1. Mesh Coloring

a.) Assign unique color to each polygon based on the polygon id.

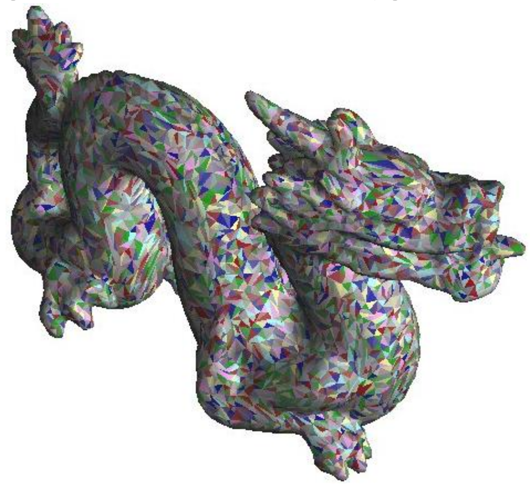

Figure 1: The dragon model is shown rendered using a coloring scheme based on coloring each triangle face according to its id.

b.) Assign color to each polygon based on its normal (R = $|N_x|$, G = $|N_y|$, B = $|N_z|$)

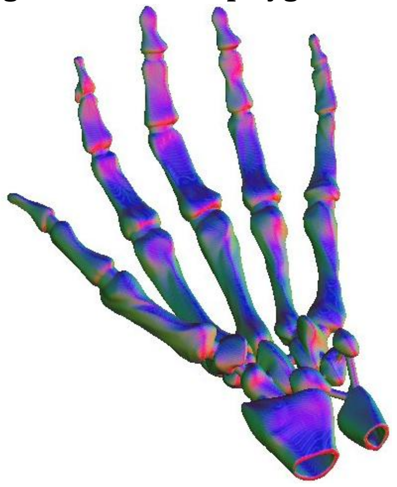

Figure 2: A skeletal hand whose color is determined according to the face normal, where red denotes the x-axis, green denotes the y-axis, and blue denotes the z-axis.

c.) Assign a color to each vertex based on its 3D coordinates according to the following map R = f(floor(Vx/L)), G = f(floor(Vy/L)), B = f(floor(Vz/L)), where L is a user-defined positive real number and f(n) = 0 if n is odd and 1 if n is even. This will produce a 3D checkerboard pattern on the model.

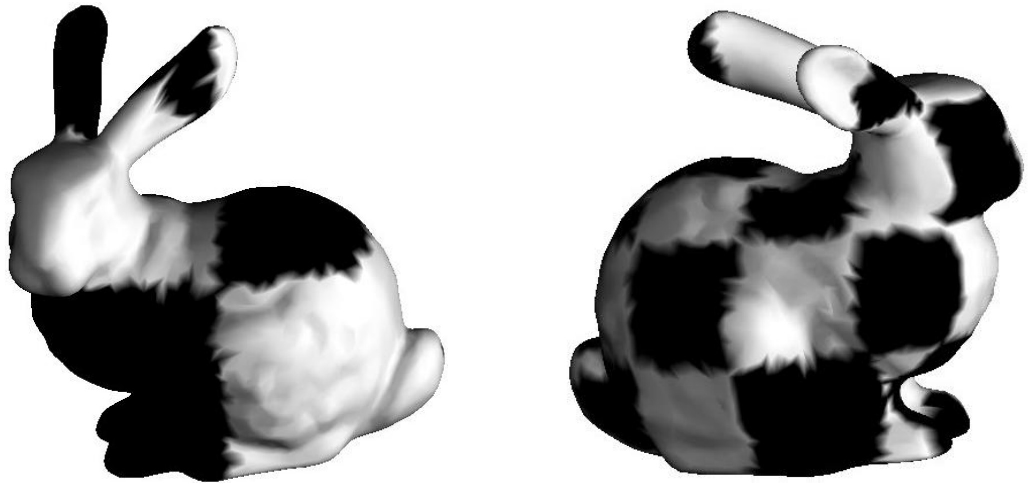

Figure 3: The Stanford bunny is textured with a 3D checkerboard pattern using an l-value of one (left) and .5 (right).

