Geometric Modeling: Project 2 Report

Dani Bell

0. Corner Lists

Construct a corner list for a 3D model.

In order to be able to start this assignment, I had to construct a corner table. Essentially, corners are a way to relate vertices, edges and triangles, and they can be used to obtain a meaningful ordering on the data that composes a model. Conceptually, corners and corner tables are fairly simple and straightforward; however, when I actually set up the data structures, I ran into a bit of trouble with memory deallocation.

Each corner contains pointers to three other corners and a pointer to a vertex, an edge and a triangle. In addition to that, each vertex must contain a list of pointers to the corners it is a part of, and vertices, edges and triangles all contain lists of pointers to each other—it provides a fast, straight-forward way to find out which vertices make up a particular triangle, which triangles an edge is a part of, etc. With all of the indirection among the data structures, I had a difficult time figuring out how to clean up the memory from the corner table without accidentally trying to delete something twice or leaving something behind. Thankfully, with some help from Dr. Zhang, Jonathan and the pre-existing code, I was able to properly deallocate the memory for the corner table. I just needed to change my destructor so that it would only delete the list of corners and not attempt to delete the data members within each corner.

1. Smoothing

Implement four weighting schemes for surface smoothing: uniform, cord, mean curvature flow (MCF) and mean value coordinates (MVC). In addition, be able to use these weighting schemes in combination with any of the following three update schemes: explicit, implicit, Gauss-Seidel.

There are three ways to update the vertices when performing smoothing. The first is the explicit method, given by

$$T^m(v_i) = T^{m-1}(v_i) + \lambda\Delta t \sum_{j\in N(i)} w_{ij}\left(T^{m-1}(v_j) - T^{m-1}(v_i)\right)$$

where

$$w_{ij} = {e_{ij}}\Big/{\sum_{j\in N(i)} e_{ij}}$$

is the particular weighting scheme being used. With the explicit method, each vertex location is computed using the previous vertex locations, and it is not until after all of the vertex locations are calculated that the vertex locations are updated. Because of this, the explicit method is unstable. It requires a small step size and if the step size is too large, it can cause the vertices to actually pass each other after their locations are updated. This will make the model blow up—the new values go to infinity (see Figure 1 for an example of this).

The Gauss-Seidel method is the second update scheme, and unlike the explicit method, it is stable. The formula for the Gauss-Seidel method is as follows

$$T^m(v_i) = T^{m-1}(v_i) + \lambda\Delta t \sum_{j\in N(i)} w_{ij}\left(T^{current}(v_j) - T^{m-1}(v_i)\right)$$

As the new vertex locations are computed, the mesh is updated and those new values are used to calculate the values for other vertices. This means that a large step size may be used for smoothing and the model will not blow up as it does with the explicit method. Also unlike the explicit method, the Gauss-Seidel is an iterative method. Because the method uses both old and new values to update vertex locations, it takes multiple iterations for the changes to propagate through all of the vertices—the first vertex will have the greatest amount of change and get closer to its neighboring vertices, its neighbors

