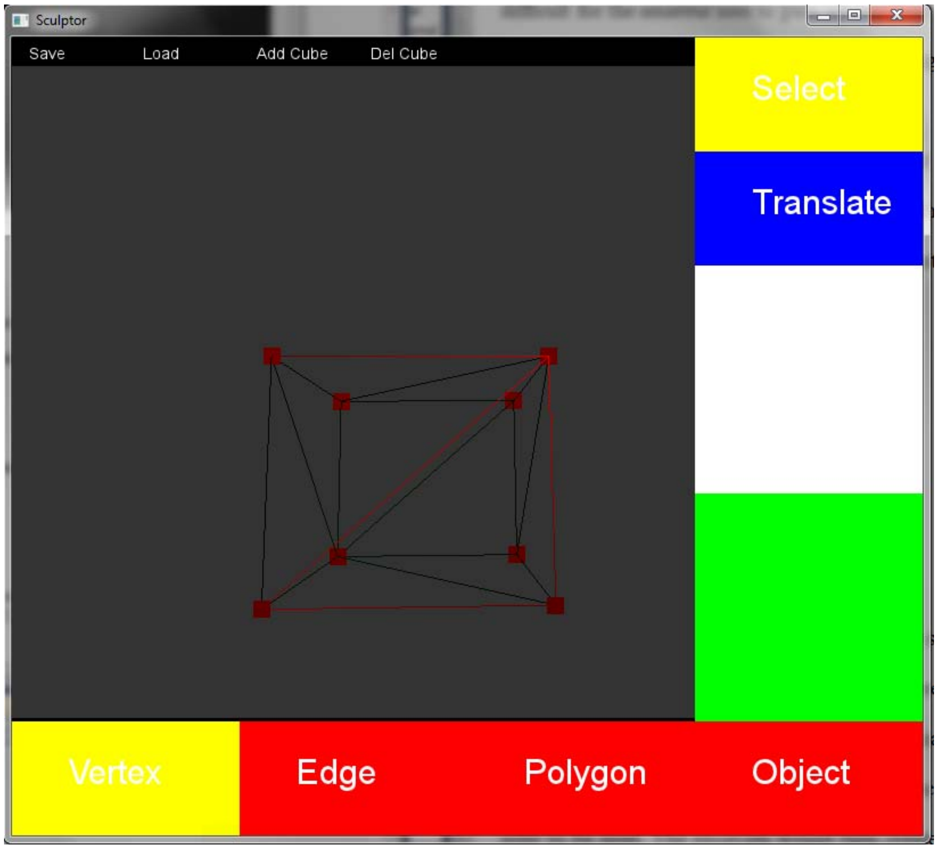


Figure 1: A simple modeling program

Program Overview

This simple modeling program is based around a starting cube which the user can transform into any object imaginable, given the time and patience. Using the bottom toolbar, the user switches between four primary control modes: Vertex, Edge, Polygon, and Object, which control how the object appears. While in Vertex Mode, the user can only select and translate vertices, but in the other modes, he can also apply rotation and scale transformations. Most operations in the program are tied to hotkeys, which allow the user to quickly switch between functions when editing the model. In some modes, additional collapse and subdivide options become available. The top toolbar allows the user to quickly save, load, and add and delete objects.

Subdivision

When modeling, artists often start with a simple primitive, which they will then subdivide to create more complicated geometry. With more faces, artists can exert finer control over their models. Subdivision is also useful for automatically creating smooth surfaces.

In the first iteration of the program, local division was implemented by specifying an edge to be split. The program would then create a new vertex V at the center point of the edge and make two new edges by connecting V with the unused points in the triangles that shared the split edge. The two triangles that shared the split edge were deleted, and then four new triangles were generated to take their place. Figure 2 is an annotated screenshot demonstrating the technique. The blue line denotes the edge that was split, the black lines show the other edges of the original triangle, and the green lines are the new edges that were created.

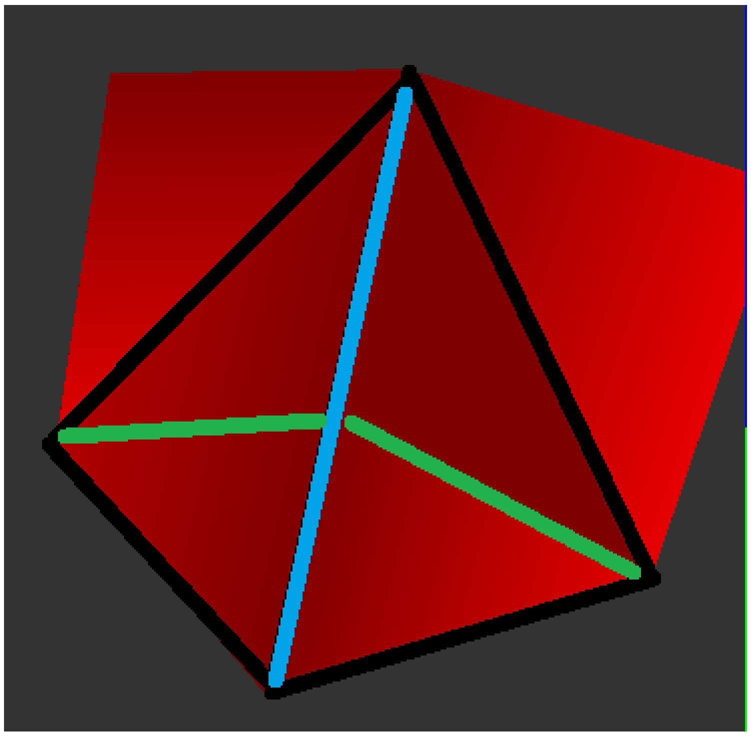


Figure 2: The 'Split Edge' technique

This method did create more complex local geometry, but the complexity was somewhat unintuitive and did not create the desirable smoothing effect. There was also no option to automatically apply the algorithm across all edges and increase the complexity of the entire object. In order to address this, I decided to replace this feature with Loop Subdivision[1], which would increase the complexity of the model by a factor of four and smooth the hard edges.