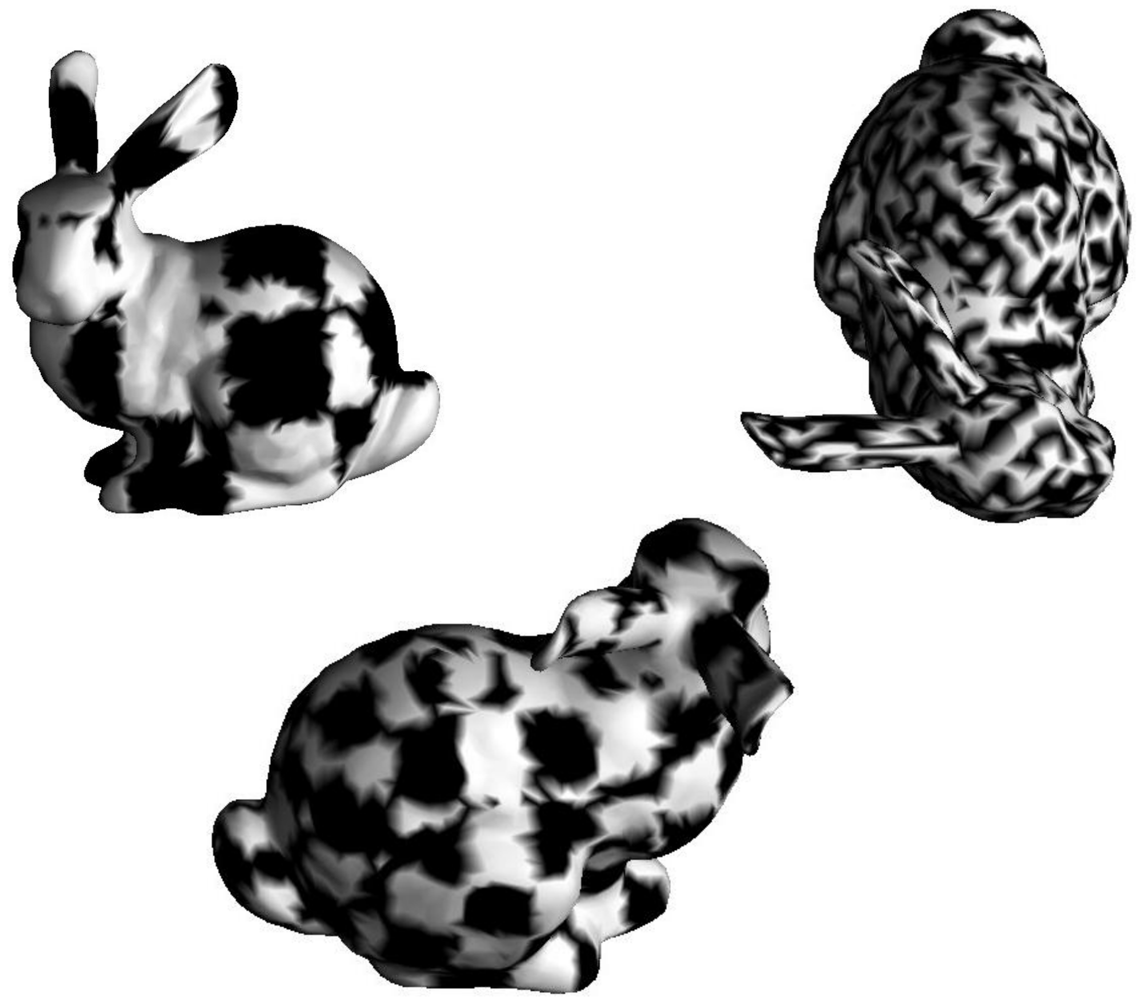

Figure 4: The progression of inconsistencies in the texturing can be seen in different views of the Stanford bunny by varying the l-values. The values used here were .5 (above left), .25 (right) and .1 (above right).