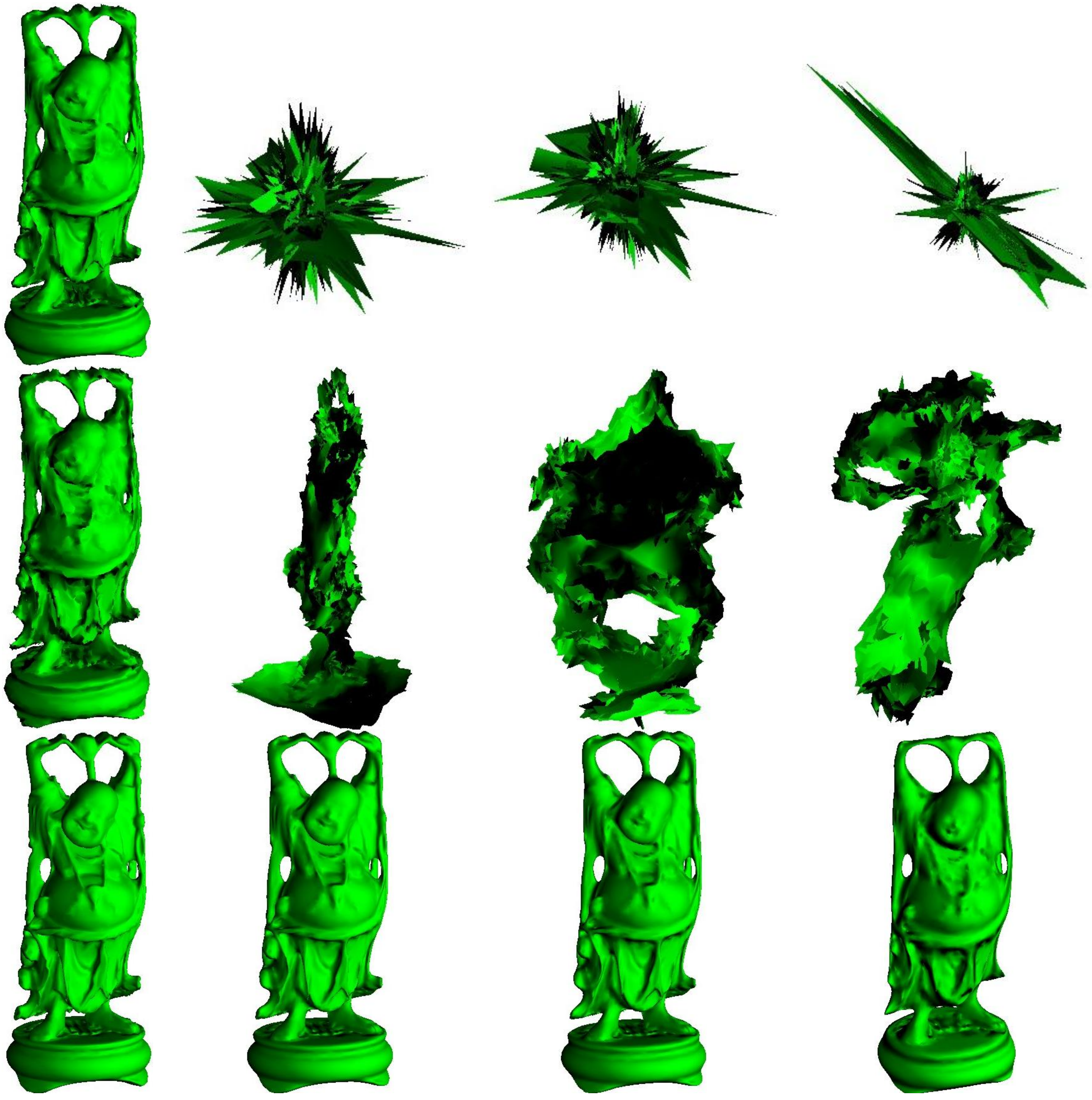


Figure 1: The top row is the happy Buddha model that has smoothing performed with the explicit update method. The middle row was smoothed using the Gauss-Seidel update scheme, and the smoothing of models on the bottom row was done using the implicit update scheme. For each image, a uniform weighting scheme was used and dt was equal to 2.0. Column one shows the results after one time step has passed. Colum two is the model after ten more iterations were run on the model from column one, and the third column has had ten more iterations run on it after column two. Finally, column four is the model after 100 iterations have been run on the model. Notice that, with the explicit scheme, the triangles have started to expand—the model is blowing up. Also, in the Gauss-Seidel method, the model still has not converged. The model resulting from the implicit method is smoothed more than what would typically be desired; however, this does demonstrate the stability of the algorithm in that the model is still recognizable as the original.

will not change as drastically since the first vertex is closer to them, etc. Also, in my experimentation, I found that sometimes it is possible to chose a step size so that the mesh never converges, an example of which can be seen in Figure 1.

The final method I implemented for this project was the implicit method, which is given by the following formula

$$T^{m}(v_i) = T^{m-1}(v_i) + \lambda\Delta t \sum_{j\in N(i)} w_{ij}\left(T^{m}(v_j) - T^{m}(v_i)\right)$$

Like the Gauss-Seidel method, the implicit method is stable so it can handle large step sizes, but it differs in that it is not iterative. It requires solving a system of linear equations to figure out where the new vertex locations should be—notice that in the sum, this is the only method that uses values from the location being computed (T^{m} instead of T^{m-1}). This means the new locations for every vertex in the mesh is solved for simultaneously, and then all of the positions are updated at once.