In order to accomplish this task, I first had to rewrite the mesh representation framework of the original program to adopt the half-edge data structure. After migrating my code base to the new framework and ensuring that all original functionality remained, I began the task of adding the new features.

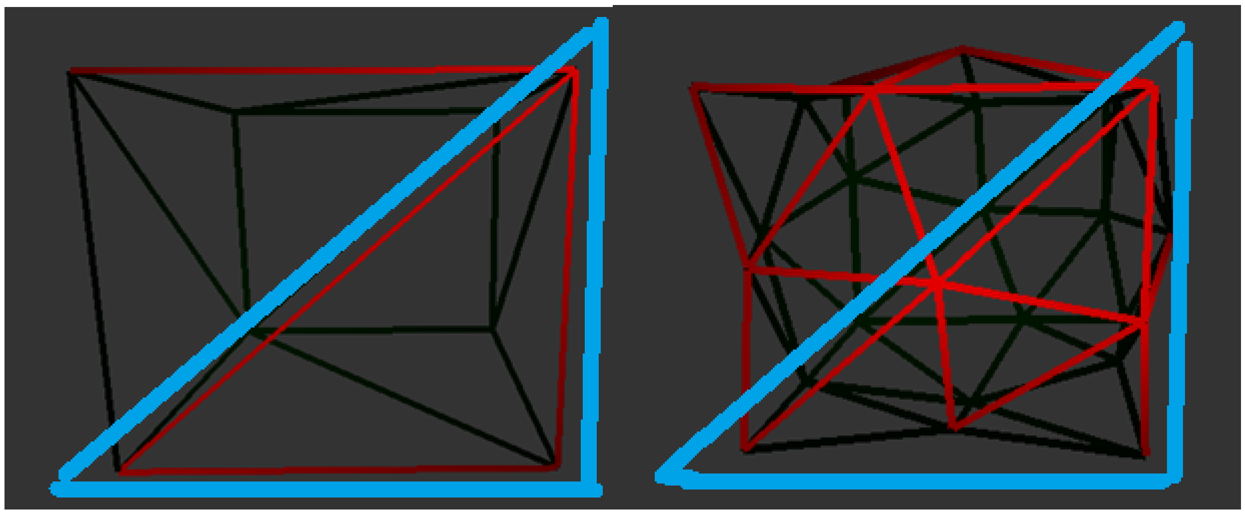


Figure 3: One iteration of Loop Subdivision without automatic crease detection

In Loop subdivision, a single triangle is split into four by adding a new vertex along each edge, adjusting the vertex positions for smoothing, and creating four new triangles using the six adjusted vertices. Figure 3 illustrates this transformation with a cube before and after one

[1] Charles Loop: *Smooth Subdivision Surfaces Based on Triangles,* M.S. Mathematics thesis, University of Utah, 1987. http://research.microsoft.com/en-us/um/people/cloop/thesis.pdf. (Last accessed May 1, 2013)

iteration of Loop Subdivision. After multiple iterations of Loop, the object becomes very smooth with $n*4^i$ faces, with n being the initial number of faces and i the number of iterations.

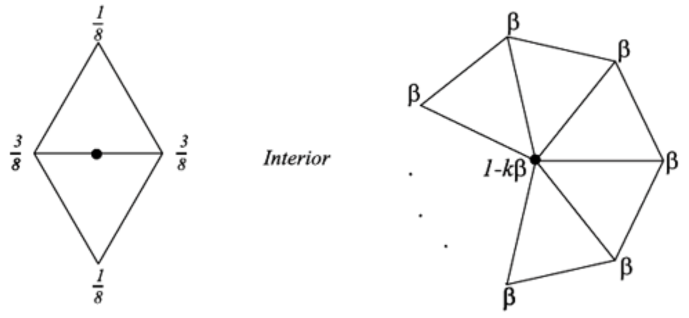


Figure 4: Normal vertex subdivision masks

The positions of the adjusted vertices are determined by the subdivision masks in Figure 4, which define different rules depending on whether a vertex is original or newly created. Various implementations of Loop calculate β differently, but for my program I chose

$$\beta = (5/8 - (3 + 2*\cos(2*PI*k))^2/64)/k,$$

where k represents the number of adjacent original vertices. Below is a side-by-side comparison of the same surface after five iterations of subdivision.