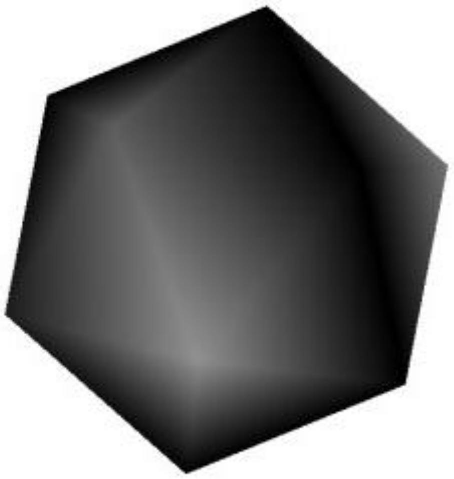

Figure 5: The 3D checkerboard algorithm applied to the icosahedron.

d.) Analysis for Problem 1

This part of the assignment helped build familiarity with OpenGL drawing commands and made it necessary to explore how to work with different types of lights and materials. There were several unique challenges that were presented in each piece described above.

In part a, the goal was to color the model in such a way that the triangles composing it are easily distinguished from one another (see Figure 1). Finding a way to randomize the colors based on id was more difficult than I originally thought it would be because the ids do not necessarily proceed in numerical order. However, this problem was solved by partitioning the ids into three sets using the modulo operator and holding one component of the color constant while using the id to compute other two (so if the id modulo three was equal to zero, the red component of that triangle would be fixed at one, but the green and blue components would both be equal to the id divided by the total number of triangles in the model).

The objective in part b was to display the local coordinate system of the model by coloring the faces of the model. In this implementation shown in Figure 2, red, green and blue indicate the direction of the x-, y-, and z-axis respectively. One problem I encountered in this part was that the model would look correct from the front, but it was completely black in the back. What had happened was that when the normal was facing away from the viewer, a negative value would be assigned to the color value for that face. Since colors can only be in [0, 1], OpenGL simply snapped the color to the closest valid value, which made it black. This meant that the positive axes were being colored correctly; however, the negative axes were being colored black. This problem was an easy one to solve—I simply took the absolute value of the colors before they were used to color the model.

By far, the 3D checkerboard (part c) was the most challenging part of problem one. I ran into a lot of problems along the way, but most of them could have been solved earlier on if I had taken more time to plan out the algorithm and understand the problem before I started trying to program the solution. In addition to that, I wrote my own floor function, which did not give the correct results in all cases. I was able to apply what I had learned in part b to fix an issue I had where the checkerboard was only being drawn correctly at the intersection of the x-, y-, and z- axes and playing around with l-values made the checkerboard look more realistic. The models shown in Figure 3 differ only in orientation and l-value used (the first model was drawn using an l-value of one while the second used an l-value of .5). They demonstrate how the l-value is directly proportional to the size of a square so the larger the l-value, the larger the squares in the checkerboard pattern will be. However, I found that this is not strictly the case. When I used l-values higher than one, the pattern did not appear to be different, and with some l-values (see Figure 4), the pattern produced did not resemble a checkerboard at all. Because of that, once I actually obtained the correct solution, I still was not convinced it was actually right because of the way the models looked. What I did not understand was that because this algorithm is attempting to map a 2D texture onto a 3D surface, the result is not always a perfect checkerboard. As can be seen in Figure 4, the texture does not always get mapped to the model in such a way that there are no seams or so the texture is even recognizable as the original once it is mapped onto the model.

Another problem that I encountered was when I tried to apply the 3D checkerboard to the icosahedron (Figure 5). In that case, the pattern did not seem to appear at all. Upon closer inspection, it looks as though the vertices are being colored correctly; however, there are just not enough vertices to make the pattern apparent.

2. Bounding Boxes

a.) Axis-aligned bounding box

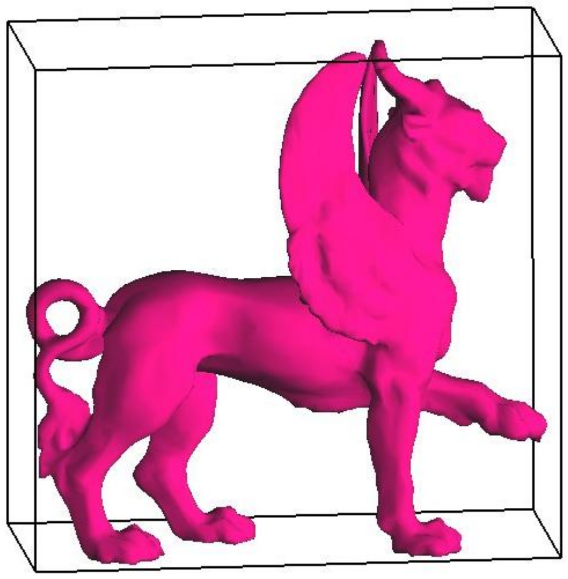

Figure 6: A bounding box based on the local axes of the feline model.