Because it is stable and it does not have to be run many times to get the effect to propagate through all the vertices, the implicit method is the superior updating scheme. That being said, there are a few cases where the Gauss-Seidel or the explicit method could be more useful. If the mesh did not need a great deal of smoothing so a small step size would be sufficient, it would be more worth-while to use the explicit method. For small step sizes, it is stable, and it would not require the extra time to solve the system of linear equations. Likewise, if there was one specific feature that needed to be smoothed and the rest left relatively like they were, the Gauss-Seidel method would be a good choice. By starting the smoothing with a vertex on that feature, the Gauss-Seidel method could be run a few times and stopped before the changes affected the rest of the mesh.

With each of these updating schemes, there are different types of weighting schemes that can be used. For this project, the first weighting scheme I implemented was uniform

$$e_{ij} = 1$$

This means that each neighboring vertex makes an equal contribution to how far the vertex will move. The next scheme was cord, which weights the vertices by the inverse length of the edge that connects the neighboring vertex to the vertex that will be moved:

$$e_{ij} = \frac{1}{l_{ij}}$$

The final two schemes—mean curvature flow and mean value coordinates—involved calculations that were a bit more complex. The mean curvature flow is calculated by

$$e_{ij} = \frac{(\cot\theta_1 + \cot\theta_2)}{2}$$

and mean value coordinates is given by

$$e_{ij} = \frac{\left(\tan\left({}^{\varphi_1}/_2\right) + \tan\left({}^{\varphi_2}/_2\right)\right)}{2}$$

See Figure 3 for how $\theta_1, \theta_2, \varphi_1$ and φ_2 are defined.

As can be seen in Figure 2, the weighting schemes produce similar results when a small amount of smoothing takes place. However, both mean value coordinates and mean curvature flow attempt to preserve some of the features of the models. In addition to that, the mean curvature flow weighting scheme can actually bring out defects in the mesh. This can be seen in Figure 3 where the bunny model has been smoothed with the mean curvature flow weighting scheme, and there are points on the bunny's back and face where the curve is enhanced instead of smoothed as it ought to be.

The mean curvature flow and mean value coordinates weighting schemes seem to produce better results when a large amount of smoothing needs to be done because the features of the model are better preserved. If only a small amount of smoothing needed to be done, it would be better to use either the uniform or the cord weighting scheme. They are much faster since the first two require at least two