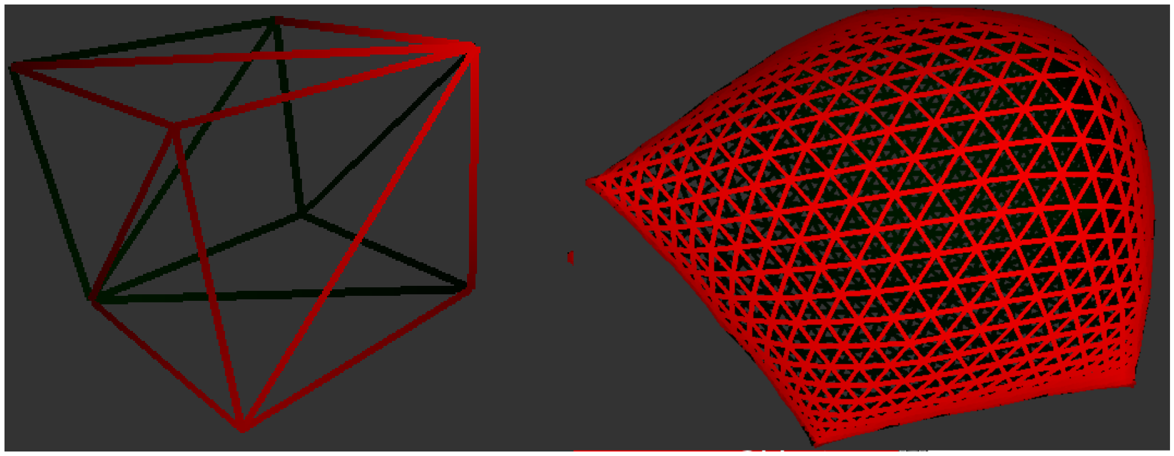


Figure 5: Before and after five iterations of subdivision

In order to give the user more control, I also implemented an algorithm which will auto-detect crease edges, which receive different weights than normal weights. The algorithm iterates over all edges in the model and measures the angle between the two attached triangles. If that angle is greater than 75º or less than 105 º, that edge is marked as a crease edge. In order to quickly apply subdivision masks, the Vertex class was updated to store pointers to other vertices with which it shares a creased edge. Users have the option to turn crease detection on and off. Figure 6 shows the same starting object after three iterations of subdivision with creases on and with creases off.

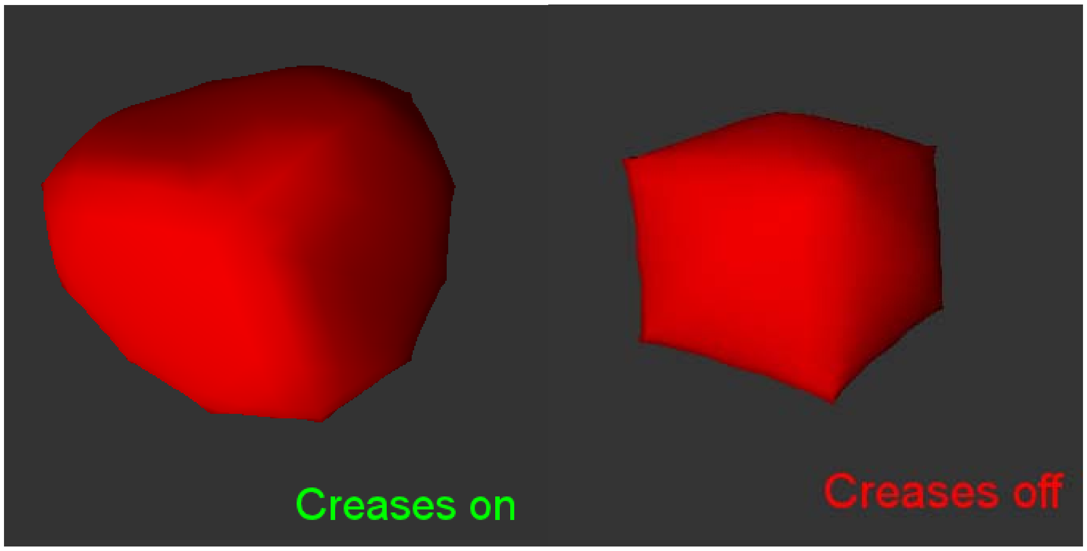


Figure 6: Three iterations with creases on and off

Loop Subdivision is an excellent algorithm to add complexity to the entire model, but I also wanted to give the user the option to subdivide locally. However, Loop's method cannot be applied locally because it does not preserve the half-edge structure requirement that each edge is opposite only one other edge. With this in mind, I implemented a local subdivision algorithm that splits a given triangle into three new triangles by creating one new vertex in the center of the triangle. This method does not produce a smoothing effect, but it does allow the user more control over the mesh. The figure below shows the differences between Loop Subdivision and localized subdivision.

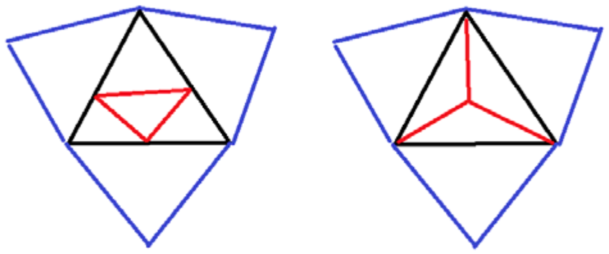


Figure 7: Loop Subdivision (Left) and Local Subdivision (Right)

Simplification

Along with the ability to increase the complexity of the object, it is also important to be able to simply an object. For interactive applications like games, it is best for models to have a reasonable number of polygons to achieve a fast frame rate. The goal of simplification is to reduce the number of triangles without losing important details. In order to achieve this, I implemented both a localized edge collapse, which allows users absolute control over the simplification process, and a general purpose triple-decimation algorithm that automatically reduces the number of triangles by thirty percent.