b.) Bounding box based on first moment (center of gravity) using vertex position

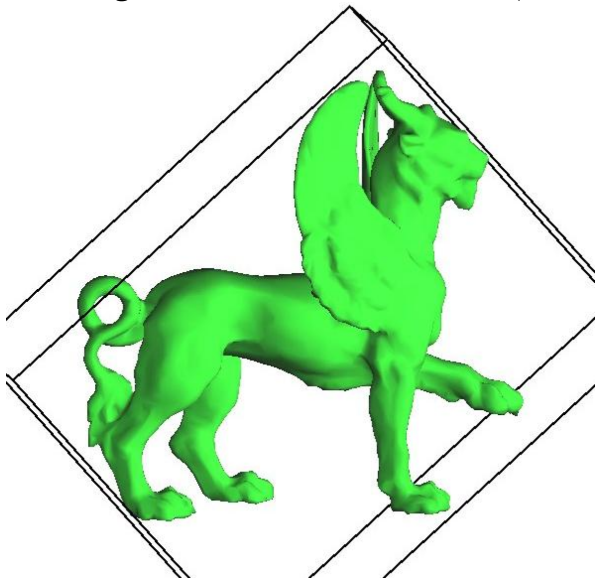

Figure 7: The bounding box calculated relative to the feline's center of gravity and the position of its vertices. The model is shown oriented in the same way that it was in Figure 5.

c.) Bounding box based on second moment (orientation) using face normals

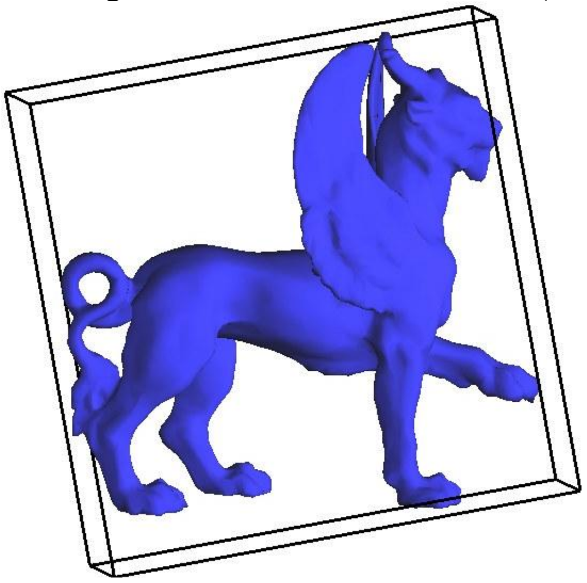

Figure 8: The feline model, oriented in the same way as the model in Figures 5 and 6, is shown with a bounding box based on the model's orientation and the surface normal of its faces.

d.) Analysis

Constructing the axis-aligned bounding box presented no problems whatsoever. The algorithm was one that I was familiar with, and it was straightforward to implement. The results can be seen in Figure 6. Likewise, once the algorithm for constructing a bounding box based on vertex position and using the first moment (Figure 7) was implemented, coding the bounding box based on surface normals using the second moment (Figure 8) was not difficult either. However, there were several significant problems that I encountered with part b of problem 2.

The first problem was the result of a fundamental misunderstanding of some of the concepts behind how to construct the bounding box. I started writing code before I understood the whole algorithm. This ended up producing some strange behavior that was difficult to track down once I had implemented my solution. As it turned out, I had misunderstood how to find the center of the bounding box after finding the eigenvectors that formed an orthonormal basis for the figure (according to the first moment and vertex positions) and the maximum and minimum point the model extended to along those vectors. All I needed was to simply average the max and min points according to their axis and add the resulting points together. This was a simple mistake that ended up costing me a lot of time. From this, I have learned the importance of grasping the conceptual material completely and not blindly implementing formulas without first understanding what they are intended to do and whether or not they accomplish that goal.