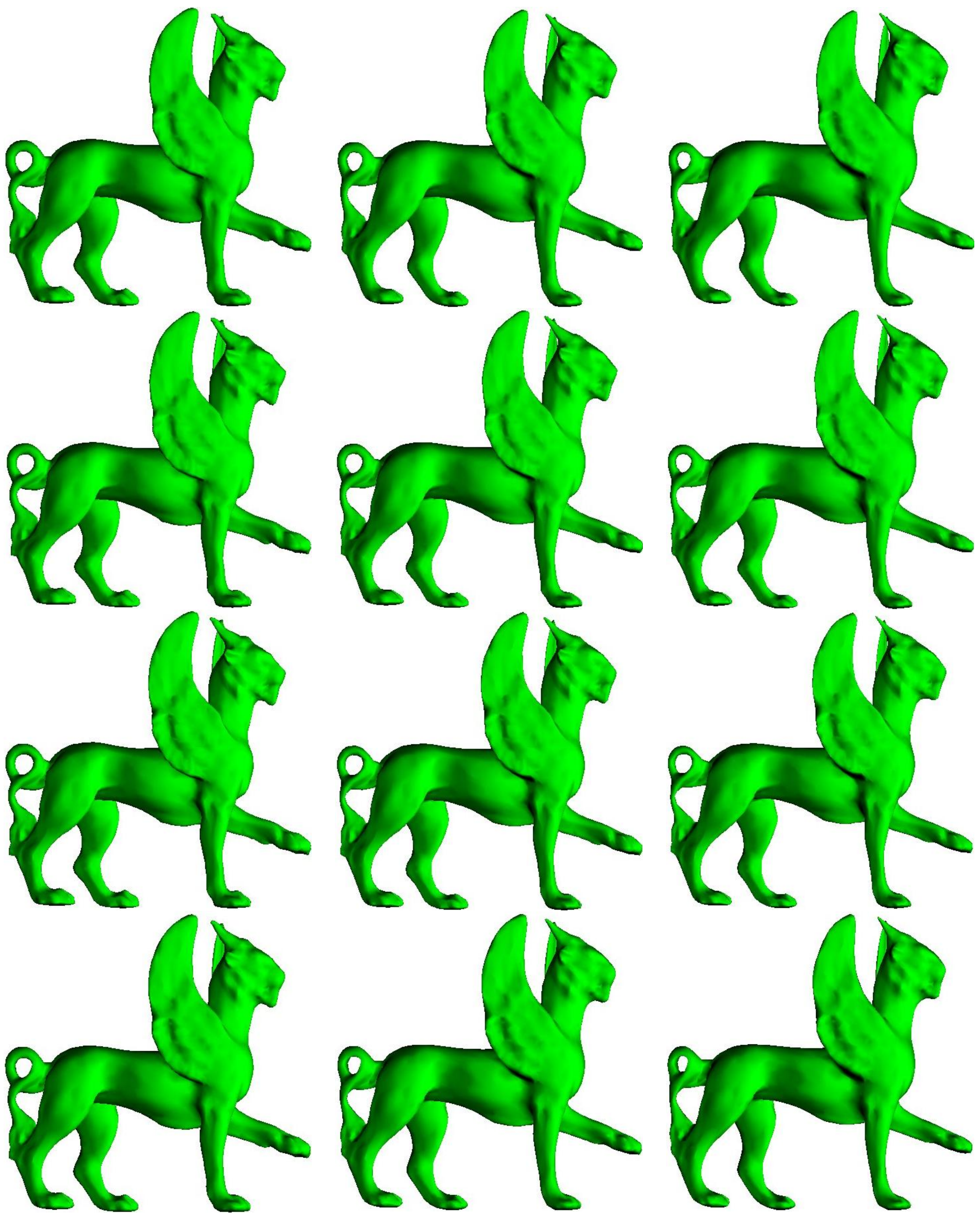


Figure 2: This figure shows the feline model smoothed according to different weighting scheme. Each model was smoothing using the implicit update scheme with a time step of 25.0. Each column shows the results of smoothing after one time step has passed—starting at one and progressing to three. Column one uses the uniform weighting scheme, column two uses cord, column three uses mean curvature flow and column four uses mean value coordinates.

trigonometry operations per neighbor just to figure out the value of one vertex. Without implementing a lookup table and introducing some error, trigonometry operations take a relatively long time to compute—especially considering the uniform weighting scheme requires one divide operation per vertex and the cord weighting scheme requires one divide operation per neighbor to determine the value of one vertex.

Because of the corner table, none of the weighting schemes were difficult to implement. Likewise, the explicit and Gauss-Seidel updating schemes were also straight-forward to translate into code. I did struggle with the implicit scheme, but once I understood what values should be put into the matrix—essentially how the formula could be broken down—implementing the implicit updating scheme was relatively simple as well.

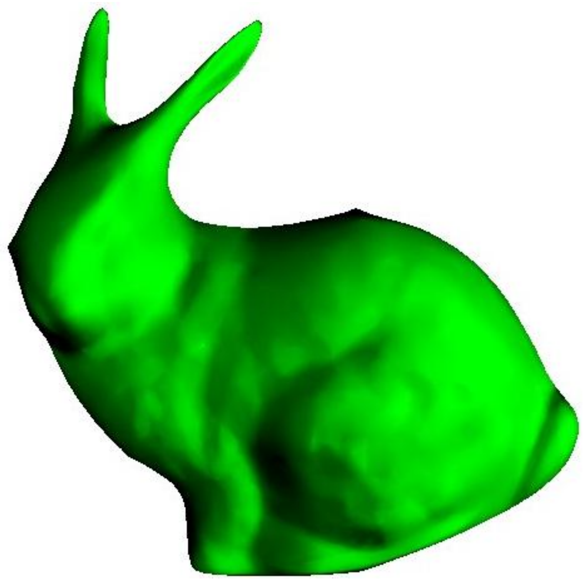
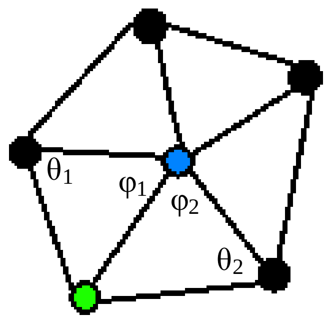


Figure 3: The Stanford bunny after smoothing has been performed using the mean curvature flow weighting scheme. Note that there are raised points on the bunny's face and back. These come about as the result of defects in the mesh (i.e. one point having four neighbors while the other points in the mesh have six neighbors).