Both algorithms are based around the idea of a half-edge collapse[2], where given a directed edge AB, the two triangles connected by AB are removed, and the triangles sharing including vertex B are migrated to vertex A. Once this is done, the vertex B is deleted.

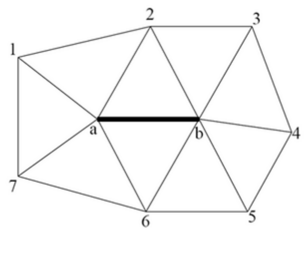


Figure 8: Half-Edge Collapse

[2] Chen, Falai and Juttler, Bert. *Advances in Geometric Modeling and Processing:* 5th International Conference, 2008.

This process can fail in some cases, so before attempting a half-edge collapse, it is important to ensure that the intersection of the one-rings of A and B consists of vertices opposite the edge AB or BA only[3]. Because the models handled by this program do not include open-geometry models with boundary vertices, once this condition is satisfied, simplification can proceed safely.

Edges with shorter lengths are prioritized over longer edges, and edges that do not meet the safety condition are marked as illegal and ignored for the rest of the iteration. Figure 9 demonstrates multiple passes of the triple decimation algorithm with images of both the shaded object and its wireframe.

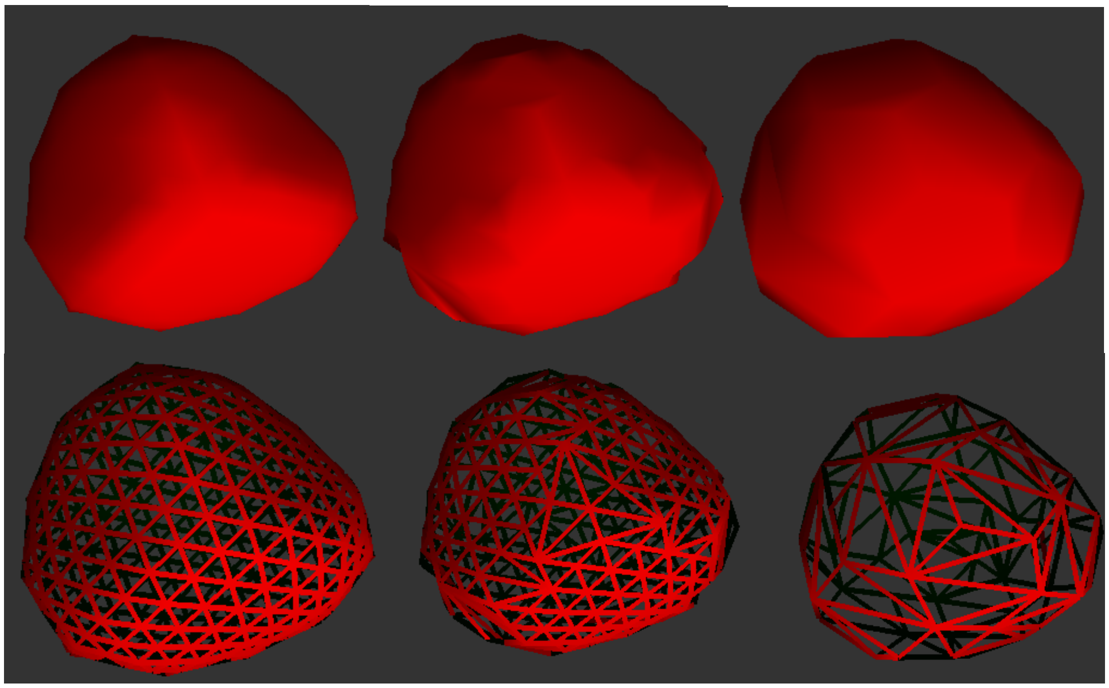


Figure 9: Object in stages of triple-decimation – 768 to 536 to 126 triangles

When the user wants to preserve detail in specified areas, he can either collapse edges manually one by one, or he can tag edges that will be preserved. Tagged edges do not persist

[3] http://books.google.com/books?id=8zX-2VRqBAkC&pg=PA118&lpg=PA118&dq=half+edge+collapse+illegal&source=bl&ots=ZZ6SXmzT1r&sig=p5lb8XNlqmDNX92Cl6_kHHg6xN0&hl=en&sa=X&ei=mKx-Ub6HLavl4APZ8IDoCw&ved=0CDEQ6AEwAA#v=snippet&q=half-edge&f=false

after a subdivide, and they can be reset in some cases of triple-decimation, so after major geometry changes it is necessary for the user to re-tag edges. Tagged edges are displayed in yellow.