The second major hurdle I encountered in part b was when I obtained a function from Numerical Recipes that used Jacobi transformations of a symmetric matrix to find the eigenvalues and eigenvectors for that matrix. What I did not realize when I used the function was that it was originally written for use with Fortran so it assumed one-based indexing instead of zero-based indexing and used column major ordering instead of row major ordering—column major ordering was less of a problem since I was working with a symmetric matrix. In addition to that, I transferred some of the code incorrectly and did not catch my error for some time because I did not test the function sufficiently before using it. However, with a great deal of help from Dr. Zhang, I was able to fix the problem. Even more importantly, I realized just how critical it is to fully test code—even if it is not your own. I also learned some new techniques for how to test the code and how to form useful test cases.

After examining the models, it appears that the smallest eigenvalue corresponds to the shortest bounding box dimension and the largest eigenvalue corresponds to the longest bounding box dimension.

In considering a hierarchical model for collision detection, the most intuitive method would involve different types of bounding boxes at different layers of the hierarchy. The most general box (surrounding the entire figure) could simply be an aabb. It does not provide the smallest bounding volume; however, it is the most computationally inexpensive method. At the next level of the hierarchy, the bunny's ears, head and torso could be bounded by boxes based on vertex positions. Those boxes would give a tighter bound and would have significantly less vertices to use in the calculations. However, if a featureless model was used, I think it would be best to simply use aabbs because it is the features of a model that would give its bounding box a unique orientation. Since an aabb, vertex-based bounding box and normal-based bounding box would all appear the same, it makes more sense to use the least computationally expensive method.

3. Silhouettes

a.) Silhouette constructed using edges of triangles

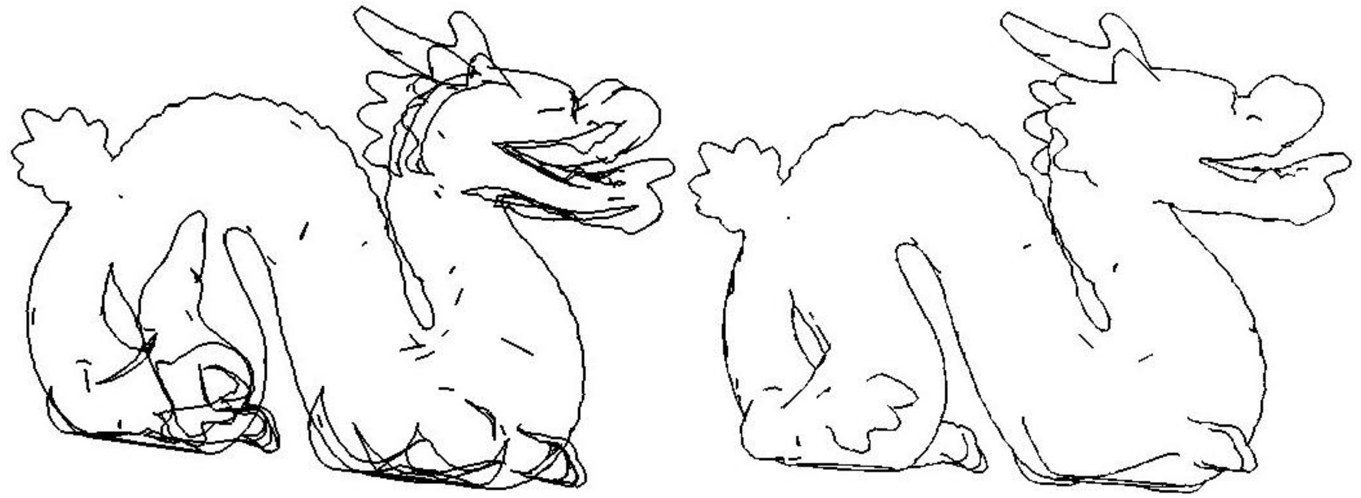

Figure 9: One of the problems encountered with computing the silhouettes using triangle edges was how to display the edges in such a way that the ones behind the model would not appear. (left) shows the dragon model with the back edges showing through and (right) shows what the final product looks like once that problem was corrected.