Figure 3: When using mean value coordinates and mean curvature flow weighting schemes, the diagram to the right shows how θ and φ are defined. In this example, the blue vertex is the one whose value is being computed, and the green vertex is the current neighboring vertex whose weight is affecting the blue vertex.



2. Morse Design

Write a GUI that allows the creation of a surface function h by specifying local maxima and minima vertices and display the function.

For this portion of the project, the user can specify local maxima and minima vertices by simply right-clicking on the vertex they wish to set. Then I treated the max points as heat sources and the min points as cold sources and used heat diffusion techniques—solving Laplace's Equation—in order to compute the function (h) at each vertex. This process is similar to the one used for the implicit smoothing scheme so implementation was easy. Additionally, writing the user interface was not difficult, but it did take a considerable amount of time to make sure the functionality that a user would want was included.

The part of this that gave me the most trouble was making the colors on the model continuous, and my solution used techniques I learned from the first project (with both the normal coloring and coloring by polygon id). As seen in Figure 4, initially, I sectioned the values into seven different parts and colored the model in discrete sections based on those parts. While this technique is useful to highlight larger areas of interest, it obscures some of the details. For example, in Figure 4, it is difficult to tell exactly where the local maxes and mins are located after the model has been colored in this way. In order to get the coloring to be continuous, I sectioned the range of values—[-1, 1]—into four major parts and defined each section to be the range of values between two specific colors instead of a specific color. For example, the color starts off red (at a max point) at a value close to one. As the value changes from one to .5, the color changes from red to yellow, and as the value changes from .5 to zero, the color changes from yellow to green. This pattern continues until blue is reached for the min point. In each section, the values passed in range over a quarter of the total range of possible values—[-1, 1]—and over half of the possible range of color values—[0, 1]. In order to achieve the continuous coloration, I simply took the absolute value, shifted the range so that it fell in [0, .5] and scaled by two so that the range was then [0, 1]—the full range of possible color values. After the value has been modified as previously described, one component of the color is held fixed and the other is set equal to the value passed in. In the previous example when the color is shifted from red to yellow, the red component would be set to one while the green component would be given the modified value. The final result from this process can be seen in Figure 4.