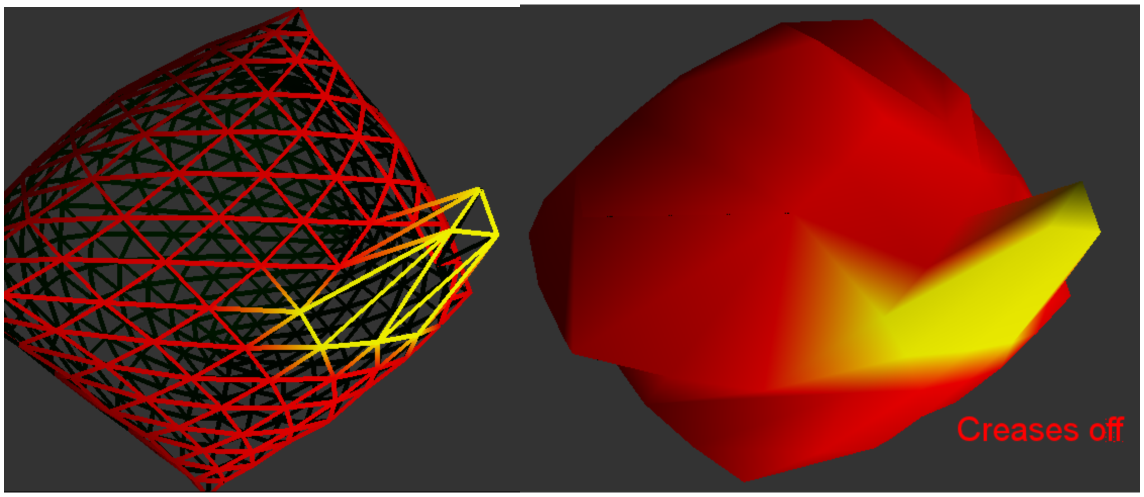


Figure 10: Triple-Decimation with tagged edges

Optimization

Because this is an interactive program, the optimization of the aforementioned algorithms was a critically important part of this project. Using SFML's included clock utility, I timed how long different parts of the algorithms took, and then focused on the parts that were consuming substantially more time than others.

After running diagnostics on Subdivision, I noticed that adding and removing triangles was by far the least efficient part of the algorithm, whereas the time for creating and adjusting vertices was negligible. In the original method, for every face, the original triangle is removed and four new triangles are created to take its place. After some investigation, I discovered the removing all triangles before creating any new triangles made the algorithm more efficient. I further improved performance by creating a new function that cleared all triangles without finding them individually. In general, the optimized subdivision halves the processing time of original subdivision.

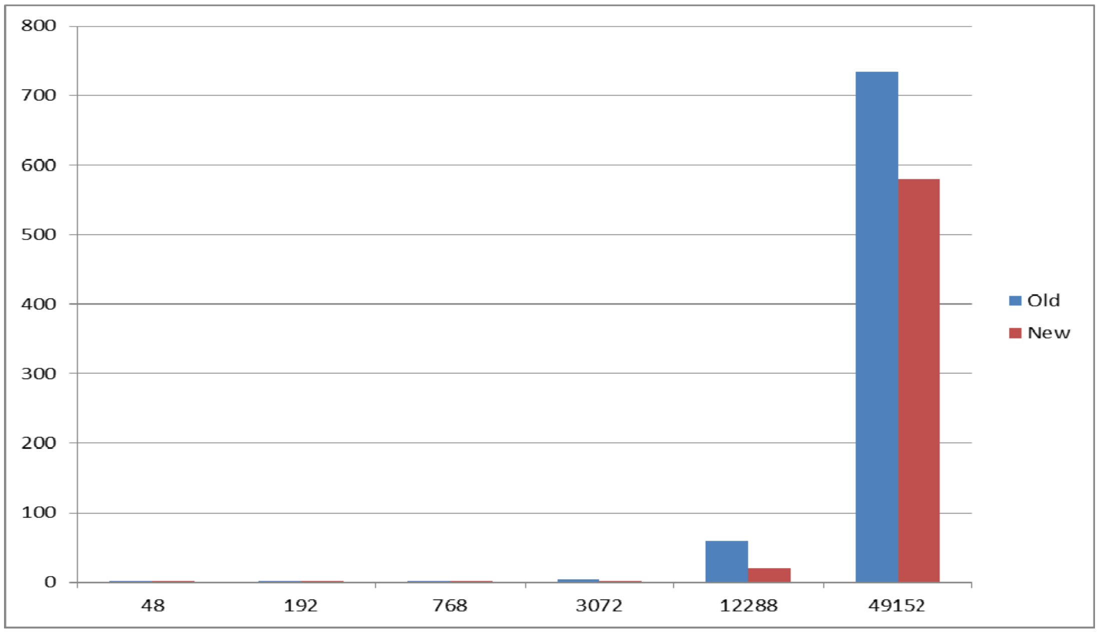


Figure 11: Subdivision Original vs Optimized Performance

When working with over five thousand polygons, even the optimized program's subdivision times are too slow for interactivity, however the bottleneck is freeing up memory, which cannot be glossed over. A potential solution to this problem would be to use a thread to clean up the memory while the main program adopted a new container for edges and triangles.

After optimizing triangle deletion and creation, I moved on to the simplification algorithm. Originally, for each iteration of triple decimation, the algorithm would run through all edges and find the shortest legal edge to collapse, making the overall process take $O(n^2)$ time. To expedite this, I first tried to implement a priority queue of Edge pointers that would automatically push the best candidate edge to the front. The construction of this structure went fairly smoothly, but I quickly learned that priority queues and heap structures in general are not inclined toward structures whose prioritization status changes often. After playing around with priority queue for awhile, I decided to opt to use a set instead. The hash structure ensured that the best candidate would still be pushed to the front, and when edge collapse changed the length of an edge, the edge was removed, edited, and re-inserted in O(log(n)) time.