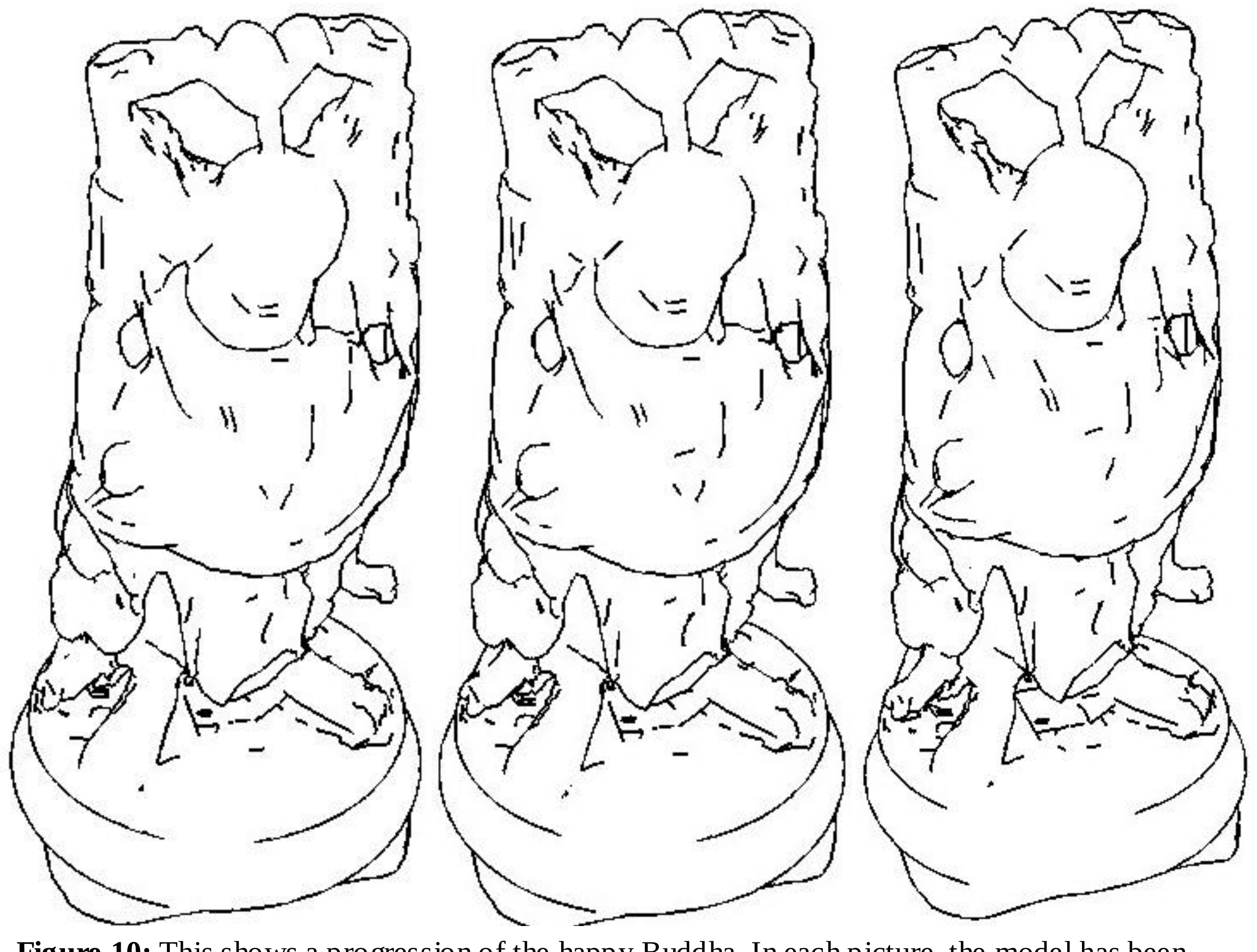

Figure 10: This shows a progression of the happy Buddha. In each picture, the model has been rotated slightly to demonstrate how even a small rotation can have a huge impact on the resulting image (the jump in edges can be seen especially well in the podium the Buddha is standing on and in the detail on the Buddha's chest).