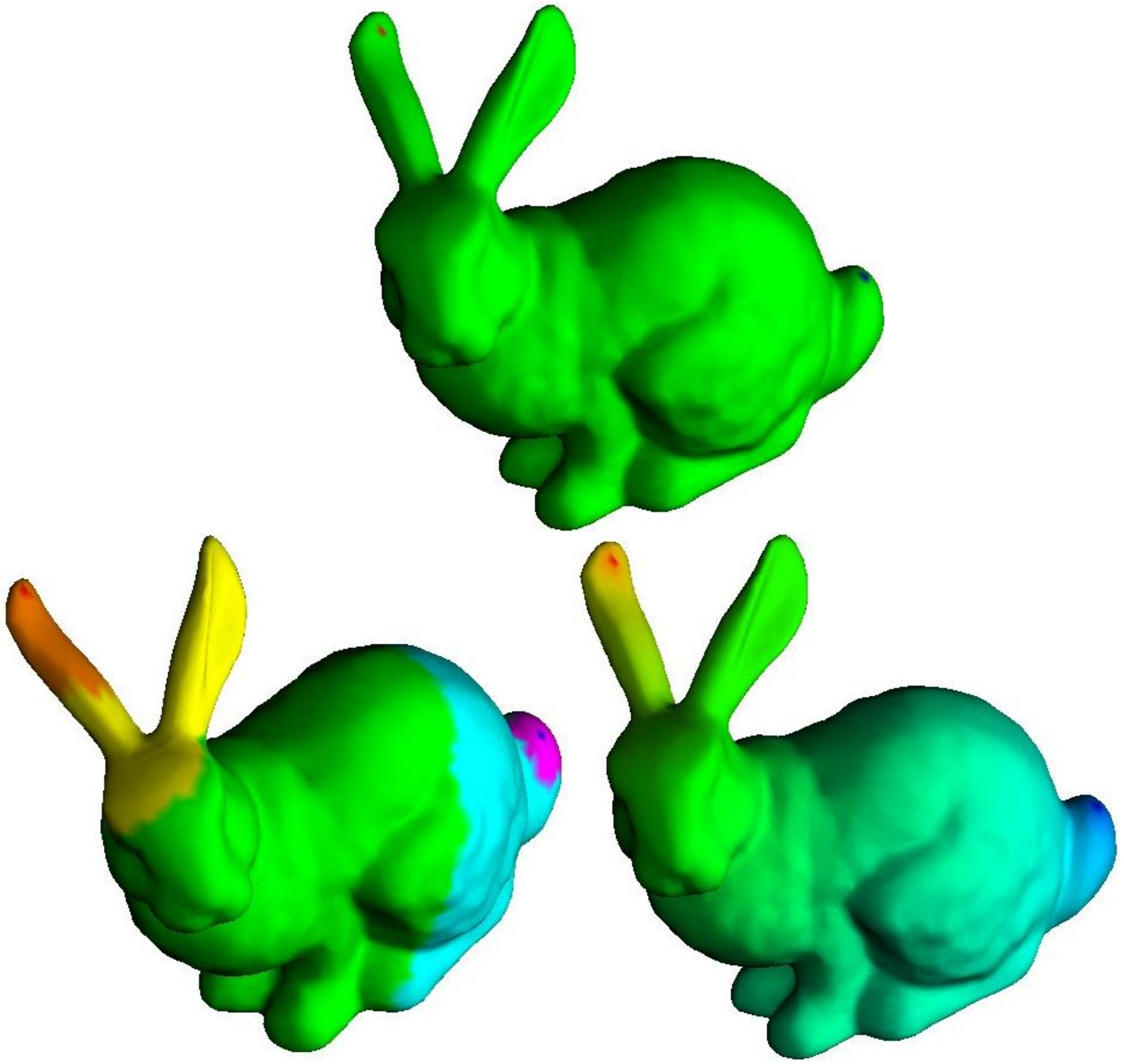
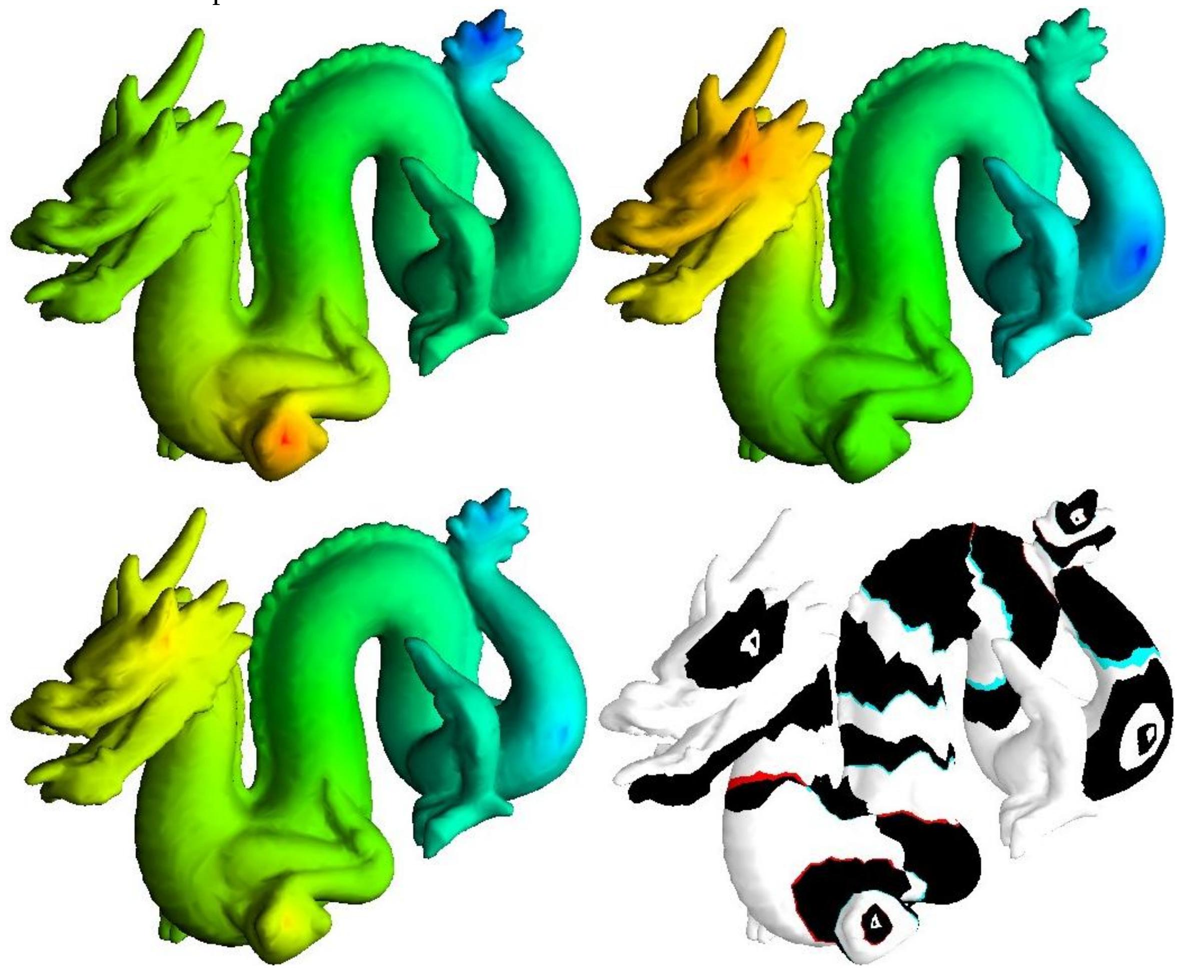