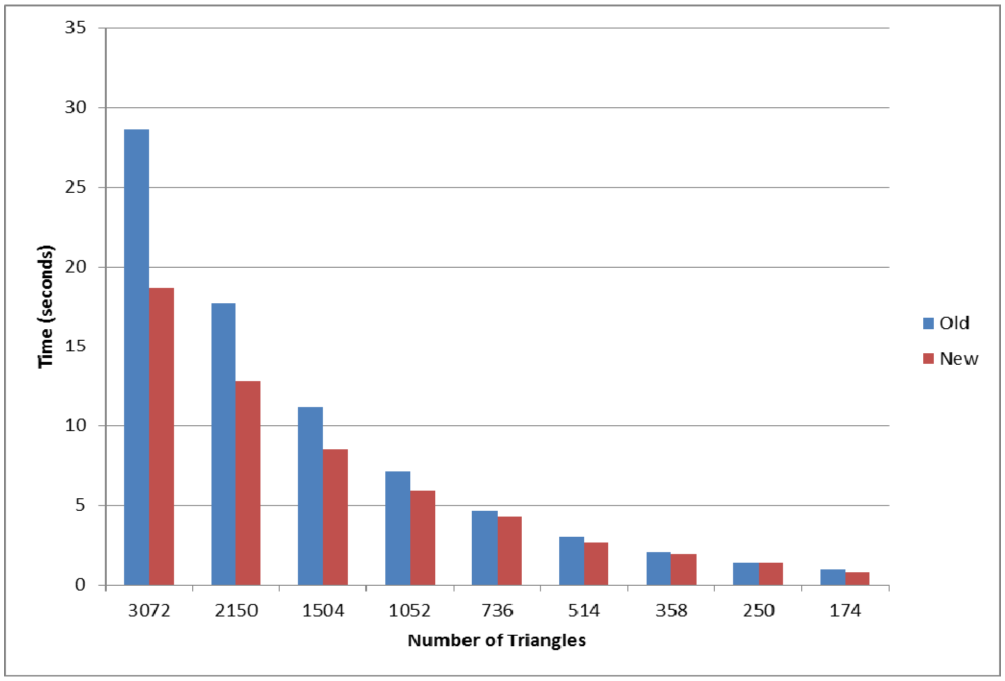


Figure 12: Decimation Optimized vs Original Performance

While working on the project, I also took measures to optimize the user interface. When users are selecting objects, they have to click directly on what they want to select, which was sometimes difficult when viewing the wireframe of the object in Vertex and Edge modes. To remedy this, I changed my OpenGL calls to draw big dots to represent vertices in Vertex mode and increased with width of the lines in the wireframe while in Edge mode. These modifications made it much easier to select and modify objects. I also changed the back face color of the models and added Gouraud shading to improve the look of the models.

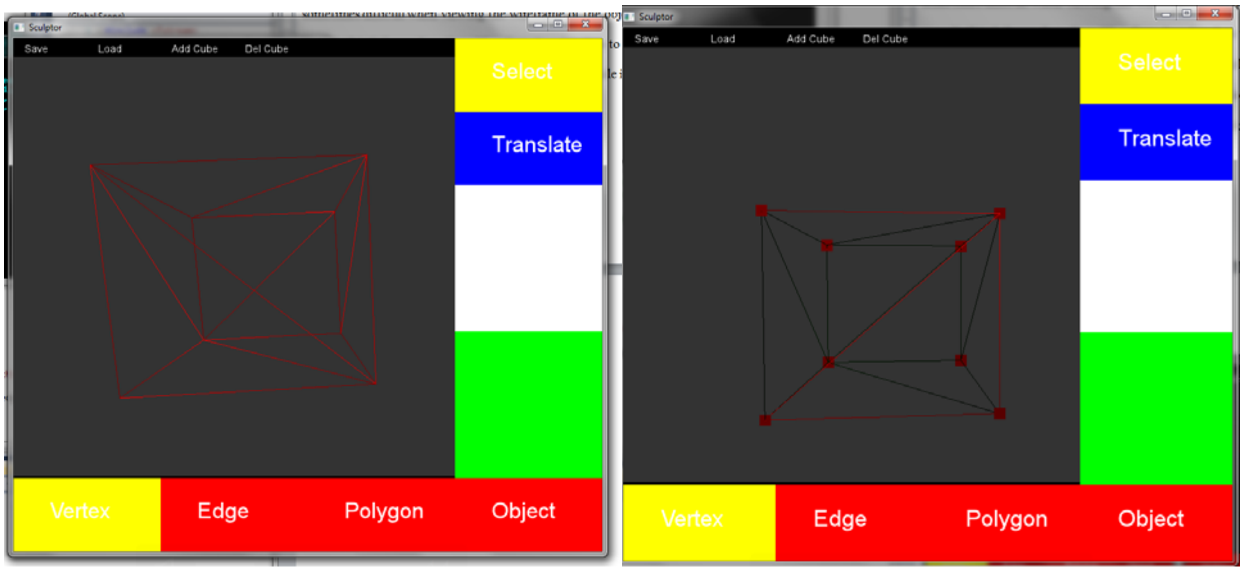
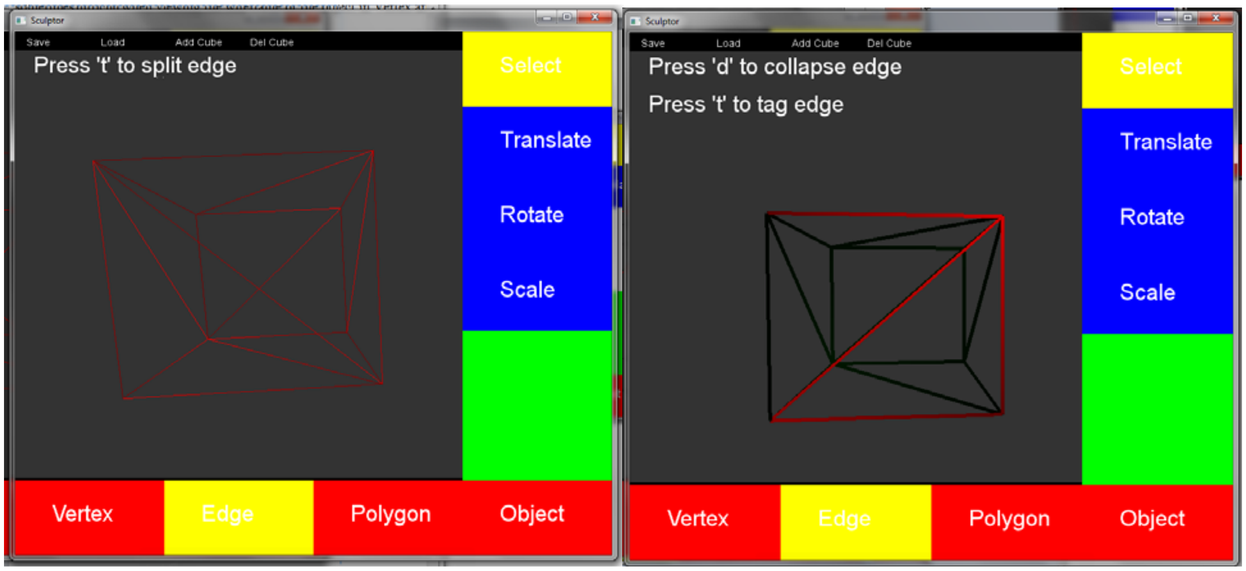