b.) Silhouette extracted from within faces

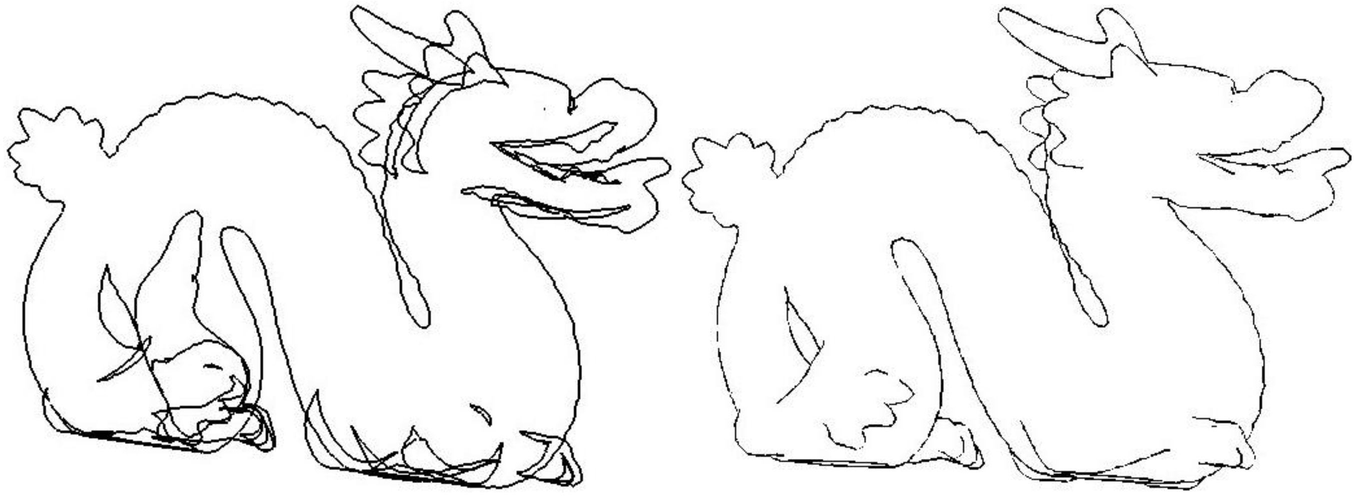

Figure 11: The same technique from Figure 9 was used here to test if the silhouette edges were really as discontinuous as they appeared.