Figure 4: The top bunny model shows where the local maxima and local minima were placed on the model. The bottom left model was colored in a discontinuous fashion, and the bottom right model was colored using the continuous method described above.

3. Texture Synthesis

Use the design tool from part 2 to construct two vertex-based function (F, G). Take a periodic example texture and apply the texture to the surface by treating (F, G) as the texture coordinates.

In this part of the project, I had to learn how to do texturing in OpenGL. For a while, I struggled with even being able to open an image file—I was not sure what type of file I should use, and many of the examples I found had just used a texture that was generated by the program. Thankfully, I was able to utilize some example code provided by Dr. Zhang that would read in a .ppm file. After that, I was able to implement the rest of the code set up the texture, but I ran into another wall when I tried to texture the object. With a lot of help from Jonathan, we discovered that, even though I was calling the correct functions to set up the texture, it was not actually being bound. In order to solve the problem, we simply bind the texture right before it is applied to the model—it is a terribly inefficient method since the texture should just need to be bound once at the start of the program, but even searches online turned up no results to the problem.



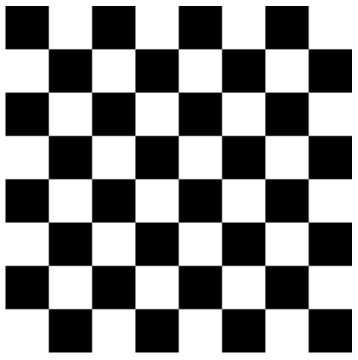


Figure 5: Top left and right models show the f and g functions that were selected while the bottom right model shows the two functions combined (simply takes the average of the two function values at each vertex). The bottom right model was textured with the texture seen to the right, and it used the function values shown in the top two models as texture coordinates.

Note that the texture was distorted greatly on the textured model in Figure 5. This is for two main reasons. First, the functions that were chosen were not exactly perpendicular to each other. By choosing two functions, the user is essentially defining an x- and a y-axis that will correspond to the u and v axes of the texture. If the functions are not perpendicular to each other, they will not vary across the model at the same rate, which can result in skewing or stretching the texture. However, the functions chosen in Figure 6 were nearly perpendicular to each other, and the same problem arose. This is because of the nature of the model itself and the particular way the u and v coordinates are being chosen. For another example, look at Figure 7. The sphere is a featureless model, and it still has distortions in the texture—even with a f and g functions that are nearly perpendicular to each other