Figure 13: Original Vertex Mode and New Vertex Mode


Press 't' to split edge

Press 'd' to collapse edge

Press 't' to tag edge

Figure 14: Original Edge Mode and New Edge Mode

Pitfalls

While working on this project, I experienced various problems, most of which I was able to successfully debug and address but some of which still persist in the final program.

The first hurdle was actually SFML 2.0, the windowing system that I also used to construct the GUI. I choose SFML 2.0 over GLFW last year because I had used SFML 1.6 in the past to make games so I was somewhat familiar with its capabilities. SFML 2.0 supports OpenGL, but it was [and still is] in Beta. After various headache-inducing sessions of booting up my program and not being able to see anything onscreen, I realized that SFML 2.0 does not take kindly to the use of VBOs. After a few hours of trying to negotiate with the program, I resorted to using the old way of glBegin to draw my triangles, and that method persists even in the final version. This means that the frame rate of my program begins to drop when rendering models with over ten thousand polygons.

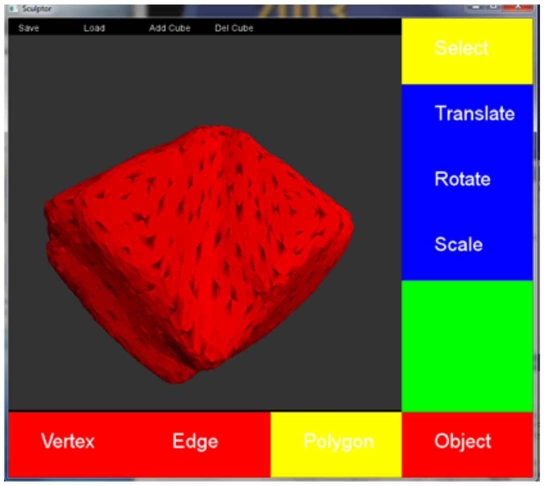


Figure 15: Gouraud Glitch

One of the most annoying problems I encountered was also linked to the glBegin way of drawing triangles. I decided early on to implement Gouraud shading to enhance the appearance of my models, but something looked off about them until the end (see Figure 15). To debug this problem, I implemented a function that would draw the normal of the vertices, which all looked correct. Finally, after stumbling around the OpenGL documentation for some time, I realized that I was drawing the vertex before declaring its normal, so vertices were receiving the normal for the previously drawn vertex. Once I fixed this, my program began to generate the beautifully shaded models present throughout the other figures in this paper.

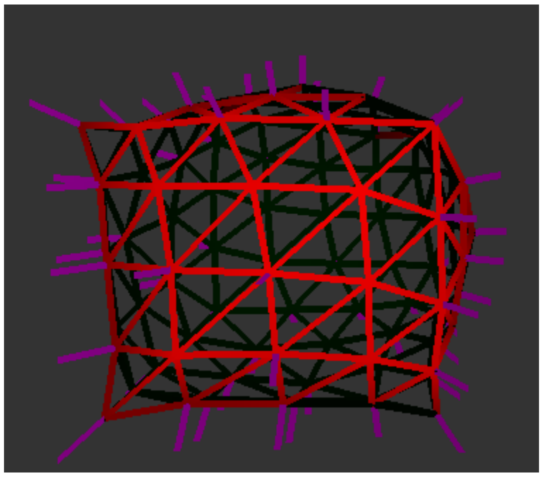


Figure 16: Visualization of Normals

Another problem I struggled with was the exact implementation of subdivision masks. While the mask itself is simple enough, at first I was unsure of whether to adjust vertices based on the original positions of their neighbors or by their adjusted positions. I also spent some time debugging the placement of newly created vertices. I wavered between whether or not to auto-detect creases until finally settling upon allowing the user to decide. My experimentation with subdivision masks produced a few interesting models.