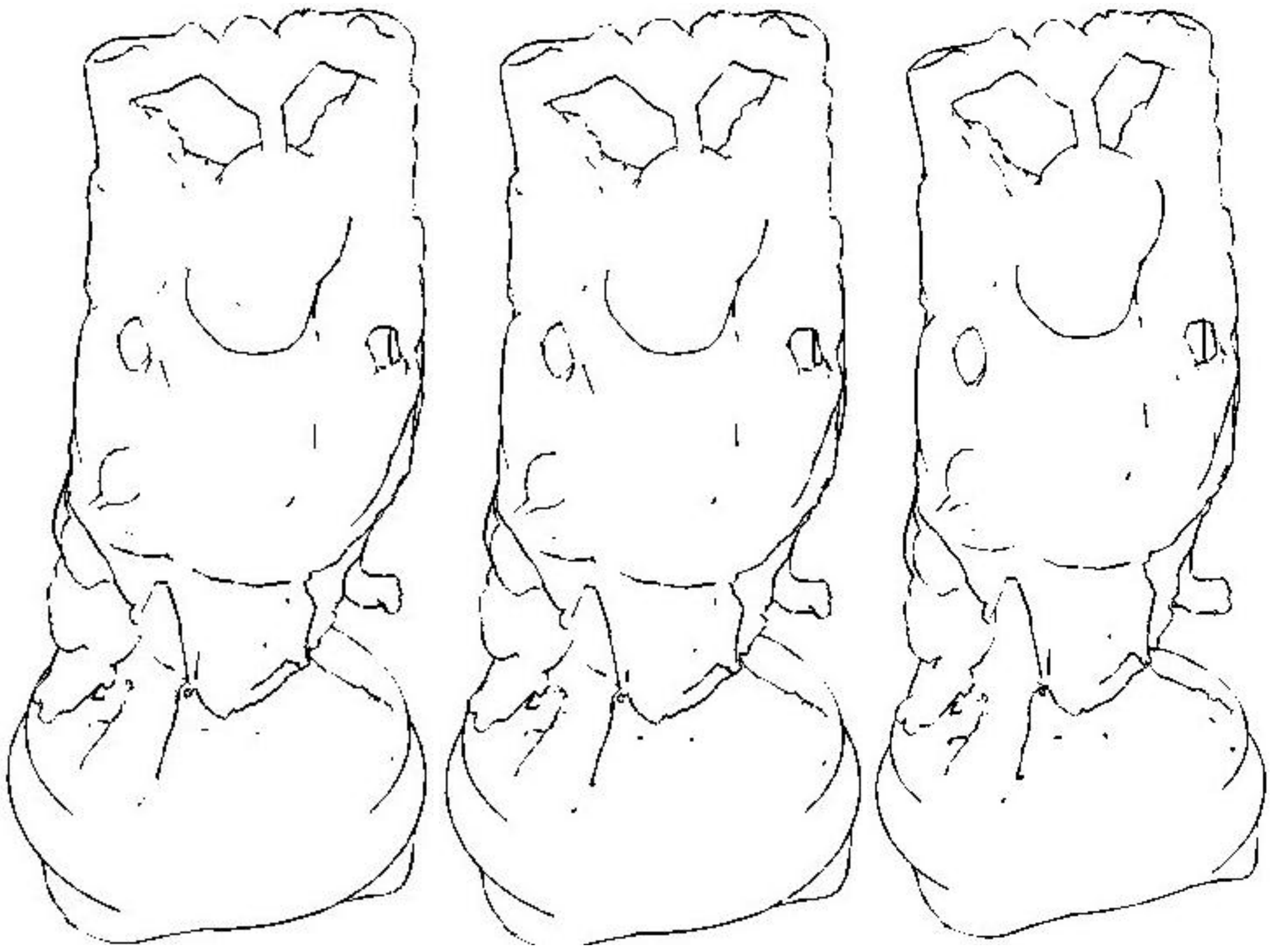

Figure 12: This shows the same progression of the happy Buddha that is shown in Figure 10; however, this time it was rendered using the face silhouette technique. Here, the difference between the images is much more subtle (no rapid jumps in lines). Notice, too, that the rendered models appear much smoother and more consistent than they do in Figure 10.

c.) Analysis

This part of the project presented the most challenges in this assignment. Part a was fairly straight-forward and the algorithm was easy to grasp. The only speed bump was figuring out how to draw the model so that it would hide the silhouette lines that should not be shown—those that would have been hidden by the front part of the model—while making the final image look as though only the silhouette lines, not the model, were being drawn (see Figure 9). This required an understanding of lighting and materials in OpenGL, but it turned out to be simpler than I had anticipated. The image of the dragon in part a shows what the program produced when the model was not draw while the image of the happy Buddha shows the final result from part a.

The real trouble came in part b. It took me longer to actually understand how all the pieces of the algorithm needed to fit together. I got confused about what space I was working in, and so I ended up working in two different spaces in the main two functions without realizing it. By talking through the algorithm with Dr. Zhang, I was able to figure out that the problem was not coming from a problem with my algorithm, but that it actually was essentially a communication problem between the two functions. It had not occurred to me previously that a problem could exist between functions, not just within functions so that was definitely an important lesson to learn.

One result that I was not expecting was that the silhouette in part b would still have gaps in the lines (this can be seen in the image of the happy Buddha). In order to see why this was happening, I stopped drawing the model (see Figure 11), which showed me that the silhouette was being computed and drawn correctly. What was happening was that when the model was being drawn, it was obscuring some pieces of the silhouette lines, making the silhouette appear discontinuous. There is a function in OpenGL that attempts to solve this problem so I used that, and I tried making my lines thicker; however, I could not get the image to look any more connected than the one included here.

There are several advantages of being able to draw silhouette edges within faces instead of simply using triangle edges (see Figures 10 and 12 for examples). The silhouette produced is much smoother and there are no rapid jumps in edges when the model is rotated slightly. Because of this, drawing the silhouettes within faces works well for organic models where there are no completely straight lines whereas rendering the silhouette of something like a Platonic solid would be faster using the edge method. A last difference between the two methods is that drawing silhouette edges within faces means the resulting silhouette will include slightly less detail than the silhouette produced from the edges of triangles. This difference is noticeable in Figures 10 and 12.

Final Remarks

Overall, this assignment has taught me valuable lessons about how to debug the code I have written, how to write code that is easier to debug and how to avoid bugs in the first place. As a result of completing this project, I now have a better understanding of OpenGL and of some of the fundamental graphics concepts I will need in order to be successful this summer. In addition, this assignment has helped me better understand the need to plan out an algorithm before implementing it and how to ask for help when I am having difficulty grasping a concept.