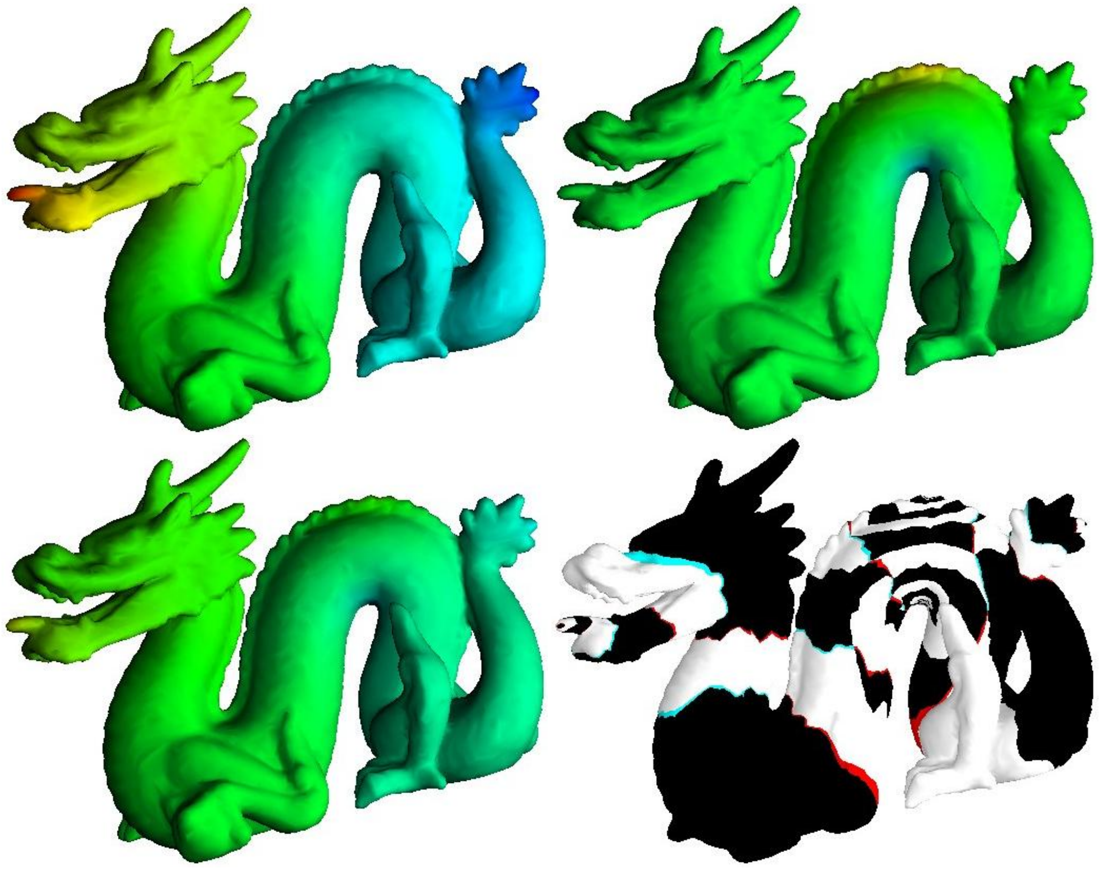


Figure 6: Top two models show the functions f and g, which are nearly perpendicular to each other. The bottom left model displays the functions together, and the bottom right model is the dragon model after it has been textured with the same texture from Figure 5, using the f and g functions displayed in this figure.

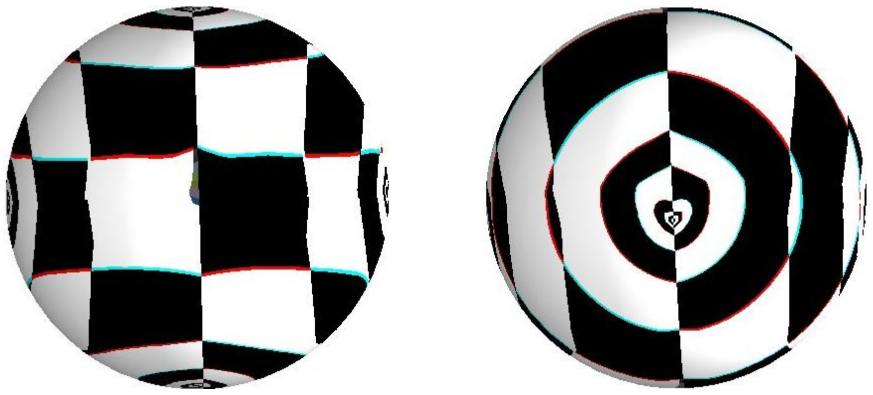


Figure : Two views of a textured sphere. The same texture used in Figures 5 and 6 was used here, and the f and g functions were nearly perpendicular.