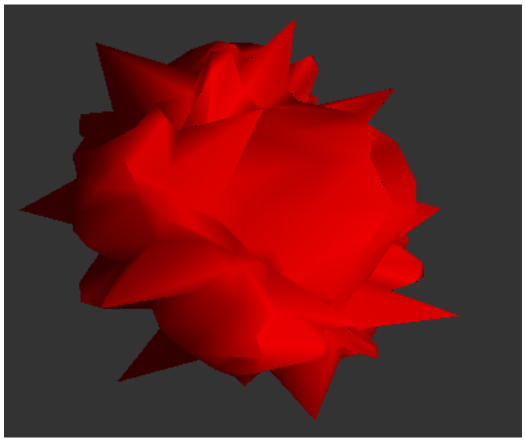


Figure 17: The Flower

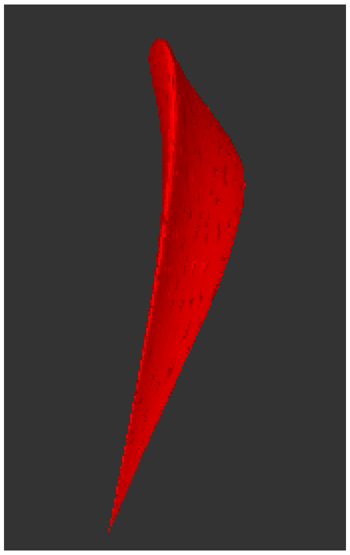


Figure 18: The Tongue

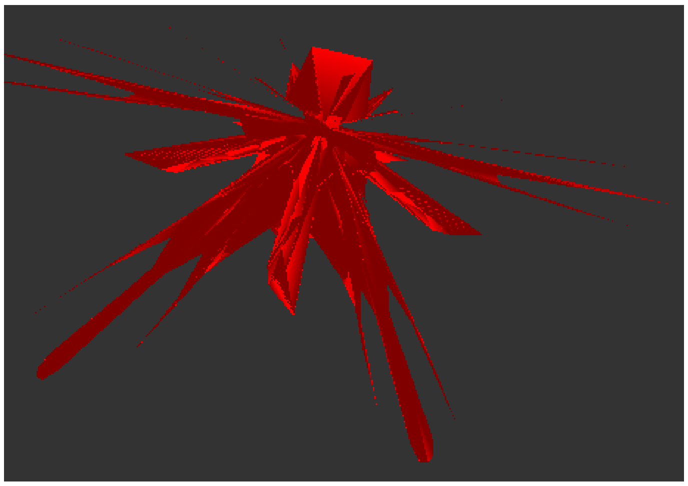


Figure 19: The Blair Witch

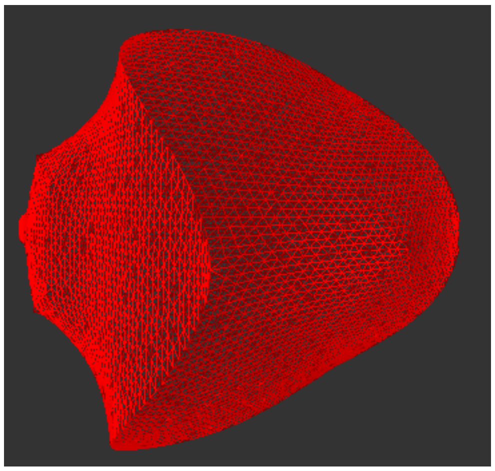


Figure 20: The Saddle

I also experienced a problem with simplification where my algorithm would run suspiciously fast and the model would reach a limit and declare that all remaining edges were illegal. Multiple calls of decimation routinely reset illegal edges, but even then the limit would remain. I discovered that the problem was caused by the function that I used to count one-ring neighbors, which wouldn't clear a vertex's list of adjacent edges before adding new edges the list. This resulted in duplicates in the list, which would produce more than two shared triangles in the safety-check operation. When I fixed this, triple decimation worked as expected.

While I was working on optimizing subdivision, I found out that the operation that took the most time was simply clearing the vector of Edge pointers, and I am still not completely sure why this is the case. For both subdivision and simplification, increasing the efficiency of memory cleanup would have probably drastically increased the performance of the program, but I was unable to determine what was making cleanup so slow in the first place.

Results

The final program is robust and the optimizations increase its performance by 25%-50%. Users can now increase and decrease the complexity of their models and still apply basic transformations to all vertices, edges, polygons, and objects. The program performance begins to decline with models over ten thousand triangles. While the program does accept open models, subdivision and decimation will fail with these, so I did not provide any open model examples to run.

Conclusion

Despite the sometimes less-than interactive operation times, the extensions to the original program have increased its functionality and usability. The goals of this project were satisfied, although future work towards further optimization would be beneficial to the program as a whole.

Over the course of this project, the workload totaled about 30 hours. Converting the project to the half-edge data structure took four hours, basic implementation of subdivision and decimation was another six hours, and the rest of the time was spent on debugging and experimentation.

References

Charles Loop: *Smooth Subdivision Surfaces Based on Triangles,* M.S. Mathematics thesis, University of Utah, 1987. http://research.microsoft.com/en-us/um/people/cloop/thesis.pdf. (Last accessed May 1, 2013).

Chen, Falai and Juttler, Bert. *Advances in Geometric Modeling and Processing:* 5th International Conference, 2008.

Botsch et. Al. *Polygon Mesh Processing*. A K Peters/CRC Press, 2010. http://books.google.com/books?id=8zX-2VR. (Last accessed May 2, 2013).